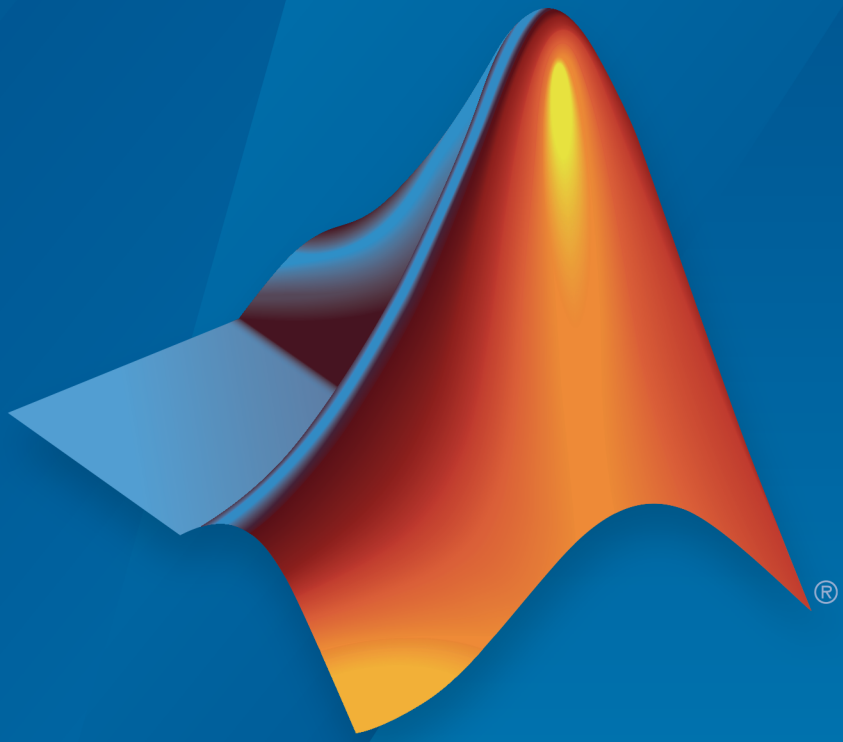
MuPAD®

Reference



MATLAB®

R2015a

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

MuPAD® Reference

Trademarks

MuPAD is a registered trademark of SciFace Software GmbH & Co. KG.
MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

September 2012	Online only	New for Version 5.9 (Release 2012b)
March 2013	Online only	Revised for Version 5.10 (Release 2013a)
September 2013	Online only	Revised for Version 5.11 (Release 2013b)
March 2014	Online only	Revised for Version 6.0 (Release 2014a)
October 2014	Online only	Revised for Version 6.1 (Release 2014b)
March 2015	Online only	Revised for Version 6.2 (Release 2015a)

Contents

1

The Standard Library

:=, _assign
., _concat
.., _range
=, _equal
<>, _unequal
~=, _approx
<, >, _less
<=, >=, _leequal
+, _plus
-, _negate
*, _mult
/, _divide
^, _power
@, _fconcat
@@, _fnest
$, _seqgen, _seqin, _seqstep
,, _exprseq
%if
;, :, _stmtseq
abs
airyAi
airyBi
alias
unalias
anames
and, _and
or, _or
not, _not
xor, _xor
==>, _implies
<=>, _equiv
append

arcsin
arccos
arctan
arccsc
arcsec
arccot
arcsinh
arccosh
arctanh
arccsch
arcsech
arccoth
arg
args
array
hfarray
assert
assign
assignElements
assume
assumeAlso
assuming, _assuming
assumingAlso, _assumingAlso
asympt
bernoulli
bernstein
bernsteinMatrix
besselI
besselJ
besselK
besselY
beta
binomial
block
blockIdents
blockTransparent
unblock
bool
break, _break
buildnumber

bytes
card
case, of, otherwise, end_case, _case
ceil
floor
round
trunc
Ci
Chi
coeff
coerce
collect
combine
complexInfinity
conjugate
contains
content
context
contfrac
copyClosure
curl
', D
dawson
debug
dedekindEta
degree
degreevec
delete, _delete
denom
densematrix
det
diff
DIGITS
dilog
dirac
discont
div, _div
divergence
divide
domtype

doprint
Ei
ellipticK
ellipticCK
ellipticF
ellipticE
ellipticCE
ellipticPi
ellipticCPi
ellipticNome
end
erf
erfc
erfi
inverf
inverfc
error
euler
eval
evalassign
|, evalAt
evalp
exp
expand
expose
expr
expr2text
extnops
extop
extsubsop
!, fact
!!, fact2
factor
factorout
FAIL
fclose
FILEPATH
finput
float
fname

fopen
for, from, to, step, end_for, _for_in, downto, _for_downto
forceGarbageCollection
forget
fourier
fourier::addpattern
fprint
frac
frandom
fread
freeIndets
freeze
unfreeze
fresnelC
fresnelS
ftextinput
funcenv
funm
gamma
lngamma
gcd
gcdex
genident
genpoly
getlasterror
getpid
getprop
gradient
ground
harmonic
has
hastype
heaviside
?, help
hessian
HISTORY
history
hold
..., hull
hypergeom

icontent
id
if, then, elif, else, end_if, _if
ifactor
ifourier
ifourier::addpattern
igamma
igcd
igcdex
ilaplace
ilaplace::addpattern
ilcm
in, _in
indets
[], _index
indexval
infinity
info
input
int
int::addpattern
int2text
interpolate
intersect, _intersect
minus, _minus
union, _union
interval
inverse
_invert
irreducible
is
isolate
isprime
isqrt
iszero
ithprime
iztrans
iztrans::addpattern
jacobiAM
jacobiSN

jacobiCN
jacobiDN
jacobiCD
jacobiSD
jacobiND
jacobiDC
jacobiNC
jacobiSC
jacobiNS
jacobiDS
jacobiCS
jacobian
jacobiZeta
kroneckerDelta
kummerU
laguerreL
lambertW
laplace
laplace::addpattern
laplacian
%, last
lasterror
_lazy_and
_lazy_or
lcm
lcoeff
ldegree
length
LEVEL
level
lhs
rhs
Li
READPATH
WRITEPATH
limit
linsolve
lllint
lmonomial
ln

log
log10
log2
lterm
match
map
mapcoeffs
maprat
matrix
max
MAXDEPTH
MAXEFFORT
MAXLEVEL
meijerG
min
mod, _mod
modp
mods
monomials
mtaylor
multcoeffs
new
newDomain
next, _next
nextprime
NIL
nops
norm
normal
simplifyFraction
NOTEBOOKFILE
NOTEBOOKPATH
nterms
nthcoeff
nthmonomial
nthterm
null
numer
O
ode

op
operator
ORDER
pade
partfrac
pathname
pdivide
piecewise
plot
display
plotfunc2d
plotfunc3d
pochhammer
poles
poly, Expr, IntMod
poly2list
polylog
potential
powermod
PRETTYPRINT
prevprime
print
->, -->, proc, name, option, local, begin, end_proc, procname
product
protect
protocol
psi
radsimp
simplifyRadical
random
rationalize
Re
Im
read
readbytes
writebytes
repeat, until, end_repeat, _repeat
while, end_while, _while
rec
rectform

rectangularPulse
rectpulse
rem
reset
return
revert
rewrite
RootOf
Rule
save, _save
select
series
Si
Ssi
Shi
sign
signIm
simplify
Simplify
sin
cos
tan
csc
sec
cot
sinh
cosh
tanh
csch
sech
coth
slot
slotAssignCounter
solve
sort
split
sqrt
strmatch
strprint
subs

subset, _subset, _notsubset
subsex
subsop
substring
_subtract
sum
sum::addpattern
surd
sysname
sysorder
system
table
taylor
tbl2text
tcoeff
testargs
testeq
testtype
text2expr
text2int
text2list
text2tbl
textinput
TEXTWIDTH
theta
rtime
time
transpose
htranspose
traperror
triangularPulse
tripulse
TRUE
FALSE
UNKNOWN
type
unassume
undefined
unit
universe

unprotect
use
unuse
val
vectorPotential
version
warning
whittakerM
whittakerW
wrightOmega
write
zeta
zip
ztrans
ztrans::addpattern

:=, _assign

Assign variables

Syntax

```
x := value
```

```
_assign(x, value)
```

```
 [x1, x2, …] := [value1, value2, …]
```

```
_assign([x1, x2, …], [value1, value2, …])
```

```
 f( X1, X2, … ) := value
```

```
_assign(f(X1, X2, …), value)
```

Description

`x := value` assigns the variable `x` a `value`.

`[x1, x2, ...] := [value1, value2, ...]` assigns the variables `x1`, `x2` etc. the corresponding values `value1`, `value2` etc.

`f(X1, X2, ...) := value` adds an entry to the remember table of the procedure `f`.

`_assign(x, value)` is equivalent to `x := value`.

`_assign([x1, x2, ...], [value1, value2, ...])` is equivalent to `[x1, x2, ...] := [value1, value2, ...]`. Both lists must have the same number of elements.

Note: If `x` is neither a list, nor a table, nor an array, nor an hfarray, nor a matrix, nor an element of a domain with a slot `"set_index"`, then an indexed assignment such as `x[i] := value` implicitly turns the identifier `x` into a table with a single entry `(i = value)`. See “Example 2” on page 1-15.

The assignment `f(X1, X2, ...) := value` adds an entry to the remember table of the procedure `f`.

Note: If `f` is neither procedure nor a function environment, then `f` is implicitly turned into a (trivial) procedure with a single entry `(X1, X2, ...) = value` in its remember table. See "Example 4" on page 1-17.

Identifiers on the left hand side of an assignment are not evaluated (use `evalassign` if this is not desired). I.e., in `x := value`, the previous value of `x`, if any, is deleted and replaced by the new value. Note, however, that the index of an indexed identifier is evaluated. I.e., in `x[i] := value`, the index `i` is replaced by its current value before the corresponding entry of `x` is assigned the value. See "Example 5" on page 1-18.

Examples

Example 1

The assignment operator `:=` can be applied to a single identifier as well as to a list of identifiers:

```
x := 42:
[x1, x2, x3] := [43, 44, 45]:
x, x1, x2, x3
```

```
42, 43, 44, 45
```

In case of lists, all variables of the left-hand side are assigned their values *simultaneously*:

```
[x1, x2] := [3, 4]:
[x1, x2] := [x2, x1]:
x1, x2
```

```
4, 3
```

The functional equivalent of the assign operator `:=` is the function `_assign`:

```
_assign(x, 13): _assign([x1, x2], [14, 15]): x, x1, x2
```

$$13,\ 14,\ 15$$

Assigned values are deleted via the keyword `delete`:

```
delete x, x1, x2:
x, x1, x2
```

$$x,\ x1,\ x2$$

Example 2

Assigning a value to an indexed identifier, a corresponding table (`table`, `DOM_TABLE`) is generated implicitly, if the identifier was not assigned a list, a table, an array, an hfarray, or a matrix before:

```
delete x:
x[1] := 7:
x
```

1	7

If `x` is a list, a table, an array, an hfarray, or a matrix, then an indexed assignment adds a further entry or changes an existing entry:

```
x[abc] := 8:
x
```

1	7
abc	8

```
x := [a, b, c, d]:
x[3] := new:
x
```

$$[a,\ b,\ \text{new},\ d]$$

```
x := array(1..2, 1..2):
```

```
x[2, 1] := value:
x
```

$$\begin{pmatrix} \text{NIL} & \text{NIL} \\ \text{value} & \text{NIL} \end{pmatrix}$$

```
delete x:
```

Example 3

For efficient use of indexed assignments (see “Example 2” on page 1-15 for an overview), programmers should note the following rules:

MuPAD® uses *reference counting* and thereby allows multiple references to identical data structures. Changing one of these logically distinct values means that the internal structure must be copied, which takes time:

```
n := 10^4:
L := [0$n]:
time((for i from 1 to n do
        L_old := L:
        L[i] := i:
      end_for))
```

```
19310
```

Compare this with the situation where only one variable or identifier refers to the internal structure:

```
n := 10^4:
L := [0$n]:
time((for i from 1 to n do
        L[i] := i:
      end_for))
```

```
60
```

For lists, there is another situation that requires copying the list structure: Changing the length of the list. The most frequently encountered example is appending to a list with `_concat` (`.`) or `append`:

```
n := 10^4:
L := []:
time((for i from 1 to n do
        L := L . [i]:
      end_for))
```

```
     13180
```

A loop written as above takes running time roughly proportional to the *square of the number of elements*. It is advisable to rewrite such loops. In the case where you know the length of the final list in advance, you can construct such a list and replace its entries inside the loop:

```
n := 10^4:
L := [NIL$n]:
time((for i from 1 to n do
        L[i] := i:
      end_for))
```

```
     60
```

If you don't know the final length, you can gain linear running time by first collecting the elements into a table:

```
n := 10^4:
T := table()
time((for i from 1 to n do
        T[nops(T)+1] := i;
      end_for;
      L := [T[i] $ i = 1..nops(T)]))
```

```
     190
```

Example 4

Consider a simple procedure:

```
f := x -> sin(x)/x:
f(0)
```

```
Error: Division by zero.
  Evaluating: f
```

The following assignment adds an entry to the remember table:

```
f(0) := 1:
f(0)
```

$$1$$

If `f` does not evaluate to a function, then a trivial procedure with a remember table is created implicitly:

```
delete f:
f(x) := x^2:
expose(f)
```

```
proc()
  name f;
  option remember;
begin
  procname(args())
end_proc
```

Note that the remember table only provides a result for the input `x`:

```
f(x), f(1.0*x), f(y)
```

$$x^2,\ f(1.0\,x),\ f(y)$$

```
delete f:
```

Example 5

The left hand side of an assignment is not evaluated. In the following, `x := 3` assigns a new value to `x`, not to `y`:

```
x := y:
x := 3:
x, y
```

$$3, y$$

Consequently, the following is not a multiple assignment to the identifiers in the list, but a single assignment to the list L:

```
L := [x1, x2]:
L := [21, 22]:
L, x1, x2
```

$$[21, 22], x1, x2$$

However, indices are evaluated in indexed assignments:

```
i := 2:
x[i] := value:
x
```

2	value

```
for i from 1 to 3 do
  x[i] := i^2;
end_for:
x
```

1	1
2	4
3	9

```
delete x, L, i:
```

Example 6

Since an assignment has a return value (the assigned value), the following command assigns values to several identifiers simultaneously:

```
a := b := c := 42:
a, b, c
```

$$42, 42, 42$$

For syntactical reasons, the inner assignment has to be enclosed by additional brackets in the following command:

```
a := sin((b := 3)):
a, b
```

$$\sin(3),\ 3$$

```
delete a, b, c:
```

Parameters

`x, x1, x2, …`

Identifiers or indexed identifiers

`value, value1, value2, …`

Arbitrary MuPAD objects

`f`

A procedure or a function environment

`X1, X2, …`

Arbitrary MuPAD objects

Return Values

`value` or `[value1, value2, ...]`, respectively.

See Also

MuPAD Functions

`anames` | `assign` | `assignElements` | `delete` | `evalassign`

., _concat

Concatenate objects

Syntax

`object`$_1$`. object`$_2$

`_concat(object`$_1$`, object`$_2$`, …)`

Description

`object_1.object_2` concatenates two objects.

`_concat( object`$_1$`, object`$_2$`, ...)` concatenates an arbitrary number of objects.

`_concat( object`$_1$`, object`$_2$`)` is equivalent to `object`$_1$`. object`$_2$. The function call `_concat( object`$_1$`, object`$_2$`, object`$_3$`, ...)` is equivalent to `(( object`$_1$`. object`$_2$`). object`$_3$`). ...`. `_concat()` returns the void object of type `DOM_NULL`.

The following combinations are possible:

object$_1$	**object$_2$**	**`object`$_1$`. object`$_2$**
string	string	string
string	identifier	string
string	integer	string
string	expression	string
identifier	string	identifier
identifier	identifier	identifier
identifier	integer	identifier
identifier	expression	identifier
list	list	list

E.g., `x.1` creates the identifier `x1`.

Note that the objects to be concatenated are evaluated before concatenation. Thus, if `x := y`, `i := 1`, the concatenation `x.i` produces the identifier `y1`. However, the resulting identifier `y1` is *not* fully evaluated. Cf. “Example 2” on page 1-22.

Examples

Example 1

We demonstrate all possible combinations of types that can be concatenated. Strings are produced if the first object is a string:

```
"x"."1", "x".y, "x".1, "x".f(a)
```

```
"x1", "xy", "x1", "xf(a)"
```

Identifiers are produced if the first object is an identifier:

```
x."1", x.y , x.1, x.f(a)
```

```
x1, xy, x1, xf(a)
```

The concatenation operator `.` also serves for concatenating lists:

```
[1, 2] . [3, 4]
```

```
[1, 2, 3, 4]
```

```
L := []: for i from 1 to 10 do L := L . [x.i] end_for: L
```

```
[x1, x2, x3, x4, x5, x6, x7, x8, x9, x10]
```

```
delete L:
```

Example 2

We demonstrate the evaluation strategy of concatenation. Before concatenation, the objects are evaluated:

```
x := "Val": i := ue: x.i
```

```
"Value"
```

```
ue := 1:  x.i
```

```
"Val1"
```

An identifier produced via concatenation is not fully evaluated:

```
delete x: x1 := 17: x.1, eval(x.1)
```

```
x1, 17
```

The `.` operator can be used to create variables dynamically. They can be assigned values immediately:

```
delete x: for i from 1 to 5 do x.i := i^2 end_for:
```

Again, the result of the concatenation is not fully evaluated:

```
x.i $ i= 1..5
```

```
x1, x2, x3, x4, x5
```

```
eval(%)
```

```
1, 4, 9, 16, 25
```

```
delete i, ue: (delete x.i) $ i = 1..5:
```

Example 3

The function `_concat` can be used to concatenate an arbitrary number of objects:

```
_concat("an", " ", "ex", "am", "ple")
```

```
"an example"
```

```
_concat("0", " ".i $ i = 1..15)
```

```
"0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15"
```

```
_concat([], [x.i] $ i = 1..10)
```

```
[x1, x2, x3, x4, x5, x6, x7, x8, x9, x10]
```

Parameters

$object_1$

A character string, an identifier, or a list

$object_2$, …

A character string, an identifier, an integer, a list, or an expression

Return Values

Object of the same type as $object_1$.

Overloaded By

object_1, object_2

See Also

MuPAD Functions
`@` | `append`

.., _range

Range operator

Syntax

```
l .. r
```

```
_range(l, r)
```

Description

`l..r` defines a "range" with the left bound `l` and the right bound `r`.

A range is a technical construct that is used to specify ranges of numbers when calling various system functions such as `int`, `array`, `op`, or the sequence operator `$`. Usually, `l..r` represents a real interval (e.g., `int(f(x), x = l..r)`), or the sequence of integers from `l` to `r`.

`_range(l, r)` is equivalent to `l..r`.

To create and operate on intervals in a mathematical sense, use the data type `Dom::Interval`.

Examples

Example 1

A range can be defined with the `..` operator as well as with a call to the function `_range`:

```
_range(1, 42), 1..42
```

```
1..42, 1..42
```

In the following call, the range represents an interval:

```
int(x, x = l..r)
```

$$\frac{r^2}{2} - \frac{l^2}{2}$$

Ranges can be used for accessing the operands of expressions or to define the size of arrays and hfarrays:

```
op(f(a, b, c, d, e), 2..4)
```

$$b, c, d$$

```
array(1..3, [a1, a2,a3])
```

$$(\,a1\;\;a2\;\;a3\,)$$

```
hfarray(1..3, 1..2)
```

$$\begin{pmatrix} 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \end{pmatrix}$$

Ranges can also be used for creating expression sequences:

```
i^3 $ i = 1..5
```

$$1, 8, 27, 64, 125$$

Example 2

The range operator `..` is a technical device that does not check its parameters with respect to their semantics. It just creates a range which is interpreted in the context in which it is used later. Any bounds are accepted:

```
float(PI) .. -sqrt(2)/3
```

$$3.141592654.. - \frac{\sqrt{2}}{3}$$

Parameters

`l, r`

Arbitrary MuPAD objects

Return Values

Expression of type `"_range"`.

Overloaded By

See Also

MuPAD Domains
`Dom::Interval`

MuPAD Functions
`$`

=, _equal

Equations (equal)

Syntax

```
x = y

_equal(x, y)
```

Description

`x = y` defines an equation.

`x = y` is equivalent to the function call `_equal(x, y)`.

The operator = returns a symbolic expression representing an equation.

The resulting expression can be evaluated to `TRUE` or `FALSE` by the function `bool`. It also serves as control conditions in `if`, `repeat`, and `while` statements. In all these cases, testing for equality is a purely syntactical test. E.g., `bool(0.5 = 1/2)` returns `FALSE` although both numbers coincide numerically.

Further, Boolean expressions can be evaluated to `TRUE`, `FALSE`, or `UNKNOWN` by the function `is`. Tests using `is` are semantical comparing `x` and `y` subject to mathematical considerations.

Equations have two operands: the left hand side and the right hand side. One may use `lhs` and `rhs` to extract these operands.

The boolean expression `not x = y` is always converted to `x <> y`.

The expression `not x <> y` is always converted to `x = y`.

Examples

Example 1

In the following, note the difference between syntactical and numerical equality. The numbers 1.5 and $\frac{3}{2}$ coincide numerically. However, 1.5 is of domain type `DOM_FLOAT`, whereas $\frac{3}{2}$ is of domain type `DOM_RAT`. Consequently, they are not regarded as equal in the following syntactical test:

```
1.5 = 3/2; bool(%)
```

$$1.5 = \frac{3}{2}$$

$$\text{FALSE}$$

If floating-point numbers are involved, one should rather use the operator `~=` instead of `=`. The functions `bool` and `is` test whether the floating-point approximations coincide up to the relative precision given by `DIGITS`:

```
1.5 ~= 3/2;
bool(1.5 ~= 3/2);
is(1.5 ~= 3/2);
```

$$1.5 \approx \frac{3}{2}$$

$$\text{TRUE}$$

$$\text{TRUE}$$

The following expressions coincide syntactically:

```
_equal(1/x, diff(ln(x),x)); bool(%)
```

$$\frac{1}{x} = \frac{1}{x}$$

```
TRUE
```

The Boolean operator `not` converts equalities and inequalities:

```
not a = b, not a <> b
```

$$a \neq b,\ a = b$$

Example 2

The examples below demonstrate how = and <> deal with non-mathematical objects and data structures:

```
if "text" = "t"."e"."x"."t" then "yes" else "no" end
```

```
"yes"
```

```
bool(table(a = PI) <> table(a = sqrt(2)))
```

```
TRUE
```

Example 3

We demonstrate the difference between the syntactical test via `bool` and the semantical test via `testeq`:

```
bool(1 = x/(x + y) + y/(x + y)), testeq(1 = x/(x + y) + y/(x + y))
```

```
FALSE, TRUE
```

Example 4

Equations and inequalities are typical input objects for system functions such as `solve`:

```
solve(x^2 - 2*x = -1, x)
```

```
{1}
```

```
solve(x^2 - 2*x <> -1, x)
```

$$\mathbb{C} \setminus \{1\}$$

Parameters

x, y

Arbitrary MuPAD objects

Return Values

Expression of type `"_equal"`.

See Also

MuPAD Functions

`<` | `<=` | `<>` | `>` | `>=` | `and` | `bool` | `FALSE` | `if` | `lhs` | `not` | `or` | `repeat` | `rhs` | `solve` | `testeq` | `TRUE` | `UNKNOWN` | `while` | `~=`

<>, _unequal

Inequalities (unequal)

Syntax

```
x <> y

_unequal(x, y)
```

Description

`x <> y` defines an inequality.

`x <> y` is equivalent to the function call `_unequal(x, y)`.

The operator `<>` returns a symbolic expression representing an inequality.

The resulting expression can be evaluated to `TRUE` or `FALSE` by the function `bool`. It also serves as control conditions in `if`, `repeat`, and `while` statements. In all these cases, testing for equality or inequality is a purely syntactical test. E.g., `bool(0.5 <> 1/2)` returns `TRUE` although both numbers coincide numerically.

Further, Boolean expressions can be evaluated to `TRUE`, `FALSE`, or `UNKNOWN` by the function `is`. Tests using `is` are semantical comparing `x` and `y` subject to mathematical considerations.

Inequalities have two operands: the left hand side and the right hand side. One may use `lhs` and `rhs` to extract these operands.

The boolean expression `not x = y` is always converted to `x <> y`.

The expression `not x <> y` is always converted to `x = y`.

Examples

Example 1

In the following, note the difference between syntactical and numerical equality. The numbers 1.5 and $\frac{3}{2}$ coincide numerically. However, 1.5 is of domain type `DOM_FLOAT`, whereas $\frac{3}{2}$ is of domain type `DOM_RAT`. Consequently, they are not regarded as equal in the following syntactical test:

```
1.5 = 3/2; bool(%)
```

$$1.5 = \frac{3}{2}$$

$$\text{FALSE}$$

If floating-point numbers are involved, one should rather use the operator `~=` instead of `=`. The functions `bool` and `is` test whether the floating-point approximations coincide up to the relative precision given by `DIGITS`:

```
1.5 ~= 3/2;
bool(1.5 ~= 3/2);
is(1.5 ~= 3/2);
```

$$1.5 \approx \frac{3}{2}$$

$$\text{TRUE}$$

$$\text{TRUE}$$

The following expressions coincide syntactically:

```
_equal(1/x, diff(ln(x),x)); bool(%)
```

$$\frac{1}{x} = \frac{1}{x}$$

```
TRUE
```

The Boolean operator `not` converts equalities and inequalities:

```
not a = b, not a <> b
```

$$a \neq b,\ a = b$$

Example 2

The examples below demonstrate how = and <> deal with non-mathematical objects and data structures:

```
if "text" = "t"."e"."x"."t" then "yes" else "no" end
```

```
"yes"
```

```
bool(table(a = PI) <> table(a = sqrt(2)))
```

```
TRUE
```

Example 3

We demonstrate the difference between the syntactical test via `bool` and the semantical test via `testeq`:

```
bool(1 = x/(x + y) + y/(x + y)), testeq(1 = x/(x + y) + y/(x + y))
```

```
FALSE, TRUE
```

Example 4

Equations and inequalities are typical input objects for system functions such as `solve`:

```
solve(x^2 - 2*x = -1, x)
```

```
{1}
```

```
solve(x^2 - 2*x <> -1, x)
```

$$\mathbb{C} \setminus \{1\}$$

Parameters

x, y

Arbitrary MuPAD objects

Return Values

Expression of type `"_unequal"`.

See Also

MuPAD Functions

`<` | `<=` | `=` | `>` | `>=` | `and` | `bool` | `FALSE` | `if` | `lhs` | `not` | `or` | `repeat` | `rhs` | `solve` | `testeq` | `TRUE` | `UNKNOWN` | `while` | `~=`

~=, _approx

Approximate equality

Syntax

```
x ~= y
```

```
_approx(x, y)
```

Description

`x ~= y` symbolizes approximate equality.

`x ~= y` is equivalent to the function call `_approx(x, y)`.

The operator ~= returns a symbolic expression representing an approximate equality for numerical values `x` and `y`. The calls `bool(x ~= y)` and `is(x ~= y)` check whether `|float((x - y)/x)| < 10^(-DIGITS)` is satisfied, provided `x ≠ 0` and `y ≠ 0`. Thus, `TRUE` is returned if `x` and `y` coincide within the relative numerical precision set by `DIGITS`. For `x = 0`, the criterion is `|float(y)| < 10^(-DIGITS)`. For `y = 0`, the criterion is `|float(x)| < 10^(-DIGITS)`. If either `x` or `y` contains a symbolic object that cannot be converted to a real or complex floating point number, the functions `bool` and `is` return the value `UNKNOWN`.

Approximate equalities have two operands: the left hand side and the right hand side. One may use `lhs` and `rhs` to extract these operands.

Note: `a ~= b` is not equivalent to `a - b ~= 0`.

Examples

Example 1

In the following, note the difference between syntactical and numerical equality. The numbers 1.5 and $\frac{3}{2}$ coincide numerically. However, 1.5 is of domain type `DOM_FLOAT`,

whereas $\frac{3}{2}$ is of domain type DOM_RAT. Consequently, they are not regarded as equal in the following syntactical test:

```
1.5 = 3/2; bool(%)
```

$$1.5 = \frac{3}{2}$$

$$\text{FALSE}$$

If floating-point numbers are involved, one should rather use the operator ~= instead of =. The functions bool and is test whether the floating-point approximations coincide up to the relative precision given by DIGITS:

```
1.5 ~= 3/2;
bool(1.5 ~= 3/2);
is(1.5 ~= 3/2);
```

$$1.5 \approx \frac{3}{2}$$

$$\text{TRUE}$$

$$\text{TRUE}$$

The following expressions coincide syntactically:

```
_equal(1/x, diff(ln(x),x)); bool(%)
```

$$\frac{1}{x} = \frac{1}{x}$$

$$\text{TRUE}$$

The Boolean operator not converts equalities and inequalities:

```
not a = b, not a <> b
```

$$a \neq b,\ a = b$$

Example 2

The examples below demonstrate how = and <> deal with non-mathematical objects and data structures:

```
if "text" = "t"."e"."x"."t" then "yes" else "no" end
```

"yes"

```
bool(table(a = PI) <> table(a = sqrt(2)))
```

TRUE

Example 3

We demonstrate the difference between the syntactical test via `bool` and the semantical test via `testeq`:

```
bool(1 = x/(x + y) + y/(x + y)), testeq(1 = x/(x + y) + y/(x + y))
```

FALSE, TRUE

Example 4

Equations and inequalities are typical input objects for system functions such as `solve`:

```
solve(x^2 - 2*x = -1, x)
```

{1}

```
solve(x^2 - 2*x <> -1, x)
```

$\mathbb{C} \setminus \{1\}$

Parameters

x, y

Arbitrary MuPAD objects

Return Values

Expression of type `"_approx"`.

See Also

MuPAD Functions

`<` | `<=` | `<>` | `=` | `>` | `>=` | `and` | `bool` | `FALSE` | `if` | `lhs` | `not` | `or` | `repeat` | `rhs` | `solve` | `testeq` | `TRUE` | `UNKNOWN` | `while`

<, >, _less

Inequalities “less than” and “greater than”

Syntax

```
x < y

x > y

_less(x, y)
```

Description

`x < y` and `x > y` define inequalities.

`x < y` represents the Boolean statement “`x` is less than `y`”. It is equivalent to the function call `_less(x,y)`.

`x > y` represents the Boolean statement “`x` is greater than `y`”. It is always converted to `y < x`, which is equivalent to the function call `_less(y,x)`.

These operators return symbolic Boolean expressions. If only real numbers of `Type::Real` are involved, these expressions can be evaluated to `TRUE` or `FALSE` by the function `bool`. They also serve as control conditions in `if`, `repeat`, and `while` statements. For floating-point intervals, these operators are interpreted as “strictly smaller than” and so on, see “Example 2” on page 1-41.

Further, Boolean expressions can be evaluated to `TRUE`, `FALSE`, or `UNKNOWN` by the function `is`. Tests using `is` can also be applied to constant symbolic expressions. See “Example 4” on page 1-42.

`bool` also handles inequalities involving character strings. It compares them with respect to the lexicographical ordering.

Inequalities have two operands: the left hand side and the right hand side. One may use `lhs` and `rhs` to extract these operands.

Examples

Example 1

The operators <, <=, >, and >= produce symbolic inequalities. They can be evaluated to TRUE or FALSE by the function `bool` if only real numbers of type `Type::Real` (integers, rationals, and floats) are involved:

```
1.5 <= 3/2; bool(%)
```

$$1.5 \le \frac{3}{2}$$

TRUE

Note that `bool` may fail to handle Boolean expressions that involve exact expressions, even if they represent real numbers:

```
_less(PI,  sqrt(2) + 17/10); bool(%)
```

$$\pi < \sqrt{2} + \frac{17}{10}$$

FALSE

```
bool(sqrt(6) < sqrt(2)*sqrt(3))
```

```
Error: Cannot evaluate to Boolean. [_less]
```

Example 2

Comparison of intervals is interpreted as “strict”, that is, all combinations of numbers in the intervals must fulfill the relation:

```
bool(0...1 < 2...3), bool(0...2 < 1...3),
bool(0...1 < 1...2)
```

TRUE, FALSE, FALSE

```
bool(0...1 <= 2...3), bool(0...2 <= 1...3),
bool(0...1 <= 1...2)
```

```
TRUE, FALSE, TRUE
```

Example 3

This examples demonstrates how character strings can be compared:

```
if "text" < "t"."e"."x"."t"."book" then "yes" else "no" end
```

```
"yes"
```

```
bool("a" >= "b")
```

```
FALSE
```

Example 4

Note that `bool` does not perform symbolic simplification and therefore cannot handle some combinations of symbolic expressions; the function `is` does perform symbolic simplification:

```
bool(sqrt(6) < sqrt(2)*sqrt(3))
```

```
Error: Cannot evaluate to Boolean. [_less]
```

```
is(sqrt(6) < sqrt(2)*sqrt(3))
```

```
FALSE
```

Example 5

Inequalities are valid input objects for the system function `solve`:

```
solve(x^2 - 2*x < 3, x)
```

$$(-1, 3) \cup \{1 + y\,i \mid y \in \mathbb{R}\}$$

```
solve(x^2 - 2*x >= 3, x)
```

$$(-\infty, -1] \cup [3, \infty)$$

Example 6

The operators < and <= can be overloaded by user-defined domains:

```
myDom := newDomain("myDom"): myDom::print := x -> extop(x):
```

Without overloading `_less` or `_leequal`, elements of this domain cannot be compared:

```
x := new(myDom, PI): y := new(myDom, sqrt(10)): bool(x < y)
```

```
Error: Cannot evaluate to Boolean. [_less]
```

Now, a slot`"_less"` is defined. It is called, when an inequality of type `"_less"` is evaluated by `bool`. The slot compares floating-point approximations if the arguments are not of type `Type::Real`:

```
myDom::_less := proc(x, y)
begin
     x := extop(x, 1):
     y := extop(y, 1):
     if not testtype(x, Type::Real) then
        x := float(x):
        if not testtype(x, Type::Real) then
           error("cannot compare")
        end_if
     end_if:
     if not testtype(y, Type::Real) then
        y := float(y):
        if not testtype(y, Type::Real) then
           error("cannot compare")
        end_if
     end_if:
     bool(x < y)
end_proc:
```

```
x, y, bool(x < y), bool(x > y)
```

$$\pi,\ \sqrt{10},\ \text{TRUE},\ \text{FALSE}$$

```
bool(new(myDom, I) < new(myDom, PI))
```

```
Error: cannot compare [myDom::_less]
```

```
delete myDom, x, y:
```

Parameters

x, y

Arbitrary MuPAD objects

Return Values

Expression of type `"_less"`.

Overloaded By

x, y

See Also

MuPAD Functions

`<=` | `<>` | `=` | `>=` | `and` | `bool` | `FALSE` | `if` | `lhs` | `not` | `or` | `repeat` | `rhs` | `solve` | `TRUE` | `UNKNOWN` | `while`

<=, >=, _leequal

Inequalities “less than or equal to” and “greater than or equal to”

Syntax

```
x <= y

x >= y

_leequal(x, y)
```

Description

`x <= y` and `x >= y` define inequalities.

`x <= y` represents the Boolean statement “`x` is less than or equal to `y`”. It is equivalent to the function call `_leequal(x,y)`.

`x >= y` represents the Boolean statement “`x` is greater than or equal to `y`”. It is always converted to `y <= x`, which is equivalent to the function call `_leequal(y,x)`.

These operators return symbolic Boolean expressions. If only real numbers of `Type::Real` are involved, these expressions can be evaluated to `TRUE` or `FALSE` by the function `bool`. They also serve as control conditions in `if`, `repeat`, and `while` statements. For floating-point intervals, these operators are interpreted as “strictly smaller than” and so on, see “Example 2” on page 1-46.

Further, Boolean expressions can be evaluated to `TRUE`, `FALSE`, or `UNKNOWN` by the function `is`. Tests using `is` can also be applied to constant symbolic expressions. See “Example 4” on page 1-47.

`bool` also handles inequalities involving character strings. It compares them with respect to the lexicographical ordering.

Inequalities have two operands: the left hand side and the right hand side. One may use `lhs` and `rhs` to extract these operands.

Examples

Example 1

The operators <, <=, >, and >= produce symbolic inequalities. They can be evaluated to TRUE or FALSE by the function `bool` if only real numbers of type `Type::Real` (integers, rationals, and floats) are involved:

```
1.5 <= 3/2; bool(%)
```

$$1.5 \leq \frac{3}{2}$$

TRUE

Note that `bool` may fail to handle Boolean expressions that involve exact expressions, even if they represent real numbers:

```
_less(PI,  sqrt(2) + 17/10); bool(%)
```

$$\pi < \sqrt{2} + \frac{17}{10}$$

FALSE

```
bool(sqrt(6) < sqrt(2)*sqrt(3))
```

```
Error: Cannot evaluate to Boolean. [_less]
```

Example 2

Comparison of intervals is interpreted as “strict”, that is, all combinations of numbers in the intervals must fulfill the relation:

```
bool(0...1 < 2...3), bool(0...2 < 1...3),
bool(0...1 < 1...2)
```

TRUE, FALSE, FALSE

```
bool(0...1 <= 2...3), bool(0...2 <= 1...3),
bool(0...1 <= 1...2)
```

```
TRUE, FALSE, TRUE
```

Example 3

This examples demonstrates how character strings can be compared:

```
if "text" < "t"."e"."x"."t"."book" then "yes" else "no" end
```

```
"yes"
```

```
bool("a" >= "b")
```

```
FALSE
```

Example 4

Note that `bool` does not perform symbolic simplification and therefore cannot handle some combinations of symbolic expressions; the function `is` does perform symbolic simplification:

```
bool(sqrt(6) < sqrt(2)*sqrt(3))
```

```
Error: Cannot evaluate to Boolean. [_less]
```

```
is(sqrt(6) < sqrt(2)*sqrt(3))
```

```
FALSE
```

Example 5

Inequalities are valid input objects for the system function `solve`:

```
solve(x^2 - 2*x < 3, x)
```

$$(-1, 3) \cup \{1 + y\,i \mid y \in \mathbb{R}\}$$

```
solve(x^2 - 2*x >= 3, x)
```

$$(-\infty, -1] \cup [3, \infty)$$

Example 6

The operators < and <= can be overloaded by user-defined domains:

```
myDom := newDomain("myDom"): myDom::print := x -> extop(x):
```

Without overloading `_less` or `_leequal`, elements of this domain cannot be compared:

```
x := new(myDom, PI): y := new(myDom, sqrt(10)): bool(x < y)
```

```
Error: Cannot evaluate to Boolean. [_less]
```

Now, a slot`"_less"` is defined. It is called, when an inequality of type `"_less"` is evaluated by `bool`. The slot compares floating-point approximations if the arguments are not of type `Type::Real`:

```
myDom::_less := proc(x, y)
begin
     x := extop(x, 1):
     y := extop(y, 1):
     if not testtype(x, Type::Real) then
        x := float(x):
        if not testtype(x, Type::Real) then
           error("cannot compare")
        end_if
     end_if:
     if not testtype(y, Type::Real) then
        y := float(y):
        if not testtype(y, Type::Real) then
           error("cannot compare")
        end_if
     end_if:
     bool(x < y)
end_proc:
```

```
x, y, bool(x < y), bool(x > y)
```

$$\pi,\ \sqrt{10},\ \text{TRUE},\ \text{FALSE}$$

```
bool(new(myDom, I) < new(myDom, PI))
```

```
Error: cannot compare [myDom::_less]
```

```
delete myDom, x, y:
```

Parameters

x, y

Arbitrary MuPAD objects

Return Values

Expression of type `"_leequal"`.

Overloaded By

x, y

See Also

MuPAD Functions

`<` | `<>` | `=` | `>` | `and` | `bool` | `FALSE` | `if` | `lhs` | `not` | `or` | `repeat` | `rhs` | `solve` | `TRUE` | `UNKNOWN` | `while`

+, _plus

Add expressions

Syntax

```
x + y + ...
```

```
_plus(x, y, …)
```

Description

`x + y + ...` computes the sum of `x`, `y` etc.

`x + y + ...` is equivalent to the function call `_plus(x, y, ...)`.

All terms that are numbers of type `Type::Numeric` are automatically combined to a single number.

Terms of a symbolic sum may be rearranged internally. Cf. “Example 1” on page 1-51. The user can control the ordering by the preference `Pref::keepOrder`. See also the documentation for `print`.

`_plus` accepts an arbitrary number of arguments. In conjunction with the sequence operator `$`, this function is the recommended tool for computing finite sums. Cf. “Example 2” on page 1-52. The function `sum` may also serve for computing such sums. However, `sum` is designed for the computation of symbolic and infinite sums. It is slower than `_plus`.

`x - y` is internally represented as `x + y*(-1)` = `_plus(x, _mult(y, -1))`. See `_subtract` for details.

For adding equalities, inequalities, and comparisons, the following rules are implemented:

- Adding an arithmetical expression adds the expression to both sides.
- Adding an equality adds the left hand sides and the right hand sides separately.
- Adding a comparison does likewise, taking care of the correct operator. Adding a comparison to an inequality is not permitted.

Cf. “Example 4” on page 1-53.

Many library domains overload `_plus` by an appropriate slot`"_plus"`. Sums involving elements of library domains are processed as follows:

A sum `x + y + ...` is searched for elements of library domains from left to right. Let `z` be the first term that is not of one of the basic types provided by the kernel (numbers, expressions, etc.). If the domain `d` = `z::dom` = `domtype(z)` has a slot`"_plus"`, it is called in the form `d::_plus(x, y, ...)`. The result returned by `d::_plus` is the result of `x + y + ...`.

Users should implement the slot `d::_plus` of their domains `d` according to the following convention:

- If all terms are elements of `d`, an appropriate sum of type `d` should be returned.
- If at least one term cannot be converted to an element of `d`, the slot should return `FAIL`.
- Care must be taken if there are terms that are not of type `d`, but can be converted to type `d`. Such terms should be converted only if the mathematical semantics is obvious to any user who uses this domain as a 'black box' (e.g., integers may be regarded as rational numbers because of the natural mathematical embedding). If in doubt, the `"_plus"` method should return `FAIL` instead of using implicit conversions. If implicit conversions are used, they must be well-documented.

Cf. “Example 6” on page 1-55 and “Example 7” on page 1-56.

Most of the library domains in the MuPAD standard installation comply with this convention.

`_plus()` returns the number 0.

Polynomials of type `DOM_POLY` are added by `+`, if they have the same indeterminates and the same coefficient ring.

For finite sets `X`, `Y`, the sum `X + Y` is the set $\{x+y \mid x \in X, y \in Y\}$.

Examples

Example 1

Numerical terms are simplified automatically:

```
3 + x + y + 2*x + 5*x - 1/2 - sin(4) + 17/4
```

$$8\,x + y - \sin(4) + \frac{27}{4}$$

The ordering of the terms of a sum is not necessarily the same as on input:

```
x + y + z + a + b + c
```

$$a + b + c + x + y + z$$

```
1 + x + x^2 + x^10
```

$$x^{10} + x^2 + x + 1$$

Internally, this sum is a symbolic call of `_plus`:

```
op(%, 0), type(%)
```

$$\text{\_plus},\ \text{"\_plus"}$$

Example 2

The functional equivalent `_plus` of the operator + is a handy tool for computing finite sums. In the following, the terms are generated via the sequence operator $:

```
_plus(i^2 $ i = 1..100)
```

$$338350$$

E.g., it is easy to add up all elements in a set:

```
S := {a, b, 1, 2, 27}: _plus(op(S))
```

$$a + b + 30$$

The following command “zips” two lists by adding corresponding elements:

```
L1 := [a, b, c]: L2 := [1, 2, 3]: zip(L1, L2, _plus)
```

$$[a+1,\ b+2,\ c+3]$$

```
delete S, L1, L2:
```

Example 3

Polynomials of type `DOM_POLY` are added by `+`, if they have the same indeterminates and the same coefficient ring:

```
poly(x^2 + 1, [x]) + poly(x^2 + x - 1, [x])
```

$$\mathrm{poly}(2\,x^2+x,\ [x])$$

If the indeterminates or the coefficient rings do not match, `_plus` returns an error:

```
poly(x, [x]) + poly(x, [x, y])
```

```
Error: The argument is invalid. [_plus]
```

```
poly(x, [x]) + poly(x, [x], Dom::Integer)
```

```
Error: The argument is invalid. [_plus]
```

Example 4

Adding a constant to an equality, an inequality, or a comparison amounts to adding it to both sides:

```
(a = b) + c, (a <> b) + c, (a <= b) + c, (a < b) + c
```

$$a+c=b+c,\ a+c\neq b+c,\ a+c\leq b+c,\ a+c<b+c$$

Adding an equality is performed by adding the left hand sides and the right hand sides separately:

```
(a = b) + (c = d), (a <> b) + (c = d),
```

```
(a <= b) + (c = d), (a < b) + (c = d)
```

$$a+c=b+d,\ a+c\neq b+d,\ a+c\leq b+d,\ a+c<b+d$$

Inequalities can only be added to equalities:

```
(a = b) + (c <> d), (a <> b) + (c <> d),
(a <= b) + (c <> d), (a < b) + (c <> d)
```

$$a+c\neq b+d,\ \text{FAIL},\ \text{FAIL},\ \text{FAIL}$$

The addition of comparisons takes of the difference between < and <= into account. Note that MuPAD uses only these two comparison operators; $a > b$ and $a \geq b$ are automatically rewritten:

```
(a = b) + (c <= d), (a <> b) + (c <= d),
(a <= b) + (c <= d), (a < b) + (c <= d);
```

$$a+c\leq b+d,\ \text{FAIL},\ a+c\leq b+d,\ a+c<b+d$$

```
(a = b) + (c < d), (a <> b) + (c < d),
(a <= b) + (c < d), (a < b) + (c < d);
```

$$a+c<b+d,\ \text{FAIL},\ a+c<b+d,\ a+c<b+d$$

```
(a = b) + (c >= d), (a <> b) + (c >= d),
(a <= b) + (c >= d), (a < b) + (c >= d);
```

$$a+d\leq b+c,\ \text{FAIL},\ a+d\leq b+c,\ a+d<b+c$$

```
(a = b) + (c > d), (a <> b) + (c > d),
(a <= b) + (c > d), (a < b) + (c > d);
```

$$a+d<b+c,\ \text{FAIL},\ a+d<b+c,\ a+d<b+c$$

Example 5

For finite sets X, Y, the sum X + Y is the set $\{x+y \mid x \in X,\ y \in Y\}$:

```
{a, b, c} + {1, 2}
```

$$\{a+1, a+2, b+1, b+2, c+1, c+2\}$$

Example 6

Various library domains such as matrix domains overload `_plus`:

```
x := Dom::Matrix(Dom::Integer)([1, 2]):
y := Dom::Matrix(Dom::Rational)([2, 3]):
x + y, y + x
```

$$\begin{pmatrix} 3 \\ 5 \end{pmatrix}, \begin{pmatrix} 3 \\ 5 \end{pmatrix}$$

If the terms in a sum `x + y` are of different type, the first term `x` tries to convert `y` to the data type of `x`. If successful, the sum is of the same type as `x`. In the previous example, `x` and `y` have different types (both are matrices, but the component domains differ). Hence the sums `x + y` and `y + x` differ syntactically, because they inherit their type from the first term:

```
bool(x + y = y + x)
```

FALSE

```
domtype(x + y), domtype(y + x)
```

Dom::Matrix(Dom::Integer), Dom::Matrix(Dom::Rational)

If `x` does not succeed to convert `y`, then `FAIL` is returned. In the following call, the component `2/3` cannot be converted to an integer:

```
y := Dom::Matrix(Dom::Rational)([2/3, 3]): x + y
```

FAIL

```
delete x, y:
```

Example 7

This example demonstrates how to implement a slot `"_plus"` for a domain. The following domain `myString` is to represent character strings. The sum of such strings is to be the concatenation of the strings.

The `"new"` method uses `expr2text` to convert any MuPAD object to a string. This string is the internal representation of elements of `myString`. The `"print"` method turns this string into the screen output:

```
myString := newDomain("myString"):
myString::new := proc(x)
begin
  if args(0) = 0 then x := "": end_if;
  case domtype(x)
    of myString do return(x);
    of DOM_STRING do return(new(dom, x));
    otherwise return(new(dom, expr2text(x)));
  end_case
end_proc:
myString::print := x -> extop(x, 1):
```

Without a `"_plus"` method, the system function `_plus` handles elements of this domain like any symbolic object:

```
y := myString(y): z := myString(z): 1 + x + y + z + 3/2
```

$$x + y + z + \frac{5}{2}$$

Now, we implement the `"_plus"` method. It checks all arguments. Arguments are converted, if they are not of type `myString`. Generally, such an implicit conversion should be avoided. In this case, however, any object has a corresponding string representation via `expr2text` and an implicit conversion is implemented. Finally, the sum of `myString` objects is defined as the concatenation of the internal strings:

```
myString::_plus := proc()
local n, Arguments, i;
begin
  print(Unquoted, "Info: myString::_plus called with the arguments:",
           args()):
  n := args(0):
```

```
  Arguments := [args()];
  for i from 1 to n do
    if domtype(Arguments[i]) <> myString then
      // convert the i-th term to myString
      Arguments[i] := myString::new(Arguments[i]):
    end_if;
  end_for:
  myString::new(_concat(extop(Arguments[i], 1) $ i = 1..n))
end_proc:
```

Now, `myString` objects can be added:

```
myString("This ") + myString("is ") + myString("a string")
```

```
Info: myString::_plus called with the arguments:, This , is , a string
```

This is a string

In the following sum, `y` and `z` are elements of `myString`. The term `y` is the first term that is an element of a library domain. Its `"_plus"` method is called and concatenates all terms to a string of type `myString`:

```
1 + x + y + z + 3/2;
```

```
                                                                  3
Info: myString::_plus called with the arguments:, 1, x, y, z, -
                                                                  2
```

1xyz3/2

```
delete myString, y, z:
```

Parameters

`x, y, …`

arithmetical expressions, polynomials of type `DOM_POLY`, sets, equations, inequalities, or comparisons

Return Values

Arithmetical expression, a polynomial, a set, an equation, an inequality, or a comparison.

Overloaded By

x, y

See Also

MuPAD Functions

`*` | `-` | `/` | `^` | `_invert` | `_subtract` | `poly` | `Pref::keepOrder` | `sum`

-, _negate

Negative of an expression

Syntax

```
- x
```

```
_negate(x)
```

Description

`- x` computes the negative of `x`.

`-x` is equivalent to the function call `_negate(x)`. Both calls represent the inverse of the element `x` of an additive group.

The negative of a number of type `Type::Numeric` is also a number.

If `x` is an element of a domain that does not have the `_negate` method (slot), MuPAD internally represents `-x` as `x*(-1) = _mult(x, -1)`.

If `x` is an element of a domain that has the `_negate` method (slot), MuPAD uses this method to compute `-x`.

The difference `x - y` is equivalent to `x + (-y) = _plus (x, _negate(y))`.

The negative of a polynomial of type `DOM_POLY` produces a polynomial. The coefficients of the resulting polynomial are the negatives of the original coefficients.

For finite sets, `-X` is the set $\{-x \mid x \in X\}$.

Examples

Example 1

Compute the negatives of the following expressions. The negative of an expression is the inverse with respect to + (`_plus`):

```
x - x = x + _negate(x)
```

$$0 = 0$$

```
-1 + x - 2*x + 23
```

$$22 - x$$

Example 2

Internally, MuPAD represents `-x` as `_mult(x, -1)`:

```
type(-x), op(-x, 0), op(-x, 1), op(-x, 2)
```

$$\text{"\_mult"}, \_mult, x, -1$$

Example 3

Compute the negative of a polynomial. The result is a polynomial with the coefficients that are the negatives of the coefficients of the original polynomial:

```
-poly(x^2 + x - 1, [x])
```

$$\text{poly}(-x^2 - x + 1, [x])$$

```
-poly(x, [x], Dom::Integer)
```

$$\text{poly}(-x, [x], \text{Dom::Integer})$$

Example 4

Compute the negative of a finite set. For finite sets, `-X` is the set $\{-x \mid x \in X\}$:

```
-{a, b, c}
```

$$\{-a, -b, -c\}$$

Example 5

Various library domains such as matrix domains or residue class domains overload `_negate`:

```
x := Dom::Matrix(Dom::IntegerMod(7))([2, 10]): x, -x, x + (-x)
```

$$\begin{pmatrix} 2 \bmod 7 \\ 3 \bmod 7 \end{pmatrix}, \begin{pmatrix} 5 \bmod 7 \\ 4 \bmod 7 \end{pmatrix}, \begin{pmatrix} 0 \bmod 7 \\ 0 \bmod 7 \end{pmatrix}$$

```
delete x:
```

Parameters

x

An arithmetical expression, a polynomial of type `DOM_POLY`, or a set

Return Values

Arithmetical expression, a polynomial, or a set.

Overloaded By

x

See Also

MuPAD Functions

`* | + | / | ^ | _invert | _subtract | poly`

*, _mult

Multiply expressions

Syntax

```
x * y * ...
```

```
_mult(x, y, …)
```

Description

`x * y * ...` computes the product of `x`, `y` etc.

`x * y * ...` is equivalent to the function call `_mult(x, y, ...)`.

All terms that are numbers of type `Type::Numeric` are automatically combined to a single number.

The terms of a symbolic product may be rearranged internally if no term belongs to a library domain that overloads `_mult`: on terms composed of kernel domains (numbers, identifiers, expressions etc.), multiplication is assumed to be commutative. Cf. “Example 1” on page 1-63.

Via overloading, the user can implement a non-commutative product for special domains.

`_mult` accepts an arbitrary number of arguments. In conjunction with the sequence operator `$`, this function is the recommended tool for computing finite products. Cf. “Example 2” on page 1-64. The function `product` may also serve for computing such products. However, `product` is designed for the computation of symbolic and infinite products. It is slower than `_mult`.

The quotient `x/y` is internally represented as `x * (1/y)` = `_mult(x, _power(y, -1))`. See `_divide` for details.

Many library domains overload `_mult` by an appropriate slot`"_mult"`. Products involving elements of library domains are processed as follows:

A product `x * y * ...` is searched for elements of library domains from left to right. Let `z` be the first term that is not of one of the basic types provided by the

kernel (numbers, expressions, etc.). If the domain `d` = `z::dom` = `domtype(z)` has a slot`"_mult"`, it is called in the form `d::_mult(x, y, ...)`. The result returned by `d::_mult` is the result of `x * y * ...`.

Cf. “Example 6” on page 1-68 and “Example 7” on page 1-69.

`_mult()` returns the number 1.

Polynomials of type `DOM_POLY` are multiplied by `*`, if they have the same indeterminates and the same coefficient ring. Use `multcoeffs` to multiply polynomials with scalar factors.

For finite sets `X`, `Y`, the product `X * Y` is the set $\{x\,y \mid x \in X, y \in Y\}$.

Equalities, inequalities, and comparisons can be multiplied with one another or with arithmetical expressions. The results of such combinations are demonstrated in “Example 5” on page 1-66.

Examples

Example 1

Numerical terms are simplified automatically:

```
3 * x * y * (1/18) * sin(4) * 4
```

$$\frac{2\,x\,y\,\sin(4)}{3}$$

The ordering of the terms of a product is not necessarily the same as on input:

```
x * y * 3 * z * a * b * c
```

$$3\,a\,b\,c\,x\,y\,z$$

Internally, this product is a symbolic call of `_mult`:

```
op(%, 0), type(%)
```

```
_mult, "_mult"
```

Note that the screen output does not necessarily reflect the internal order of the terms in a product:

```
op(%2)
```

$$a, b, c, x, y, z, 3$$

In particular, a numerical factor is internally stored as the last operand. On the screen, a numerical factor is displayed in front of the remaining terms:

```
3 * x * y * 4
```

$$12\,x\,y$$

```
op(%)
```

$$x, y, 12$$

Example 2

The functional equivalent `_mult` of the operator `*` is a handy tool for computing finite products. In the following, the terms are generated via the sequence operator `$`:

```
_mult(i $ i = 1..20)
```

$$2432902008176640000$$

E.g., it is easy to multiply all elements in a set:

```
S := {a, b, 1, 2, 27}: _mult(op(S))
```

$$54\,a\,b$$

The following command “zips” two lists by multiplying corresponding elements:

```
L1 := [1, 2, 3]: L2 := [a, b, c]: zip(L1, L2, _mult)
```

$$[a, 2\,b, 3\,c]$$

```
delete S, L1, L2:
```

Example 3

Polynomials of type `DOM_POLY` are multiplied by `*`, if they have the same indeterminates and the same coefficient ring:

```
poly(x^2 + 1, [x]) * poly(x^2 + x - 1, [x])
```

$$\mathrm{poly}(x^4 + x^3 + x - 1, [x])$$

If the indeterminates or the coefficient rings do not match, `_mult` returns an error:

```
poly(x, [x]) * poly(x, [x, y])
```

```
Error: The argument is invalid. [_mult]
```

```
poly(x, [x]) * poly(x, [x], Dom::Integer)
```

```
Error: The argument is invalid. [_mult]
```

Using `*`, you can multiply polynomials by scalar factors:

```
2 * y * poly(x, [x])
```

$$\mathrm{poly}((2\,y)\,x, [x])$$

Use `multcoeffs` instead:

```
multcoeffs(poly(x^2 - 2, [x]), 2*y)
```

$$\mathrm{poly}((2\,y)\,x^2 - 4\,y, [x])$$

Example 4

For finite sets X, Y, the product X * Y is the set $\{x\,y \mid x \in X, y \in Y\}$:

```
{a, b, c} * {1, 2}
```

$$\{a, b, c, 2\,a, 2\,b, 2\,c\}$$

Note that complex numbers of type DOM_INT, DOM_RAT, DOM_COMPLEX, DOM_FLOAT, and identifiers are implicitly converted to one-element sets:

```
2 * {a, b, c}
```

$$\{2\,a, 2\,b, 2\,c\}$$

```
a * {b, c}, PI * {3, 4}
```

$$\{a\,b, a\,c\}, \{3\,\pi, 4\,\pi\}$$

Example 5

Multiplying by a constant expression is performed on both sides of an equation:

```
(a = b) * c
```

$$a\,c = b\,c$$

For inequalities, this step is only performed if the constant is known to be non-zero:

```
assume(d <> 0):
(a <> b) * c, (a <> b) * d;
delete d:
```

$$(a \neq b)\,c, a\,d \neq b\,d$$

The multiplication of a comparison with a constant is only defined for real numbers. Even for these, the result depends on the sign of the constant, since multiplication with a negative constant changes the direction of the comparison:

```
(a < b) * 2, (a < b) * (-3)
```

$$2\,a < 2\,b,\ -3\,b < -3\,a$$

```
(a < b) * I
```

```
Error: Inequalities must not be multiplied by complex numbers. [_less::_mult]
```

```
(a < b) * c, (a <= b) * c
```

$$\begin{cases} a\,c < b\,c & \text{if } 0 < c \\ b\,c < a\,c & \text{if } c < 0 \end{cases}, \begin{cases} a\,c \le b\,c & \text{if } 0 \le c \\ b\,c \le a\,c & \text{if } c \le 0 \end{cases}$$

Multiplication of two equalities is performed by multiplying the left hand sides and the right hand sides separately:

```
(a = b) * (c = d)
```

$$a\,c = b\,d$$

Inequalities cannot be multiplied with one another or with comparisons; multiplication with equalities is, however, defined, if at least one operand of the equation is known to be non-zero:

```
assume(d <> 0):
(a <> b) * (c = d);
delete d:
```

$$a\,c \neq b\,d$$

In other cases, the product is not expanded:

```
delete c, d:
(a <> b) * (c = d)
```

$$(a \neq b)\,(c = d)$$

Multiplication of comparisons with equalities and comparisons is performed similar to the cases above:

```
assume(c > 0):
(a < b) * (c = d);
delete c:
```

$$a\,c < b\,d$$

```
(a <= b) * (c <= d)
```

$$\begin{cases} a\,c \le b\,d & \text{if } 0 \le c \wedge 0 \le d \\ b\,d \le a\,c & \text{if } c \le 0 \wedge d \le 0 \end{cases}$$

Example 6

Various library domains such as matrix domains overload `_mult`. The multiplication is not commutative:

```
x := Dom::Matrix(Dom::Integer)([[1, 2], [3, 4]]):
y := Dom::Matrix(Dom::Rational)([[10, 11], [12, 13]]):
x * y, y * x
```

$$\begin{pmatrix} 34 & 37 \\ 78 & 85 \end{pmatrix}, \begin{pmatrix} 43 & 64 \\ 51 & 76 \end{pmatrix}$$

If the terms in `x * y` are of different type, the first term `x` tries to convert `y` to the data type of `x`. If successful, the product is of the same type as `x`. In the previous example, `x` and `y` have different types (both are matrices, but the component domains differ). Hence `x * y` and `y * x` have different types that is inherited from the first term:

```
domtype(x * y), domtype(y * x)
```

$$\text{Dom::Matrix(Dom::Integer)},\ \text{Dom::Matrix(Dom::Rational)}$$

If `x` does not succeed to convert `y`, then `y` tries to convert `x`. In the following call, the component `27/2` cannot be converted to an integer. Consequently, in `x * y`, the term `y` converts `x` and produces a result that coincides with the domain type of `y`:

```
y := Dom::Matrix(Dom::Rational)([[10, 11], [12, 27/2]]):
x * y, y * x
```

$$\begin{pmatrix} 34 & 38 \\ 78 & 87 \end{pmatrix}, \begin{pmatrix} 43 & 64 \\ \frac{105}{2} & 78 \end{pmatrix}$$

```
domtype(x * y), domtype(y * x)
```

$$\text{Dom::Matrix(Dom::Rational)}, \text{Dom::Matrix(Dom::Rational)}$$

```
delete x, y:
```

Example 7

This example demonstrates how to implement a slot `"_mult"` for a domain. The following domain `myString` is to represent character strings. Via overloading of `_mult`, integer multiples of such strings should produce the concatenation of an appropriate number of copies of the string.

The `"new"` method uses `expr2text` to convert any MuPAD object to a string. This string is the internal representation of elements of `myString`. The `"print"` method turns this string into the screen output:

```
myString := newDomain("myString"):
myString::new := proc(x)
begin
  if args(0) = 0 then x := "": end_if;
  case domtype(x)
    of myString do return(x);
    of DOM_STRING do return(new(dom, x));
    otherwise return(new(dom, expr2text(x)));
  end_case
end_proc:
myString::print := x -> extop(x, 1):
```

Without a `"_mult"` method, the system function `_mult` handles elements of this domain like any symbolic object:

```
y := myString(y):
z := myString(z):
4 * x * y * z * 3/2
```

$$6\,x\,y\,z$$

Now, we implement the "_mult" method. It uses split to pick out all integer terms in its argument list and multiplies them. The result is an integer n. If there is exactly one other term left (this must be a string of type myString), it is copied n times. The concatenation of the copies is returned:

```
myString::_mult:= proc()
local Arguments, intfactors, others, dummy, n;
begin
  print(Unquoted, "Info: myString::_mult called with the arguments:",
           args());
  Arguments := [args()];
  // split the argument list into integers and other factors:
  [intfactors, others, dummy] :=
      split(Arguments, testtype, DOM_INT);
  // multiply all integer factors:
  n := _mult(op(intfactors));
  if nops(others) <> 1 then
     return(FAIL)
  end_if;
  myString::new(_concat(extop(others[1], 1) $ n))
end_proc:
```

Now, integer multiples of myString objects can be constructed via the * operator:

```
2 * myString("string") * 3
```

```
Info: myString::_mult called with the arguments:, 2, string, 3
```

stringstringstringstringstringstring

Only products of integers and myString objects are allowed:

```
3/2 * myString("a ") * myString("string")
```

```
                                                    3
Info: myString::_mult called with the arguments:, -, a , string
                                                    2
```

FAIL

```
delete myString, y, z:
```

Parameters

`x, y, …`

arithmetical expressions, polynomials of type `DOM_POLY`, sets, equations, inequalities, or comparisons

Return Values

Arithmetical expression, a polynomial, a set, an equation, an inequality, or a comparison.

Overloaded By

x, y

See Also

MuPAD Functions

`+` | `-` | `/` | `^` | `_invert` | `_subtract` | `poly` | `product`

/, _divide

Divide expressions

Syntax

```
x / y
```

```
_divide(x, y)
```

Description

`x/y` computes the quotient of `x` and `y`.

`x/y` is equivalent to the function call `_divide(x, y)`.

For numbers of type `Type::Numeric`, the quotient is returned as a number.

If neither `x` nor `y` are elements of library domains with `"_divide"` methods, `x/y` is internally represented as `x * y^(-1)` = `_mult(x, _power(y, -1))`.

If `x` or `y` is an element of a domain with a slot`"_divide"`, then this method is used to compute `x/y`. Many library domains overload the `/` operator by an appropriate `"_divide"` slot. Quotients are processed as follows:

`x/y` is searched for elements of library domains from left to right. Let `z` (either `x` or `y`) be the first term that is not of one of the basic types provided by the kernel (numbers, expressions, etc.). If the domain `d` = `z::dom` = `domtype(z)` has a slot`"_divide"`, it is called in the form `d::_divide(x, y)`. The result returned by `d::_divide` is the result of `x/y`.

Cf. examples “Example 4” on page 1-74 and “Example 5” on page 1-75.

Polynomials of type `DOM_POLY` can be divided by `/`, if they have the same indeterminates and the same coefficient ring, and if exact division is possible. The function `divide` can be used to compute the quotient of polynomials with a remainder term.

For finite sets `X`, `Y`, the quotient `X/Y` is the set $\left\{ \frac{x}{y} \;\middle|\; x \in X, y \in Y \right\}$.

Examples

Example 1

The quotient of numbers is simplified to a number:

```
1234/234, 7.5/7, 6*I/2
```

$$\frac{617}{117},\ 1.071428571,\ 3\,i$$

Internally, a symbolic quotient `x/y` is represented as the product `x * y^(-1)`:

```
type(x/y), op(x/y, 0), op(x/y, 1), op(x/y, 2)
```

$$\text{"\_mult"},\ \_\text{mult},\ x,\ \frac{1}{y}$$

```
op(op(x/y, 2), 0), op(op(x/y, 2), 1), op(op(x/y, 2), 2)
```

$$\_\text{power},\ y,\ -1$$

Example 2

For finite sets `X`, `Y`, the quotient `X/Y` is the set $\left\{\frac{x}{y}\ \middle|\ x \in X,\ y \in Y\right\}$:

```
{a, b, c} / {2, 3}
```

$$\left\{\frac{a}{2},\ \frac{a}{3},\ \frac{b}{2},\ \frac{b}{3},\ \frac{c}{2},\ \frac{c}{3}\right\}$$

Example 3

Polynomials of type `DOM_POLY` can be divided by `/` if they have the same indeterminates, the same coefficient ring, and if exact division is possible:

```
poly(x^2 - 1, [x]) / poly(x - 1, [x])
```

$$\mathrm{poly}(x+1, [x])$$

```
poly(x^2 - 1, [x]) / poly(x - 2, [x])
```

$$\mathrm{FAIL}$$

The function `divide` provides division with a remainder:

```
divide(poly(x^2 - 1, [x]), poly(x - 2, [x]))
```

$$\mathrm{poly}(x+2, [x]), \mathrm{poly}(3, [x])$$

The polynomials must have the same indeterminates and the same coefficient ring:

```
poly(x^2 - 1, [x, y]) / poly(x - 1, [x])
```

```
Error: The argument is invalid. [divide]
```

Example 4

Various library domains such as matrix domains overload `_divide`. The matrix domain defines `x/y` as `x * (1/y)`, where `1/y` is the inverse of `y`:

```
x := Dom::Matrix(Dom::Integer)([[1, 2], [3, 4]]):
y := Dom::Matrix(Dom::Rational)([[10, 11], [12, 13]]):
x/y
```

$$\begin{pmatrix} \frac{11}{2} & -\frac{9}{2} \\ \frac{9}{2} & -\frac{7}{2} \end{pmatrix}$$

The inverse of `x` has rational entries. Therefore, `1/x` returns `FAIL`, because the component ring of `x` is `Dom::Integer`. Consequently, also `y/x` returns `FAIL`:

```
y/x
```

$$\mathrm{FAIL}$$

```
delete x, y:
```

Example 5

This example demonstrates the behavior of `_divide` on user-defined domains. In the first case below, the user-defined domain does not have a `"_divide"` slot. Thus `x/y` is transformed to `x * (1/y)`:

```
Do := newDomain("Do"): x := new(Do, 1): y := new(Do, 2):
x/y; op(x/y, 0..2)
```

$$\frac{\text{new(Do, 1)}}{\text{new(Do, 2)}}$$

$$\text{\_mult, new(Do, 1), } \frac{1}{\text{new(Do, 2)}}$$

After the slot `"_divide"` is defined in the domain `Do`, this method is used to divide elements:

```
Do::_divide := proc() begin "The Result" end: x/y
```

"The Result"

```
delete Do, x, y:
```

Parameters

`x, y, …`

arithmetical expressions, polynomials of type `DOM_POLY`, or sets

Return Values

Arithmetical expression, a polynomial, or a set.

Overloaded By

x, y

See Also

MuPAD Functions

`*` | `+` | `-` | `^` | `_invert` | `_subtract` | `div` | `divide` | `pdivide` | `poly`

^, _power

Raise an expression to a power

Syntax

```
x ^ y
```

```
_power(x, y)
```

Description

`x^y` computes the `y`-th power of `x`.

`x^y` is equivalent to the function call `_power(x, y)`.

The power operator `^` is left associative: `x^y^z` is parsed as `(x^y)^z`. Cf. “Example 2” on page 1-78.

If `x` is a polynomial of type `DOM_POLY`, then `y` must be a nonnegative integer smaller than 2^{31}.

`_power` is overloaded for matrix domains (`matrix`). In particular, `x^(-1)` returns the inverse of the matrix `x`.

Use `powermod` to compute modular powers. Cf. “Example 3” on page 1-79.

Mathematically, the call `sqrt(x)` is equivalent to `x^(1/2)`. Note, however, that `sqrt` tries to simplify the result. Cf. “Example 4” on page 1-79.

If `x` or `y` is an element of a domain with a slot`"_power"`, then this method is used to compute `x^y`. Many library domains overload the `^` operator by an appropriate `"_power"` slot. Powers are processed as follows:

`x^y` is searched for elements of library domains from left to right. Let `z` (either `x` or `y`) be the first term that is not of one of the basic types provided by the kernel (numbers, expressions, etc.). If the domain `d = z::dom = domtype(z)` has a slot`"_power"`, it is called in the form `d::_power(x, y)`. The result returned by `d::_power` is the result of `x^y`.

See “Example 6” on page 1-79 and “Example 7” on page 1-80.

For finite sets X, Y, the power X^Y is the set $\{x^y \mid x \in X, y \in Y\}$.

Examples

Example 1

Some powers are computed:

```
2^10, I^(-5), 0.3^(1/3), x^(1/2) + y^(-1/2), (x^(-10) + 1)^2
```

$$1024,\ -i,\ 0.6694329501,\ \sqrt{x} + \frac{1}{\sqrt{y}},\ \left(\frac{1}{x^{10}} + 1\right)^2$$

Use `expand` to “expand” powers of sums:

```
(x + y)^2 = expand((x + y)^2)
```

$$(x+y)^2 = x^2 + 2\,x\,y + y^2$$

Note that identities such as `(x*y)^z = x^z * y^z` only hold in certain areas of the complex plane:

```
((-1)*(-1))^(1/2) <> (-1)^(1/2) * (-1)^(1/2)
```

$$1 \neq -1$$

Consequently, the following `expand` command does not expand its argument:

```
expand((x*y)^(1/2))
```

$$\sqrt{x\,y}$$

Example 2

The power operator ^ is left associative:

```
2^3^4 = (2^3)^4, x^y^z
```

$$4096 = 4096,\ (x^y)^z$$

Example 3

Modular powers can be computed directly using `^` and `mod`. However, `powermod` is more efficient:

```
123^12345 mod 17 = powermod(123, 12345, 17)
```

$$4 = 4$$

Example 4

The function `sqrt` produces simpler results than `_power`:

```
sqrt(4*x*y), (4*x*y)^(1/2)
```

$$2\sqrt{x\,y},\ \sqrt{4\,x\,y}$$

Example 5

For finite sets, `X^Y` is the set $\{x^y \mid x \in X, y \in Y\}$:

```
{a, b, c}^2, {a, b, c}^{q, r, s}
```

$$\{a^2, b^2, c^2\},\ \{a^q, a^r, b^q, a^s, b^r, c^q, b^s, c^r, c^s\}$$

Example 6

Various library domains such as matrix domains or residue class domains overload `_power`:

```
x := Dom::Matrix(Dom::IntegerMod(7))([[2, 3], [3, 4]]):
x^2, x^(-1), x^3 * x^(-3)
```

$$\begin{pmatrix} 6 \bmod 7 & 4 \bmod 7 \\ 4 \bmod 7 & 4 \bmod 7 \end{pmatrix}, \begin{pmatrix} 3 \bmod 7 & 3 \bmod 7 \\ 3 \bmod 7 & 5 \bmod 7 \end{pmatrix}, \begin{pmatrix} 1 \bmod 7 & 0 \bmod 7 \\ 0 \bmod 7 & 1 \bmod 7 \end{pmatrix}$$

```
delete x:
```

Example 7

This example demonstrates the behavior of `_power` on user-defined domains. Without a `"power"` slot, powers of domain elements are handled like any other symbolic powers:

```
myDomain := newDomain("myDomain"): x := new(myDomain, 1): x^2
```

$$\text{new(myDomain, 1)}^2$$

```
type(x^2), op(x^2, 0), op(x^2, 1), op(x^2, 2)
```

"_power", _power, new(myDomain, 1), 2

After the `"_power"` slot is defined, this method is used to compute powers of `myDomain` objects:

```
myDomain::_power := proc() begin "The result" end: x^2
```

"The result"

```
delete myDomain, x:
```

Parameters

`x, y`

arithmetical expressions, polynomials of type `DOM_POLY`, floating-point intervals, or sets

Return Values

Arithmetical expression, a polynomial, a floating-point interval, or a set.

Overloaded By

x, y

See Also

MuPAD Functions

`*` | `+` | `-` | `/` | `_invert` | `_subtract` | `numlib::ispower` | `powermod` | `surd`

@, _fconcat

Compose functions

Syntax

```
f @ g @ ...
```

```
_fconcat(f, g, …)
```

Description

`f@g` represents the composition $x \to f(g(x))$ of the functions `f` and `g`.

In MuPAD, functions are usually represented by procedures of type `DOM_PROC`, functionenvironments, or functional expressions such as `f@g@exp + id^2`. In fact, practically any MuPAD object may serve as a function.

`f @ g` is equivalent to the function call `_fconcat(f, g)`.

`_fconcat()` returns the identity map `id`; `_fconcat(f)` returns `f`.

Examples

Example 1

The following function `h` is the composition of the system functions `abs` and `sin`:

```
h := abs@sin
```

$$\mathrm{abs} \circ \sin$$

```
h(x), h(y + 2), h(0.5)
```

$$|\sin(x)|,\ |\sin(y+2)|,\ 0.4794255386$$

The following functional expressions represent polynomials:

```
f := id^3 + 3*id - 1: f(x), (f@f)(x)
```

$$x^3 + 3\,x - 1,\ 9\,x + (x^3 + 3\,x - 1)^3 + 3\,x^3 - 4$$

The random generator `random` produces nonnegative integers with 12 digits. The following composition of `float` and `random` produces random floating-point numbers between 0.0 and 1.0:

```
rand := float@random/10^12: rand() $ k = 1..12
```

$$0.4274196691,\ 0.3211106933,\ 0.3436330737,\ 0.4742561436,\ 0.558458719,\ 0.7467538305,$$
$$0.03206222209,\ 0.7229741218,\ 0.6043056139,\ 0.7455800374,\ 0.2598119527,\ 0.3100754872$$

In conjunction with the function `map`, the composition operator `@` is a handy tool to apply composed functions to the operands of a data structure:

```
map([1, 2, 3, 4], (PI + id^2)@sin),
map({1, 2, 3, 4}, cos@float)
```

$$[\pi + \sin(1)^2,\ \pi + \sin(2)^2,\ \pi + \sin(3)^2,\ \pi + \sin(4)^2],$$

$$\{-0.9899924966,\ -0.6536436209,\ -0.4161468365,\ 0.5403023059\}$$

```
delete h, f, rand:
```

Example 2

Some simplifications of functional expressions are possible via `simplify`:

```
exp@ln + cos@arccos = simplify(cos@arccos + exp@ln)
```

$$\exp \circ \ln + \cos \circ \arccos = 2\ \mathrm{id}$$

Parameters

`f, g, …`

functions

Return Values

Expression of type `"_fconcat"`.

Overloaded By

f, g

See Also

MuPAD Functions
`@@`

@@, _fnest

Iterate a function

Syntax

```
f @@ n

_fnest(f, n)
```

Description

`f@@n` represents the `n`-fold iterate `x -> f(f(...(f(x))...))` of the function `f`.

The statement `f@@n` is equivalent to the call `_fnest(f, n)`.

For positive `n`, `f@@n` is also equivalent to `_fconcat(f $ n)`.

`f@@0` returns the identity map `id`.

If `f` is a function environment with the slot `"inverse"` set, `n` can also be negative. Cf. “Example 2” on page 1-86.

Iteration is only reasonable for functions that accept their own return values as input. Note that `fp::fixargs` is a handy tool for converting functions with parameters to univariate functions which may be suitable for iteration. Cf. “Example 3” on page 1-86.

Examples

Example 1

For a nonnegative integer `n`, `f@@n` is equivalent to an `_fconcat` call:

```
f@@4, (f@@4)(x)
```

$$f \circ f \circ f \circ f,\ f(f(f(f(x))))$$

`@@` simplifies the composition of symbolic iterates:

```
(f@@n)@@m
```

$$f@@(m\ n)$$

The iterate may be called like any other MuPAD function. If `f` evaluates to a procedure and `n` to an integer, a corresponding value is computed:

```
f := x -> x^2:  (f@@n)(x) $ n = 0..10
```

$$x,\ x^2,\ x^4,\ x^8,\ x^{16},\ x^{32},\ x^{64},\ x^{128},\ x^{256},\ x^{512},\ x^{1024}$$

```
delete f:
```

Example 2

For functions with a known inverse function, `n` may be negative. The function `f` must have been declared as a function environment with the `"inverse"` slot. Examples of such functions include the trigonometric functions which are implemented as function environments in MuPAD:

```
sin::"inverse", sin@@-3, (sin@@(-3))(x)
```

$$\text{"arcsin"},\ \arcsin \circ \arcsin \circ \arcsin,\ \arcsin(\arcsin(\arcsin(x)))$$

Example 3

`@@` can only be used for functions that accept their own output domain as an input, i.e., $f\colon M \to M$ for some set M. If you want to use `@@` with a function which needs additional parameters, `fp::fixargs` is a handy tool to generate a corresponding univariate function. In the following call, the function `f: x -> g(x, p)` is iterated:

```
g := (x, y) -> x^2 + y: f := fp::fixargs(g, 1, p): (f@@4)(x)
```

$$p + (p + (p + (x^2 + p)^2)^2)^2$$

```
delete g, f:
```

Parameters

f

A function

n

An integer or a symbolic name

Return Values

Function

See Also

MuPAD Functions
`@` | `fp::fixargs` | `fp::fold` | `fp::nest` | `fp::nestvals`

$, _seqgen, _seqin, _seqstep

Create an expression sequence

Syntax

```
$ a .. b
_seqgen(a .. b)
$ c .. d step e
_seqstep(c .. d, e)
f $ n
_seqgen(f, n)
f $ c step e
_seqstep(f, c, e)
f $ i = a .. b
_seqgen(f, i, a .. b)
f $ i = c .. d step e
_seqstep(f, i, c .. d, e)
f $ i in object
_seqin(f, i, object)
```

Description

`$ a..b` creates the sequence of integers from `a` through `b`.

`$c..d step e` creates the sequence of numbers from `c` through `d` with increment `e`.

`f $ n` creates the sequence `f,..., f` consisting of `n` copies of `f`.

`f $ c step e` creates the sequence `f,..., f` consisting of `trunc(c/e)` copies of `f`.

`f(i) $ i = a..b` creates the sequence `f(a), f(a+1),..., f(b)`.

`f(i) $ i = c..d step e` creates the sequence `f(c), f(c+e),..., f(c+j*e)`, with `j` such that `c+j*e <= d` and `c+(j+1)*e > d`.

`f(i) $ i in object` creates the sequence `f(i1), f(i2), ...`, where `i1`, `i2` etc. are the operands of the `object`.

The `$` operator is a most useful tool. It serves for generating sequences of objects. Sequences are used to define sets or lists, and may be passed as arguments to system functions. Cf. “Example 1” on page 1-90.

`$ a..b` and the equivalent function call `_seqgen(a..b)` produce the sequence of integers `a, a + 1, ... , b`. The void object of type `DOM_NULL` is produced if `a > b`.

`$ c..d step e` and the equivalent function call `_seqstep(c..d, e)` produce the sequence of numbers `c, c + e, ... , c + j*e`, with `j` such that `c + j*e <= d` and `c + (j + 1)*e > d`. The void object of type `DOM_NULL` is produced if `c > d`.

`f $ n` and the equivalent function call `_seqgen(f, n)` produce a sequence of `n` copies of the object `f`. Note that `f` is evaluated only once, before the sequence is created. The empty sequence of type `DOM_NULL` is produced if `n` is not positive.

`f $ c step e` and the equivalent function call `_seqstep(f, c, e)` produce a sequence of `trunc(c/e)` copies of the object `f`. Note that `f` is evaluated only once, before the sequence is created. The empty sequence of type `DOM_NULL` is produced if `trunc(c/e)` is not positive.

`f $ i = a..b` and the equivalent function call `_seqgen(f, i, a..b)` successively set `i := a` through `i := b` and evaluate `f` with these values. After this (or in case of an error, earlier), the previous value of `i` is restored.

Note that `f` is not evaluated before the first assignment. The void object of type `DOM_NULL` is produced if `a > b`.

`f $ i = c..d step e` and the equivalent function call `_seqstep(f, i, c..d, e)` successively set `i := c`, `i := c + e`, `...` until the value of `i` exceeds `d` and evaluate `f` with these values. After this (or in case of an error, earlier), the previous value of `i` is restored.

Note that `f` is not evaluated before the first assignment. The void object of type `DOM_NULL` is produced if `c > d`.

`f $ i in object` and the equivalent function call `_seqin(f, i, object)` successively assign the operands of the `object` to `i`: they set `i := op(object, 1)` through `i := op(object, n)` and evaluate `f` with these values, returning the result. (`n = nops(object)` is the number of operands.)

Note that `f` is not evaluated before the assignments. The empty sequence of type `DOM_NULL` is produced if the `object` has no operands.

The "loop variable" `i` in `f $ i = a..b` and `f $ i in object` may have a value. This value is restored after the `$` statement returns.

Examples

Example 1

The following sequence can be passed as arguments to the function `_plus`, which adds up its arguments:

```
i^2 $ i = 1..5
```

$$1, 4, 9, 16, 25$$

```
_plus(i^2 $ i = 1..5)
```

$$55$$

The 5-th derivative of the expression `exp(x^2)` is:

```
diff(exp(x^2), x $ 5)
```

$$120\,x\,e^{x^2} + 160\,x^3\,e^{x^2} + 32\,x^5\,e^{x^2}$$

We compute the first derivatives of `sin(x)`:

```
diff(sin(x), x $ i) $ i = 0..5
```

$$\sin(x),\ \cos(x),\ -\sin(x),\ -\cos(x),\ \sin(x),\ \cos(x)$$

We use `ithprime` to compute the first 10 prime numbers:

```
ithprime(i) $ i = 1..10
```

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29$$

We select all primes from the set of integers between 1990 and 2010:

```
select({$ 1990..2010}, isprime)
```

$$\{1993, 1997, 1999, 2003\}$$

The 3×3matrix with entries $A_{ij} = i\,j$ is generated:

```
n := 3: matrix([[i*j $ j = 1..n] $ i = 1..n])
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{pmatrix}$$

```
delete n:
```

Example 2

In `f $ n`, the object `f` is evaluated only once. The result is copied `n` times. Consequently, the following call produces copies of one single random number:

```
random() $ 3
```

$$427419669081, 427419669081, 427419669081$$

The following call evaluates `random` for each value of of `i`:

```
random() $ i = 1..3
```

$$321110693270, 343633073697, 474256143563$$

Example 3

In the following call, `i` runs through the list:

```
i^2 $ i in [3, 2, 1]
```

```
9, 4, 1
```

Note that the screen output of sets does not necessarily coincide with the internal ordering:

```
set := {i^2 $ i = 1..19}:
set;
[op(set)]
```

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361}
```

```
[1, 4, 361, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324]
```

The $ operator respects the internal ordering:

```
i^2 $ i in set
```

```
1, 16, 130321, 81, 256, 625, 1296, 2401, 4096, 6561, 10000, 14641, 20736, 28561, 38416, 50625,
  65536, 83521, 104976
```

```
delete set:
```

Example 4

Arbitrary objects `f` are allowed in `f $ i = a..b`. In the following call, `f` is an assignment (it has to be enclosed in brackets). The sequence computes a table `f[i] = i!`:

```
f[0] := 1: (f[i] := i*f[i - 1]) $ i = 1..4: f
```

0	1
1	1
2	2
3	6
4	24

```
delete f:
```

Example 5

Apart from the usual sequence generator with the step size 1, `_seqstep` allows arbitrary integer, rational, or real numbers as step sizes:

```
1 $ 2 step 0.5
```

$$1, 1, 1, 1$$

```
$ 1..2 step .2
```

$$1, 1.2, 1.4, 1.6, 1.8, 2.0$$

```
f(i) $ i = 1..2 step 1/2
```

$$f(1), f\left(\frac{3}{2}\right), f(2)$$

Like in a `for`-loop, the step size can be negative:

```
f(i) $ i = 5..1 step -2
```

$$f(5), f(3), f(1)$$

In contrast to `_seqgen` the range bounds in `_seqstep` can be rational or floating-point numbers:

```
1 $ 5/2 step 0.5
```

$$1, 1, 1, 1, 1$$

```
$ 1.1..2.1 step .2
```

$$1.1, 1.3, 1.5, 1.7, 1.9, 2.1$$

```
f(i) $ i = 1/2..5/2 step 1/2
```

$$f\left(\frac{1}{2}\right), f(1), f\left(\frac{3}{2}\right), f(2), f\left(\frac{5}{2}\right)$$

Example 6

the $-expression returns symbolically, if the given range is symbolic:

```
x $ n, $ a..b, f(i) $ i = a..b
```

$$x\ \$\ n,\ \$\ a..b,\ f(i)\ \$\ i = a..b$$

Parameters

`f, object`

Arbitrary MuPAD objects

`n, a, b`

integers

`c, d, e`

integer, rational, or floating-point numbers

`i`

An identifier or a local variable (`DOM_VAR`) of a procedure

Return Values

Expression sequence of type "_exprseq" or the void object of type `DOM_NULL`.

Overloaded By

a..b, c..d, e, f, i, n, object

See Also

MuPAD Functions
`_exprseq` | `null`

,, _exprseq

Expression sequences

Syntax

```
object1, object2, ...

_exprseq(object1, object2, …)
```

Description

The function call `_exprseq(object1, object2, ...)` is the internal representation of the expression sequence `object1, object2, ...`.

In MuPAD, “sequences” are ordered collections of objects separated by commas. You may think of the comma as an operator that concatenates sequences. Internally, sequences are represented as function calls `_exprseq(object1, object2, ...)`. On the screen, sequences are printed as `object1, object2, ...`.

`_exprseq()` and the equivalent call `null()` yield the void object of type `DOM_NULL`.

When evaluating an expression sequence, all void objects of type `DOM_NULL` are removed from it, automatically.

The `$` operator is a useful tool for generating sequences.

When a MuPAD function or procedure is called with more than one argument, the parameters are passed as an expression sequence.

Examples

Example 1

A sequence is generated by “concatenating” objects with commas. The resulting object is of type `"_exprseq"`:

```
a, b, sin(x)
```

$$a, b, \sin(x)$$

```
op(%, 0), type(%)
```

$$\text{\_exprseq}, \text{"\_exprseq"}$$

On the screen, `_exprseq` just returns its argument sequence:

```
_exprseq(1, 2, x^2 + 5) = (1, 2, x^2 + 5)
```

$$(1, 2, x^2 + 5) = (1, 2, x^2 + 5)$$

Example 2

The object of domain `DOM_NULL` (the “empty sequence”) is automatically removed from expression sequences:

```
1, 2, null(), 3
```

$$1, 2, 3$$

Expression sequences are flattened. The following sequence does not have 2 operands, where the second operand is a sequence. Instead, it is flattened to a sequence with 3 operands:

```
x := 1: y := 2, 3: x, y
```

$$1, 2, 3$$

```
delete x, y:
```

Example 3

Sequences are used to build sets and lists. Sequences can also be passed to functions that accept several arguments:

```
s := 1, 2, 3: {s}, [s], f(s)
```

$$\{1, 2, 3\}, [1, 2, 3], f(1, 2, 3)$$

```
delete s:
```

Parameters

$\mathbf{object}_1$, $\mathbf{object}_2$, …

Arbitrary MuPAD objects

Return Values

Expression of type `"_exprseq"` or the void object of type `DOM_NULL`.

See Also

MuPAD Functions

`_stmtseq` | `null`

%if

Conditional creation of code by the parser

Syntax

```
%if condition1
then casetrue1
   elif condition2 then casetrue2
   elif condition3 then casetrue3
   ...
   else casefalse
 end_if
```

Description

`%if` controls the creation of code by the parser depending on a condition.

This statement is one of the more esoteric features of MuPAD. It is *not* executed at run time by the MuPAD interpreter. It controls the creation of code for the interpreter by the parser.

`%if` may be used to create different versions of a library which share a common code basis, or to insert debugging code which should not appear in the release version.

The first condition is executed by the parser in a Boolean context and must yield one of the Boolean values `TRUE` or `FALSE`:

- If the condition yields `TRUE`, the statement sequence `casetrue` is the code that is created by the parser for the `%if`-statement. The rest of the statement is ignored by the parser, no code is created for it.
- If the condition yields `FALSE`, then the condition of the next `elif`-part if evaluated and the parser continues as before.
- If all conditions evaluate to `FALSE` and no more `elif`-parts exist, the parser inserts the code of the statement sequence `casefalse` as the code for the `%if`-statement. If no `casefalse` exists, `NIL` is produced.

The whole statement sequence is read by the parser and must be syntactically correct. Also the parts that do not result in code must be syntactically correct.

Note that instead of `end_if`, one may also simply use the keyword `end`.

In case of an empty statement sequence, the parser creates `NIL` as code.

Note: The conditions are parsed in the lexical context where they occur, but are *evaluated by the parser in the context where the parser is executed*. This is the case because the environment where the conditions are exically bound simply does not exist during parsing. Thus, one must ensure that names in the conditions do not conflict with names of local variables or arguments in the surrounding lexical context. The parser does not check this!

No function exists in the interpreter which can execute the `%if`-statement. The reason is that the statement is implemented by the parser, not by the interpreter.

Examples

Example 1

In the following example, we create debugging code in a procedure depending on the value of the global identifier `DEBUG`.

Note that this example is somewhat academic, as the function `prog::trace` is a much more elegant way to trace a procedure during debugging.

```
DEBUG := TRUE:
p := proc(x) begin
     %if DEBUG = TRUE then
         print("entering p")
     end;
     x^2
end_proc:
p(2)
```

```
"entering p"
```

```
4
```

When we look at p, we see that only the print command was inserted by the parser:

```
expose(p)
```

```
proc(x)
  name p;
begin
  print("entering p");
  x^2
end_proc
```

Now we set DEBUG to FALSE and parse the procedure again to create the release version. No debug output is printed:

```
DEBUG := FALSE:
p := proc(x) begin
     %if DEBUG = TRUE then
         print("entering p")
     end;
     x^2
end_proc:
p(2)
```

```
     4
```

If we look at the procedure we see that NIL was inserted for the %if-statement:

```
expose(p)
```

```
proc(x)
  name p;
begin
  NIL;
  x^2
end_proc
```

Parameters

condition

A Boolean expression

`casetrue`

A statement sequence

`casefalse`

A statement sequence

Algorithms

This statement may remind C programmers of conditional compilation. In C, this is implemented by a pre-processor which is run before the parser. In MuPAD, such a pre-processor does not exist. The `%if`-statement is part of the parsing process.

See Also

MuPAD Functions
`if`

;, :, _stmtseq

Statement sequences

Syntax

```
object1; object2; ...

object1: object2: ...

_stmtseq(object1, object2, …)
```

Description

The function call `_stmtseq (object1, object2, ...)` is equivalent to the statement sequence `(object1; object2; ...)`.

The function call `_stmtseq (object1, object2, ...)` evaluates the statements `(object1; object2; ...)` from left to right.

`_stmtseq ()` returns the void object of type `DOM_NULL`.

Examples

Example 1

Usually, statements are entered imperatively:

```
x := 2; x := x^2 + 17; sin(x + 1)
```

```
2
```

```
21
```

```
sin(22)
```

This sequence of statements is turned into a single command (a “statement sequence”) by enclosing it in brackets. Now, only the result of the “statement sequence” is printed. It is the result of the last statement inside the sequence:

```
(x := 2; x := x^2 + 17; sin(x + 1))
```

$$\sin(22)$$

Alternatively, the statement sequence can be entered via `_stmtseq`. For syntactical reasons, the assignments have to be enclosed in brackets when using them as arguments for `_stmtseq`. Only the return value of the statement sequence (the return value of the last statement) is printed:

```
_stmtseq((x := 2), (x := x^2 + 17),  sin(x + 1))
```

$$\sin(22)$$

Statement sequences can be iterated:

```
x := 1: (x := x + 1; x := x^2; print(i, x)) $ i = 1..4
```

1, 4

2, 25

3, 676

4, 458329

```
delete x:
```

Parameters

$\mathtt{object}_1$, $\mathtt{object}_2$, …

Arbitrary MuPAD objects and statements

Return Values

Return value of the last statement in the sequence.

See Also

MuPAD Functions

`_exprseq`

abs

Absolute value of a real or complex number

Syntax

```
abs(z)
```

```
abs(L)
```

Description

`abs(z)` returns the absolute value of the number `z`.

For many constant expressions, `abs` returns the absolute value as an explicit number or expression. Cf. “Example 1” on page 1-107.

A symbolic call of `abs` is returned if the absolute value cannot be determined (e.g., because the argument involves identifiers). The result is subject to certain simplifications. In particular, `abs` extracts constant factors. Properties of identifiers are taken into account. See “Example 2” on page 1-107 and “Example 3” on page 1-108.

The `expand` function rewrites the absolute value of a product to a product of absolute values. E.g., `expand(abs(x*y))` yields `abs(x)*abs(y)`. Cf. “Example 4” on page 1-108.

The symbolic constants `CATALAN`, `E`, `EULER`, and `PI` are processed by `abs`. Cf. “Example 5” on page 1-108.

The absolute value of symbolic function calls can be defined via the slot `"abs"` of function environments. Cf. “Example 7” on page 1-109.

In the same way, the absolute value of domain elements can be defined via overloading. Cf. “Example 8” on page 1-109.

This function is automatically mapped to all entries of container objects such as arrays, lists, matrices, polynomials, sets, and tables.

Environment Interactions

`abs` respects properties of identifiers.

Examples

Example 1

For many constant expressions, the absolute value can be computed explicitly:

```
abs(1.2), abs(-8/3), abs(3 + I), abs(sqrt(-3))
```

$$1.2,\ \frac{8}{3},\ \sqrt{10},\ \sqrt{3}$$

```
abs(sin(42)), abs(PI^2 - 10), abs(exp(3) - tan(157/100))
```

$$-\sin(42),\ 10-\pi^2,\ \tan\left(\frac{157}{100}\right)-e^3$$

```
abs(exp(3 + I) - sqrt(2))
```

$$\sqrt{e^6\sin(1)^2+(\cos(1)\,e^3-\sqrt{2})^2}$$

Example 2

Symbolic calls are returned if the argument contains identifiers without properties:

```
abs(x), abs(x + 1), abs(sin(x + y))
```

$$|x|,\ |x+1|,\ |\sin(x+y)|$$

The result is subject to some simplifications. In particular, `abs` splits off constant factors in products:

```
abs(PI*x*y), abs((1 + I)*x), abs(sin(4)*(x + sqrt(3)))
```

$$\pi\,|x\,y|,\ \sqrt{2}\,|x|,\ -\sin(4)\,|x+\sqrt{3}|$$

Example 3

`abs` is sensitive to properties of identifiers:

```
assume(x < 0):  abs(3*x), abs(PI - x), abs(I*x)
```

$$-3\,x,\ \pi - x,\ -x$$

```
unassume(x):
```

Example 4

The `expand` function produces products of `abs` calls:

```
abs(x*(y + 1)), expand(abs(x*(y + 1)))
```

$$|x\,(y+1)|,\ |y+1|\,|x|$$

Example 5

The absolute value of the symbolic constants `PI`, `EULER`, etc. are known:

```
abs(PI), abs(EULER + CATALAN^2)
```

$$\pi,\ \mathrm{EULER} + \mathrm{CATALAN}^2$$

Example 6

Expressions containing `abs` can be differentiated:

```
diff(abs(x), x),  diff(abs(x), x, x)
```

$$\mathrm{sign}(x),\ 2\,\delta(x)$$

Example 7

The slot `"abs"` of a function environment `f` defines the absolute value of symbolic calls of `f`:

```
abs(f(x))
```

$$|f(x)|$$

```
f := funcenv(f):
f::abs := x -> f(x)/sign(f(x)):
abs(f(x))
```

$$\frac{f(x)}{\mathrm{sign}(f(x))}$$

```
delete f:
```

Example 8

The slot `"abs"` of a domain `d` defines the absolute value of its elements:

```
d := newDomain("d"):
e1 := new(d, 2):
e2 := new(d, x):
d::abs := x -> abs(extop(x, 1)):
abs(e1), abs(e2)
```

$$2,\ |x|$$

```
delete d, e1, e2:
```

Parameters

z

An arithmetical expression

L

A container object: an array, an hfarray, a list, a matrix, a polynomial, a set, or a table.

Return Values

arithmetical expression or a container object containing such expressions

Overloaded By

z

See Also

MuPAD Functions

`conjugate` | `Im` | `norm` | `Re` | `sign`

airyAi

Airy function of the first kind

Syntax

```
airyAi(z)

airyAi(z, n)
```

Description

`airyAi(z)` represents the Airy function of the first kind. The Airy functions `airyAi(z)` and `airyBi(z)` are linearly independent solutions of the differential equation $\frac{\partial^2}{\partial z^2} y - z\, y = 0$.

The call `airyAi(z)` is equivalent to `airyAi(z, 0)`.

`airyAi(z, n)` represents the n-th derivative of `airyAi(z)` with respect to z.

For $n \geq 2$, derivatives of the Airy functions are automatically expressed in terms of the Airy functions and their first derivative. See “Example 1” on page 1-112.

`airyAi` returns special values for $z = 0$ and $z = \pm\infty$. For all other symbolic values of z, unevaluated function calls are returned. See “Example 2” on page 1-112.

Environment Interactions

When called with floating-point arguments, this function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Second and higher derivatives of Airy functions are rewritten in terms of Airy functions and their first derivatives:

```
airyAi(x), airyAi(x, 1), airyBi(sin(x), 3)
```

$$\text{airyAi}(x, 0),\ \text{airyAi}(x, 1),\ \text{airyBi}(\sin(x), 0) + \sin(x)\ \text{airyBi}(\sin(x), 1)$$

Example 2

For $z = 0$, special values are returned:

```
airyAi(0), airyBi(0, 1), airyAi(0, 27)
```

$$\frac{3^{1/3}}{3\,\Gamma\left(\frac{2}{3}\right)},\ \frac{3^{2/3}\,\Gamma\left(\frac{2}{3}\right)}{2\,\pi},\ \frac{608608000\ 3^{1/3}}{3\,\Gamma\left(\frac{2}{3}\right)}$$

For $n = 0, 1$ and any symbolic $z \neq 0$, $z \neq \pm\infty$, a symbolic call is returned:

```
airyAi(-1), airyBi(x, 1)
```

$$\text{airyAi}(-1, 0),\ \text{airyBi}(x, 1)$$

floating-point values are returned for floating-point arguments:

```
airyBi(0.0), airyAi(-3.24819, 1), airyBi(-3.45 + 2.75*I);
```

$$0.6149266274,\ 0.00001031967672,\ -16.85910551 - 32.61659997\,i$$

Example 3

The functions `diff`, `float`, `limit`, and `series` handle expressions involving the Airy functions

```
diff(airyBi(x^2), x), float(airyAi(PI))
```

$$2\,x\,\mathrm{airyBi}(x^2,\,1),\,0.00508935348$$

```
limit(airyAi(-x), x = infinity),
series(airyBi(x, 1), x = infinity, 4)
```

$$0,\ \frac{x^{1/4}\,\sigma_1}{\sqrt{\pi}} - \frac{7\,\sigma_1}{48\,\sqrt{\pi}\,x^{5/4}} - \frac{455\,\sigma_1}{4608\,\sqrt{\pi}\,x^{11/4}} + O\left(\frac{\sigma_1}{x^{15/4}}\right)$$

where

$$\sigma_1 = \left(e^{x^{3/2}}\right)^{2/3}$$

Parameters

z

Arithmetical expression

n

Arithmetical expression representing a nonnegative integer

Return Values

Arithmetical expression.

Overloaded By

z

See Also

MuPAD Functions
`airyBi` | `besselI` | `besselJ` | `besselK`

airyBi

Airy function of the second kind

Syntax

```
airyBi(z)

airyBi(z, n)
```

Description

`airyBi(z)` represents the Airy function of the second kind. The Airy functions `airyAi(z)` and `airyBi(z)` are linearly independent solutions of the differential equation $\frac{\partial^2}{\partial z^2} y - z\, y = 0$.

The call `airyBi(z)` is equivalent to `airyBi(z, 0)`.

`airyBi(z, n)` represents the n-th derivative of `airyBi(z)` with respect to z.

For $n \geq 2$, derivatives of the Airy functions are automatically expressed in terms of the Airy functions and their first derivative. See “Example 1” on page 1-116.

`airyBi` returns special values for $z = 0$ and $z = \pm\infty$. For all other symbolic values of z, unevaluated function calls are returned. See “Example 2” on page 1-116.

Environment Interactions

When called with floating-point arguments, this functions is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Second and higher derivatives of Airy functions are rewritten in terms of Airy functions and their first derivatives:

```
airyAi(x), airyAi(x, 1), airyBi(sin(x), 3)
```

$$\text{airyAi}(x, 0),\ \text{airyAi}(x, 1),\ \text{airyBi}(\sin(x), 0) + \sin(x)\ \text{airyBi}(\sin(x), 1)$$

Example 2

For $z = 0$, special values are returned:

```
airyAi(0), airyBi(0, 1), airyAi(0, 27)
```

$$\frac{3^{1/3}}{3\,\Gamma\left(\frac{2}{3}\right)},\ \frac{3^{2/3}\,\Gamma\left(\frac{2}{3}\right)}{2\,\pi},\ \frac{608608000\ 3^{1/3}}{3\,\Gamma\left(\frac{2}{3}\right)}$$

For $n = 0, 1$ and any symbolic $z \neq 0$, $z \neq \pm\infty$, a symbolic call is returned:

```
airyAi(-1), airyBi(x, 1)
```

$$\text{airyAi}(-1, 0),\ \text{airyBi}(x, 1)$$

floating-point values are returned for floating-point arguments:

```
airyBi(0.0), airyAi(-3.24819, 1), airyBi(-3.45 + 2.75*I);
```

$$0.6149266274,\ 0.00001031967672,\ -16.85910551 - 32.61659997\,i$$

Example 3

The functions `diff`, `float`, `limit`, and `series` handle expressions involving the Airy functions

```
diff(airyBi(x^2), x), float(airyAi(PI))
```

$$2\,x\,\mathrm{airyBi}(x^2, 1),\ 0.00508935348$$

```
limit(airyAi(-x), x = infinity),
series(airyBi(x, 1), x = infinity, 4)
```

$$0,\ \frac{x^{1/4}\,\sigma_1}{\sqrt{\pi}} - \frac{7\,\sigma_1}{48\,\sqrt{\pi}\,x^{5/4}} - \frac{455\,\sigma_1}{4608\,\sqrt{\pi}\,x^{11/4}} + O\left(\frac{\sigma_1}{x^{15/4}}\right)$$

where

$$\sigma_1 = \left(e^{x^{3/2}}\right)^{2/3}$$

Parameters

z

Arithmetical expression

n

Arithmetical expression representing a nonnegative integer

Return Values

Arithmetical expression.

Overloaded By

z

See Also

MuPAD Functions
`airyAi` | `besselI` | `besselJ` | `besselK`

alias

Defines an abbreviation or a macro

Syntax

```
alias(x1 = object1, x2 = object2, …, <Global>)
```

```
alias(<Global>)
```

Description

`alias(x = object)` defines `x` as an abbreviation for `object`.

`alias(x = object)` defines an abbreviation. It changes the configuration of the parser such that the identifier`x` is replaced by `object` whenever it occurs in the input, and such that `object` is in turn replaced by `x` in the output.

`alias(f(y1, y2, ...) = object)` defines `f` to be a macro. For arbitrary objects `a1, a2, ...`, `f(a1, a2, ...)` is equivalent to `object` with `a1` substituted for `y1`, `a2` substituted for `y2`, etc.

`alias(f(y1, y2, ...) = object)` defines a macro. It changes the configuration of the parser such that a function call of the form `f(a1, a2, ...)`, where `a1,a2,...` is a sequence of arbitrary objects of the same length as `y1,y2,...`, is replaced by `object` with `a1` substituted for `y1`, `a2` substituted for `y2`, etc.

No substitution takes place if the number of parameters `y1,y2,...` differs from the number of arguments `a1,a2,...`. No substitution takes place in the output.

It is valid to define a macro with no arguments via `alias(f()=object)`.

Multiple alias definitions may be given in a single call; abbreviations and macros may be mixed.

`alias()` displays a list of all currently defined aliases and macros.

`alias()` displays all currently defined aliases as a sequence of equations. For an abbreviation defined via `alias(x = object)`, the equation `x = object` is printed. For a macro defined via `alias(f(y1, y2, ...) = object)`, the equation `f(y1,`

`y2, ...) = object` is printed. If no aliases are defined, the message "No alias defined" is printed. See "Example 11" on page 1-127.

`alias` does not evaluate its arguments. Hence it has no effect if the aliased identifier has a value, and `alias` creates an alias for the right hand side of the alias definition and not for its evaluation. Cf. "Example 2" on page 1-122.

`alias` does not flatten its arguments. Thus an expression sequence is a valid right hand side of an alias definition. See "Example 5" on page 1-125.

An alias definition causes a substitution similar to the effect of `subs`, not just a textual replacement. Cf. "Example 3" on page 1-123.

Each identifier may be aliased to only one object. Each object may be abbreviated in only one way; otherwise `alias` aborts with an error.

An alias is in effect from the time when the call to `alias` has been evaluated. It affects exactly those inputs that are *parsed* after that moment. Cf. "Example 9" on page 1-126. In particular, an alias definition inside a procedure does not affect the rest of the procedure.

By default, back-substitution of aliases in the output happens only for abbreviations and not for macros. After a command of the form `alias(x = object)`, both the unevaluated object `object` and its evaluation are replaced by the unevaluated identifier `x` in the output. Cf. "Example 2" on page 1-122.

The user can control the behavior of the back-substitution in the output with the function `Pref::alias`; see the corresponding help page for details.

Substitutions in the output only happen for the results of computations at interactive level. The behavior of the functions `fprint`, `print`, `expr2text`, or `write` is not affected.

Alias substitutions are performed in parallel, both in the input and in the output. Thus it is not possible to define nested aliases. See "Example 10" on page 1-126.

If an identifier is used as an abbreviation, it is not possible to enter this identifier in its literal meaning any longer.

Note: In particular, it is necessary to use `unalias` before another abbreviation or macro for the same identifier can be defined. Cf. "Example 4" on page 1-124.

If a macro `f(y1,y2,...,yn)` with n arguments has been defined, it is not possible to enter a call to `f` with n arguments in its literal meaning any longer. However, entering a call to `f` with a different number of arguments is still possible. Cf. “Example 5” on page 1-125.

Macros with different numbers of arguments can be defined at the same time. See “Example 4” on page 1-124.

An alias definition affects all kinds of input: interactive input on the command line, input via the function `input`, input from a file using `finput`, `fread`, or `read` (for the latter two only if option `Plain` is not set), and input from a string using `text2expr`. Cf. “Example 8” on page 1-126.

An alias definition has no effect on the identifier used as an alias. In particular, that identifier retains its value and its properties. The alias and the aliased object are still distinguished by the evaluator. Cf. “Example 6” on page 1-125.

Assigning a value to one of the identifiers on the left hand side of an alias definition, or deleting its value has no effect on the alias substitution, neither in the input nor in the output. See “Example 7” on page 1-125.

Environment Interactions

`alias` with at least one argument and `unalias` change the parser configuration in the way described in the “Details” section.

Examples

Example 1

We define `d` as a shortcut for `diff`:

```
delete f, g, x, y: alias(d = diff):
d(sin(x), x) = diff(sin(x), x);
d(f(x, y), x) = diff(f(x, y), x)
```

$$\cos(x) = \cos(x)$$

$$d(f(x, y), x) = d(f(x, y), x)$$

We define a macro `Dx(f)` for `diff(f(x), x)`. Note that `hold` does not prevent alias substitution:

```
alias(Dx(f) = diff(f(x), x)):
Dx(sin); Dx(f + g); hold(Dx(f + g))
```

$$\cos(x)$$

$$d(f(x), x) + d(g(x), x)$$

$$d(f(x) + g(x), x)$$

After the call `unalias(d, Dx)`, no alias substitutions happen any longer:

```
unalias(d, Dx):
d(sin(x), x), diff(sin(x), x), d(f(x, y), x), diff(f(x, y), x);
Dx(sin), Dx(f + g)
```

$$d(\sin(x), x),\ \cos(x),\ d(f(x, y), x),\ \frac{\partial}{\partial x} f(x, y)$$

$$\mathrm{Dx}(\sin),\ \mathrm{Dx}(f + g)$$

Example 2

Suppose we want to avoid typing `longhardtotypeident` and therefore define an abbreviation `a` for it:

```
longhardtotypeident := 10;  alias(a = longhardtotypeident):
```

$$10$$

Since `alias` does not evaluate its arguments, `a` is now an abbreviation for `longhardtotypeident` and not for the number `10`:

```
type(a), type(hold(a))
```

DOM_INT, DOM_IDENT

```
a + 1, hold(a) + 1, eval(hold(a) + 1)
```

$11, a + 1, 11$

```
longhardtotypeident := 2:
a + 1, hold(a) + 1, eval(hold(a) + 1)
```

$3, a + 1, 3$

However, by default alias back-substitution in the output happens for both the identifier and its current value:

```
2, 10, longhardtotypeident, hold(longhardtotypeident)
```

$a, 10, a, a$

The command `Pref::alias(FALSE)` switches alias resubstitution off:

```
p := Pref::alias(FALSE):
a, hold(a), 2, longhardtotypeident, hold(longhardtotypeident);
Pref::alias(p): unalias(a):
```

2, longhardtotypeident, 2, 2, longhardtotypeident

Example 3

Aliases are substituted and not just replaced textually. In the following example, `3*succ(u)` is replaced by `3*(u+1)`, and not by `3*u+1`, which a search-and-replace function in a text editor would produce:

```
alias(succ(x) = x + 1): 3*succ(u);
unalias(succ):
```

$$3\,u+3$$

Example 4

We define a to be an abbreviation for b. Then the next alias definition is really an alias definition for b:

```
delete a, b:
alias(a = b): alias(a = 2): type(a), type(b); unalias(b):
```

DOM_IDENT, DOM_INT

Use `unalias` first before defining another alias for the identifier a:

```
unalias(a): alias(a = 2): type(a), type(b); unalias(a):
```

DOM_INT, DOM_IDENT

A macro definition, however, can be added if the newly defined macro has a different number of arguments. `unalias(a)` removes all macros defined for a:

```
alias(a(x)=sin(x^2)): a(y); alias(a(x)=cos(x^2)):
```

$$\sin(y^2)$$

```
Error: The operand is invalid. [_power]
  Evaluating: alias
```

```
alias(a(x, y) = sin(x + y)):
a(u, v);
alias():
unalias(a):
```

$$\sin(u+v)$$

```
a(x)   = sin(x^2)

a(x, y) = sin(x + y)
```

Example 5

A macro definition has no effect when called with the wrong number of arguments, and the sequence of arguments is not flattened:

```
alias(plus(x, y) = x + y):
plus(1), plus(3, 2), plus((3, 2));
unalias(plus):
```

$$\text{plus}(1),\ 5,\ \text{plus}(3,\ 2)$$

Expression sequences may appear on the right hand side of an alias definition, but they have to be enclosed in parenthesis:

```
alias(x = (1, 2)): f := 0, 1, 2, x;
nops(f); unalias(x):
```

$$0,\ 1,\ 2,\ 1,\ 2$$

$$5$$

Example 6

An identifier used as an abbreviation may still exist in its literal meaning inside expressions that were entered before the alias definition:

```
delete x: f := [x, 1]: alias(x = 1): f;
map(f, type); unalias(x):
```

$$[x,\ x]$$

$$[\text{DOM\_IDENT},\ \text{DOM\_INT}]$$

Example 7

It does not matter whether the identifier used as an alias has a value:

```
a := 5: alias(a = 7): 7, 5; print(a); unalias(a):
```

```
a, 5

7

delete a:
```

Example 8

Alias definitions also apply to input from files or strings:

```
alias(a = 3): type(text2expr("a")); unalias(a)

    DOM_INT
```

Example 9

An alias is valid for all input that is *parsed* after executing `alias`. A statement in a command line is not parsed before the previous commands in that command line have been executed. In the following example, the alias is already in effect for the second statement:

```
alias(a = 3): type(a); unalias(a)

    DOM_INT
```

This can be changed by entering additional parentheses:

```
(alias(a = 3): type(a)); unalias(a)

    DOM_IDENT
```

Example 10

We define `b` to be an alias for `c`, which in turn is defined to be an alias for `2`. It is recommended to avoid such chains of alias definitions beacuse of some probably unwanted effects.

```
alias(b=c): alias(c=2):
```

Now each `b` in the input is replaced by `c`, but no additional substitution step is taken to replace this again by `2`:

```
print(b)
```

$$c$$

On the other hand, the number `2` is replaced by `c` in every output and that `c` is then replaced by `b`:

```
2
```

$$b$$

```
unalias(c): unalias(b):
```

Example 11

When called without arguments, `alias` just displays all aliasses that are currently in effect:

```
alias(a = 5, F(x) = sin(x^2)):
alias(); unalias(F, a):
```

```
F(x) = sin(x^2)
```

```
a    = 5
```

Parameters

x_1, x_2, …

identifiers or symbolic expressions of the form `f(`y_1`, `y_2`, …)`, with identifiers `f`, y_1, y_2, `...`

$object_1$, $object_2$, …

Any MuPAD objects

Options

`Global`

Definition of an alias in the global parser context.

When an alias is defined in a library or package source file, it will be deleted automatically after reading the file. With option `Global` the alias is not active in the file being read, but in the interactive level after reading of the file is finished.

Return Values

Both `alias` and `unalias` return the void object of type `DOM_NULL`.

Algorithms

The aliases are stored in the parser configuration table displayed by `_parser_config()`. Note that by default, alias back-substitution happens for the right hand sides of the equations in this table, but not for the indices. Use `print(_parser_config())` to display this table without alias back-substitution.

Aliases are not in effect while a file is read using `read` or `fread` with option `Plain`. Conversely, if an alias is defined in a file which is read with option `Plain`, the alias is only in effect until the file has been read completely.

See Also

MuPAD Functions

`:=` | `finput` | `fprint` | `fread` | `input` | `Pref::alias` | `print` | `proc` | `read` | `subs` | `text2expr` | `unalias` | `write`

unalias

Deletes an alias-definition

Syntax

```
unalias(x1, x2, …, <Global>)
```

```
unalias(<Global>)
```

Description

`unalias(x)` deletes the abbreviation or macro `x`. To delete a macro defined by `alias(f(y1, y2, ...) = object)`, use `unalias(f)`. If no alias for `x` or `f`, respectively, is defined currently, the call is ignored.

`unalias()` deletes all abbreviations and macros.

Multiple alias definitions may be deleted by a single call of `unalias`. The call `unalias()` deletes all currently defined aliases.

`unalias` does not evaluate its arguments.

If an identifier is used as an abbreviation, it is not possible to enter this identifier in its literal meaning any longer.

Note: In particular, it is necessary to use `unalias` before another abbreviation or macro for the same identifier can be defined. Cf. “Example 4” on page 1-132.

Assigning a value to one of the identifiers on the left hand side of an alias definition, or deleting its value has no effect on the alias substitution, neither in the input nor in the output. See “Example 7” on page 1-134.

Environment Interactions

`alias` with at least one argument and `unalias` change the parser configuration in the way described in the “Details” section.

Examples

Example 1

We define `d` as a shortcut for `diff`:

```
delete f, g, x, y: alias(d = diff):
d(sin(x), x) = diff(sin(x), x);
d(f(x, y), x) = diff(f(x, y), x)
```

$$\cos(x) = \cos(x)$$

$$d(f(x, y), x) = d(f(x, y), x)$$

We define a macro `Dx(f)` for `diff(f(x), x)`. Note that `hold` does not prevent alias substitution:

```
alias(Dx(f) = diff(f(x), x)):
Dx(sin); Dx(f + g); hold(Dx(f + g))
```

$$\cos(x)$$

$$d(f(x), x) + d(g(x), x)$$

$$d(f(x) + g(x), x)$$

After the call `unalias(d, Dx)`, no alias substitutions happen any longer:

```
unalias(d, Dx):
d(sin(x), x), diff(sin(x), x), d(f(x, y), x), diff(f(x, y), x);
Dx(sin), Dx(f + g)
```

$$d(\sin(x), x),\ \cos(x),\ d(f(x, y), x),\ \frac{\partial}{\partial x} f(x, y)$$

$$\mathrm{Dx}(\sin),\ \mathrm{Dx}(f + g)$$

Example 2

Suppose we want to avoid typing `longhardtotypeident` and therefore define an abbreviation `a` for it:

```
longhardtotypeident := 10;  alias(a = longhardtotypeident):
```

$$10$$

Since `alias` does not evaluate its arguments, `a` is now an abbreviation for `longhardtotypeident` and not for the number `10`:

```
type(a), type(hold(a))
```

$$\text{DOM\_INT}, \text{DOM\_IDENT}$$

```
a + 1, hold(a) + 1, eval(hold(a) + 1)
```

$$11, a + 1, 11$$

```
longhardtotypeident := 2:
a + 1, hold(a) + 1, eval(hold(a) + 1)
```

$$3, a + 1, 3$$

However, by default alias back-substitution in the output happens for both the identifier and its current value:

```
2, 10, longhardtotypeident, hold(longhardtotypeident)
```

$$a, 10, a, a$$

The command `Pref::alias(FALSE)` switches alias resubstitution off:

```
p := Pref::alias(FALSE):
a, hold(a), 2, longhardtotypeident, hold(longhardtotypeident);
Pref::alias(p): unalias(a):
```

2, longhardtotypeident, 2, 2, longhardtotypeident

Example 3

Aliases are substituted and not just replaced textually. In the following example, `3*succ(u)` is replaced by `3*(u+1)`, and not by `3*u+1`, which a search-and-replace function in a text editor would produce:

```
alias(succ(x) = x + 1): 3*succ(u);
unalias(succ):
```

$$3\,u+3$$

Example 4

We define `a` to be an abbreviation for `b`. Then the next alias definition is really an alias definition for `b`:

```
delete a, b:
alias(a = b): alias(a = 2): type(a), type(b); unalias(b):
```

DOM_IDENT, DOM_INT

Use `unalias` first before defining another alias for the identifier `a`:

```
unalias(a): alias(a = 2): type(a), type(b); unalias(a):
```

DOM_INT, DOM_IDENT

A macro definition, however, can be added if the newly defined macro has a different number of arguments. `unalias(a)` removes all macros defined for `a`:

```
alias(a(x)=sin(x^2)): a(y); alias(a(x)=cos(x^2)):
```

$$\sin(y^2)$$

```
Error: The operand is invalid. [_power]
  Evaluating: alias
```

```
alias(a(x, y) = sin(x + y)):
a(u, v);
alias():
unalias(a):
```

$$\sin(u+v)$$

```
a(x)   = sin(x^2)

a(x, y) = sin(x + y)
```

Example 5

A macro definition has no effect when called with the wrong number of arguments, and the sequence of arguments is not flattened:

```
alias(plus(x, y) = x + y):
plus(1), plus(3, 2), plus((3, 2));
unalias(plus):
```

$$\text{plus}(1),\ 5,\ \text{plus}(3,\ 2)$$

Expression sequences may appear on the right hand side of an alias definition, but they have to be enclosed in parenthesis:

```
alias(x = (1, 2)): f := 0, 1, 2, x;
nops(f); unalias(x):
```

$$0,\ 1,\ 2,\ 1,\ 2$$

$$5$$

Example 6

An identifier used as an abbreviation may still exist in its literal meaning inside expressions that were entered before the alias definition:

```
delete x: f := [x, 1]: alias(x = 1): f;
map(f, type); unalias(x):
```

$[x, x]$

[DOM_IDENT, DOM_INT]

Example 7

It does not matter whether the identifier used as an alias has a value:

```
a := 5: alias(a = 7): 7, 5; print(a); unalias(a):
```

a, 5

7

```
delete a:
```

Example 8

Alias definitions also apply to input from files or strings:

```
alias(a = 3): type(text2expr("a")); unalias(a)
```

DOM_INT

Example 9

An alias is valid for all input that is *parsed* after executing `alias`. A statement in a command line is not parsed before the previous commands in that command line have been executed. In the following example, the alias is already in effect for the second statement:

```
alias(a = 3): type(a); unalias(a)
```

```
DOM_INT
```

This can be changed by entering additional parentheses:

```
(alias(a = 3): type(a)); unalias(a)
```

```
DOM_IDENT
```

Example 10

We define `b` to be an alias for `c`, which in turn is defined to be an alias for `2`. It is recommended to avoid such chains of alias definitions beacuse of some probably unwanted effects.

```
alias(b=c): alias(c=2):
```

Now each `b` in the input is replaced by `c`, but no additional substitution step is taken to replace this again by `2`:

```
print(b)
```

```
c
```

On the other hand, the number `2` is replaced by `c` in every output and that `c` is then replaced by `b`:

```
2
```

```
b
```

```
unalias(c): unalias(b):
```

Example 11

When called without arguments, `alias` just displays all aliasses that are currently in effect:

```
alias(a = 5, F(x) = sin(x^2)):
```

```
alias(); unalias(F, a):

F(x) = sin(x^2)

a    = 5
```

Parameters

x_1, x_2, …

identifiers or symbolic expressions of the form `f(`y_1`, `y_2`, …)`, with identifiers `f, `y_1`, `y_2`, ...`

$object_1$, $object_2$, …

Any MuPAD objects

Options

`Global`

Definition of an alias in the global parser context.

When an alias is defined in a library or package source file, it will be deleted automatically after reading the file. With option `Global` the alias is not active in the file being read, but in the interactive level after reading of the file is finished.

Return Values

Both `alias` and `unalias` return the void object of type `DOM_NULL`.

Algorithms

The aliases are stored in the parser configuration table displayed by `_parser_config()`. Note that by default, alias back-substitution happens for the right hand sides of the equations in this table, but not for the indices. Use `print(_parser_config())` to display this table without alias back-substitution.

Aliases are not in effect while a file is read using `read` or `fread` with option `Plain`. Conversely, if an alias is defined in a file which is read with option `Plain`, the alias is only in effect until the file has been read completely.

See Also

MuPAD Functions

`:=` | `alias` | `finput` | `fprint` | `fread` | `input` | `Pref::alias` | `print` | `proc` | `read` | `subs` | `text2expr` | `write`

anames

Identifiers that have values or properties

Syntax

```
anames(<All>, <User>)
```

```
anames(<Properties>, <User>)
```

```
anames(<Protected>, <User>)
```

```
anames(d, <User>)
```

Description

`anames(All)` returns all identifiers that have values.

`anames(Properties)` returns all identifiers that have properties.

`anames(Protected)` returns all identifiers that are protected.

`anames(d)` returns all identifiers that have values from the given domain `d`.

The result returned by `anames` is a set of *unevaluated* identifiers.

`anames` does not take into account slots of function environments or domains. Moreover, functions of a MuPAD library are considered only if they are exported.

Examples

Example 1

`anames(All, User)` returns all user-defined identifiers:

```
a := b:  b := 2:  c := {2, 3}:
```

```
anames(All, User)
```

$$\{a, b, c\}$$

If the first argument is a domain, only identifiers with *values* from that domain are returned. These may differ from the identifiers whose *evaluation* belongs to the domain:

```
a, b;
anames(DOM_IDENT, User);
anames(DOM_INT, User)
```

$$2, 2$$

$$\{a\}$$

$$\{b\}$$

Example 2

`anames(Properties)` returns all identifiers that have been attached properties via `assume`:

```
assume(x > y): anames(Properties)
```

$$\{x, y\}$$

Example 3

`anames(Protected)` returns all identifiers that are protected via `protect`; since all system functions are protected, we use `anames(Protected, User)`:

```
protect(a): anames(Protected, User)
```

$$\{a\}$$

Parameters

`d`

A domain

Options

`All`

Get all identifiers that have values

`Properties`

Get all identifiers that have properties

`Protected`

Get all identifiers that are protected

`User`

Exclude all system variables

If the option `User` is given, only those identifiers are returned that have been assigned a value or a property, respectively, by the user.

Return Values

`set` of identifiers.

See Also

MuPAD Domains
`DOM_IDENT`

MuPAD Functions
`:=` | `_assign` | `assume`

and, _and

Logical “and”

Syntax

```
b1 and b2
```

```
_and(b1, b2, …)
```

Description

`b1 and b2` represents the logical ‘and’ of the Boolean expressions `b1`, `b2`.

MuPAD uses a three state logic with the Boolean constants `TRUE`, `FALSE`, and `UNKNOWN`. These are processed as follows:

and	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

Boolean expressions may be composed of these constants as well as of arbitrary arithmetical expressions. Typically, equations such as `x = y` and inequalities such as `x <> y`, `x < y`, `x <= y` etc. are used to construct Boolean expressions.

`_and(b1, b2, ...)` is equivalent to `b1 and b2 and ...`. This expression represents `TRUE` if each single expression evaluates to `TRUE`. It represents `FALSE` if at least one expression evaluates to `FALSE`. It represents `UNKNOWN` if at least one expression evaluates to `UNKNOWN` and all others evaluate to `TRUE`.

`_and()` returns `TRUE`.

Combinations of the constants `TRUE`, `FALSE`, `UNKNOWN` inside a Boolean expression are simplified automatically. However, symbolic Boolean subexpressions, equalities, and inequalities are not evaluated and simplified by logical operators. Use `bool` to evaluate such expressions to one of the Boolean constants. Note, however, that `bool` can

evaluate inequalities `x < y`, `x <= y` etc. only if they are composed of numbers of type `Type::Real`. Cf. “Example 2” on page 1-143.

Use `simplify` with the option `logic` to simplify expressions involving symbolic Boolean subexpressions. Cf. “Example 3” on page 1-144.

The precedences of the logical operators are as follows: The operator `not` is stronger binding than `and`, i.e, `not b1 and b2` = `(not b1) and b2`. The operator `and` is stronger binding than `xor`, i.e., `b1 and b2 or b3` = `(b1 and b2) xor b3`. The operator `xor` is stronger binding than `or`, i.e., `b1 xor b2 or b3` = `(b1 xor b2) or b3`. The operator `or` is stronger binding than `==>`, i.e., `b1 or b2 ==> b3` = `(b1 or b2) ==> b3`. The operator `==>` is stronger binding than `<=>`, i.e., `b1 ==> b2 <=> b3` = `(b1 ==> b2) <=> b3`.

If in doubt, use brackets to make sure that the expression is parsed as desired.

In the conditional context of `if`, `repeat`, and `while` statements, Boolean expressions are evaluated via “lazy evaluation” (see `_lazy_and`, `_lazy_or`). In any other context, all operands are evaluated.

Examples

Example 1

Combinations of the Boolean constants `TRUE`, `FALSE`, and `UNKNOWN` are simplified automatically to one of these constants:

```
TRUE and not (FALSE or TRUE)
```

```
    FALSE
```

```
FALSE and UNKNOWN, TRUE and UNKNOWN
```

```
    FALSE, UNKNOWN
```

```
FALSE or UNKNOWN, TRUE or UNKNOWN
```

```
    UNKNOWN, TRUE
```

```
not UNKNOWN
```

UNKNOWN

Example 2

Logical operators simplify subexpressions that evaluate to the constants TRUE, FALSE, UNKNOWN.

```
b1 or b2 and TRUE
```

$b1 \vee b2$

```
FALSE or ((not b1) and TRUE)
```

$\neg b1$

```
b1 and (b2 or FALSE) and UNKNOWN
```

$UNKNOWN \wedge b1 \wedge b2$

```
FALSE or (b1 and UNKNOWN) or x < 1
```

$(UNKNOWN \wedge b1) \vee x < 1$

```
TRUE and ((b1 and FALSE) or (b1 and TRUE))
```

$b1$

However, equalities and inequalities are not evaluated:

```
(x = x) and (1 < 2) and (2 < 3) and (3 < 4)
```

$x = x \wedge 1 < 2 \wedge 2 < 3 \wedge 3 < 4$

Boolean evaluation is enforced via `bool`:

```
bool(%)
```

```
TRUE
```

Example 3

Expressions involving symbolic Boolean subexpressions are not simplified by `and`, `or`, `not`. Simplification has to be requested explicitly via the function `simplify`:

```
(b1 and b2) or (b1 and (not b2)) and (1 < 2)
```

$$(b1 \wedge b2) \vee (b1 \wedge \neg b2 \wedge 1 < 2)$$

```
simplify(%, logic)
```

```
b1
```

Example 4

The Boolean functions `_and` and `_or` accept arbitrary sequences of Boolean expressions. The following call uses `isprime` to check whether *all* elements of the given set are prime:

```
set := {1987, 1993, 1997, 1999, 2001}:
_and(isprime(i) $ i in set)
```

```
FALSE
```

The following call checks whether *at least one* of the numbers is prime:

```
_or(isprime(i) $ i in set)
```

```
TRUE
```

```
delete set:
```

Parameters

b_1, b_2, …

Boolean expressions

Return Values

Boolean expression.

Overloaded By

b, b_1, b_2

See Also

MuPAD Functions

`<=>` | `==>` | `_lazy_and` | `_lazy_or` | `bool` | `FALSE` | `is` | `not` | `or` | `TRUE` | `UNKNOWN` | `xor`

or, _or

Logical “or”

Syntax

```
b1 or b2
```

```
_or(b1, b2, …)
```

Description

`b1 or b2` represents the non-exclusive logical ‘or’ of the Boolean expressions `b1`, `b2`.

MuPAD uses a three state logic with the Boolean constants `TRUE`, `FALSE`, and `UNKNOWN`. These are processed as follows:

or	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Boolean expressions may be composed of these constants as well as of arbitrary arithmetical expressions. Typically, equations such as `x = y` and inequalities such as `x <> y`, `x < y`, `x <= y` etc. are used to construct Boolean expressions.

`_or(b1, b2, ...)` is equivalent to `b1 or b2 or ...`. This expression represents `FALSE` if each single expression evaluates to `FALSE`. It represents `TRUE` if at least one expression evaluates to `TRUE`. It represents `UNKNOWN` if at least one expression evaluates to `UNKNOWN` and all others evaluate to `FALSE`.

`_or()` returns `FALSE`.

Combinations of the constants `TRUE`, `FALSE`, `UNKNOWN` inside a Boolean expression are simplified automatically. However, symbolic Boolean subexpressions, equalities, and inequalities are not evaluated and simplified by logical operators. Use `bool` to evaluate such expressions to one of the Boolean constants. Note, however, that `bool` can

evaluate inequalities `x < y`, `x <= y` etc. only if they are composed of numbers of type `Type::Real`. Cf. “Example 2” on page 1-148.

Use `simplify` with the option `logic` to simplify expressions involving symbolic Boolean subexpressions. Cf. “Example 3” on page 1-149.

The precedences of the logical operators are as follows: The operator `not` is stronger binding than `and`, i.e, `not b1 and b2` = `(not b1) and b2`. The operator `and` is stronger binding than `xor`, i.e., `b1 and b2 or b3` = `(b1 and b2) xor b3`. The operator `xor` is stronger binding than `or`, i.e., `b1 xor b2 or b3` = `(b1 xor b2) or b3`. The operator `or` is stronger binding than `==>`, i.e., `b1 or b2 ==> b3` = `(b1 or b2) ==> b3`. The operator `==>` is stronger binding than `<=>`, i.e., `b1 ==> b2 <=> b3` = `(b1 ==> b2) <=> b3`.

If in doubt, use brackets to make sure that the expression is parsed as desired.

In the conditional context of `if`, `repeat`, and `while` statements, Boolean expressions are evaluated via “lazy evaluation” (see `_lazy_and`, `_lazy_or`). In any other context, all operands are evaluated.

Examples

Example 1

Combinations of the Boolean constants `TRUE`, `FALSE`, and `UNKNOWN` are simplified automatically to one of these constants:

```
TRUE and not (FALSE or TRUE)
```

```
     FALSE
```

```
FALSE and UNKNOWN, TRUE and UNKNOWN
```

```
     FALSE, UNKNOWN
```

```
FALSE or UNKNOWN, TRUE or UNKNOWN
```

```
     UNKNOWN, TRUE
```

```
not UNKNOWN
```

$$\text{UNKNOWN}$$

Example 2

Logical operators simplify subexpressions that evaluate to the constants TRUE, FALSE, UNKNOWN.

```
b1 or b2 and TRUE
```

$$b1 \vee b2$$

```
FALSE or ((not b1) and TRUE)
```

$$\neg b1$$

```
b1 and (b2 or FALSE) and UNKNOWN
```

$$\text{UNKNOWN} \wedge b1 \wedge b2$$

```
FALSE or (b1 and UNKNOWN) or x < 1
```

$$(\text{UNKNOWN} \wedge b1) \vee x < 1$$

```
TRUE and ((b1 and FALSE) or (b1 and TRUE))
```

$$b1$$

However, equalities and inequalities are not evaluated:

```
(x = x) and (1 < 2) and (2 < 3) and (3 < 4)
```

$$x = x \wedge 1 < 2 \wedge 2 < 3 \wedge 3 < 4$$

Boolean evaluation is enforced via `bool`:

```
bool(%)
```

```
TRUE
```

Example 3

Expressions involving symbolic Boolean subexpressions are not simplified by `and`, `or`, `not`. Simplification has to be requested explicitly via the function `simplify`:

```
(b1 and b2) or (b1 and (not b2)) and (1 < 2)
```

$$(b1 \wedge b2) \vee (b1 \wedge \neg b2 \wedge 1 < 2)$$

```
simplify(%, logic)
```

```
b1
```

Example 4

The Boolean functions `_and` and `_or` accept arbitrary sequences of Boolean expressions. The following call uses `isprime` to check whether *all* elements of the given set are prime:

```
set := {1987, 1993, 1997, 1999, 2001}:
_and(isprime(i) $ i in set)
```

```
FALSE
```

The following call checks whether *at least one* of the numbers is prime:

```
_or(isprime(i) $ i in set)
```

```
TRUE
```

```
delete set:
```

Parameters

b_1, b_2, …

Boolean expressions

Return Values

Boolean expression.

Overloaded By

b, b_1, b_2

See Also

MuPAD Functions

`<=>` | `==>` | `_lazy_and` | `_lazy_or` | `and` | `bool` | `FALSE` | `is` | `not` | `TRUE` | `UNKNOWN` | `xor`

not, _not

Logical negation

Syntax

```
not b
```

```
_not(b)
```

Description

`not b` represents the logical negation of the Boolean expression `b`.

MuPAD uses a three state logic with the Boolean constants `TRUE`, `FALSE`, and `UNKNOWN`. These are processed as follows:

`not TRUE = FALSE`, `not FALSE = TRUE`, `not UNKNOWN = UNKNOWN` .

Boolean expressions may be composed of these constants as well as of arbitrary arithmetical expressions. Typically, equations such as `x = y` and inequalities such as `x <> y`, `x < y`, `x <= y` etc. are used to construct Boolean expressions.

`_not(b)` is equivalent to `not b`.

Combinations of the constants `TRUE`, `FALSE`, `UNKNOWN` inside a Boolean expression are simplified automatically. However, symbolic Boolean subexpressions, equalities, and inequalities are not evaluated and simplified by logical operators. Use `bool` to evaluate such expressions to one of the Boolean constants. Note, however, that `bool` can evaluate inequalities `x < y`, `x <= y` etc. only if they are composed of numbers of type `Type::Real`. Cf. “Example 2” on page 1-152.

Use `simplify` with the option `logic` to simplify expressions involving symbolic Boolean subexpressions. Cf. “Example 3” on page 1-153.

The precedences of the logical operators are as follows: The operator `not` is stronger binding than `and`, i.e, `not b1 and b2` = `(not b1) and b2`. The operator `and` is stronger binding than `xor`, i.e., `b1 and b2 or b3` = `(b1 and b2) xor b3`. The operator `xor` is stronger binding than `or`, i.e., `b1 xor b2 or b3` = `(b1 xor b2) or`

b3. The operator or is stronger binding than ==>, i.e., b1 or b2 ==> b3 = (b1 or b2) ==> b3. The operator ==> is stronger binding than <=>, i.e., b1 ==> b2 <=> b3 = (b1 ==> b2) <=> b3.

If in doubt, use brackets to make sure that the expression is parsed as desired.

In the conditional context of if, repeat, and while statements, Boolean expressions are evaluated via "lazy evaluation" (see _lazy_and, _lazy_or). In any other context, all operands are evaluated.

Examples

Example 1

Combinations of the Boolean constants TRUE, FALSE, and UNKNOWN are simplified automatically to one of these constants:

```
TRUE and not (FALSE or TRUE)
```

```
    FALSE
```

```
FALSE and UNKNOWN, TRUE and UNKNOWN
```

```
    FALSE, UNKNOWN
```

```
FALSE or UNKNOWN, TRUE or UNKNOWN
```

```
    UNKNOWN, TRUE
```

```
not UNKNOWN
```

```
    UNKNOWN
```

Example 2

Logical operators simplify subexpressions that evaluate to the constants TRUE, FALSE, UNKNOWN.

```
b1 or b2 and TRUE
```

$$b1 \vee b2$$

```
FALSE or ((not b1) and TRUE)
```

$$\neg b1$$

```
b1 and (b2 or FALSE) and UNKNOWN
```

$$UNKNOWN \wedge b1 \wedge b2$$

```
FALSE or (b1 and UNKNOWN) or x < 1
```

$$(UNKNOWN \wedge b1) \vee x < 1$$

```
TRUE and ((b1 and FALSE) or (b1 and TRUE))
```

$$b1$$

However, equalities and inequalities are not evaluated:

```
(x = x) and (1 < 2) and (2 < 3) and (3 < 4)
```

$$x = x \wedge 1 < 2 \wedge 2 < 3 \wedge 3 < 4$$

Boolean evaluation is enforced via `bool`:

```
bool(%)
```

$$TRUE$$

Example 3

Expressions involving symbolic Boolean subexpressions are not simplified by `and`, `or`, `not`. Simplification has to be requested explicitly via the function `simplify`:

```
(b1 and b2) or (b1 and (not b2)) and (1 < 2)
```

$$(b1 \wedge b2) \vee (b1 \wedge \neg b2 \wedge 1 < 2)$$

```
simplify(%, logic)
```

$$b1$$

Example 4

The Boolean functions `_and` and `_or` accept arbitrary sequences of Boolean expressions. The following call uses `isprime` to check whether *all* elements of the given set are prime:

```
set := {1987, 1993, 1997, 1999, 2001}:
_and(isprime(i) $ i in set)
```

FALSE

The following call checks whether *at least one* of the numbers is prime:

```
_or(isprime(i) $ i in set)
```

TRUE

```
delete set:
```

Parameters

b

Boolean expressions

Return Values

Boolean expression.

Overloaded By

b, b_1, b_2

See Also

MuPAD Functions

<=> | ==> | _lazy_and | _lazy_or | and | bool | FALSE | is | or | TRUE | UNKNOWN | xor

xor, _xor

Logical exclusive-or

Syntax

```
b1 xor b2
```

```
_xor(b1, b2, …)
```

Description

`b1 xor b2` represents the exclusive logical 'or' of the Boolean expressions `b1`, `b2`.

MuPAD uses a three state logic with the Boolean constants `TRUE`, `FALSE`, and `UNKNOWN`. These are processed as follows:

The operator `xor` is defined as follows: `a xor b` is equivalent to `(a or b) and not (a and b)`.

Boolean expressions may be composed of these constants as well as of arbitrary arithmetical expressions. Typically, equations such as `x = y` and inequalities such as `x <> y`, `x < y`, `x <= y` etc. are used to construct Boolean expressions.

`_xor(b1, b2, ...)` is equivalent to `b1 xor b2 xor ...`. This expression represents `TRUE` if an odd number of operands evaluate to `TRUE` and the others evaluate to `FALSE`. It represents `FALSE` if an even number of operands evaluate to `TRUE` and the others evaluate to `FALSE`. It represents `UNKNOWN` if at least one operand evaluates to `UNKNOWN`.

Combinations of the constants `TRUE`, `FALSE`, `UNKNOWN` inside a Boolean expression are simplified automatically. However, symbolic Boolean subexpressions, equalities, and inequalities are not evaluated and simplified by logical operators. Use `bool` to evaluate such expressions to one of the Boolean constants. Note, however, that `bool` can evaluate inequalities `x < y`, `x <= y` etc. only if they are composed of numbers of type `Type::Real`. Cf. "Example 2" on page 1-158.

Use `simplify` with the option `logic` to simplify expressions involving symbolic Boolean subexpressions. Cf. "Example 3" on page 1-159.

The precedences of the logical operators are as follows: The operator `not` is stronger binding than `and`, i.e, `not b1 and b2` = `(not b1) and b2`. The operator `and` is stronger binding than `xor`, i.e., `b1 and b2 or b3` = `(b1 and b2) xor b3`. The operator `xor` is stronger binding than `or`, i.e., `b1 xor b2 or b3` = `(b1 xor b2) or b3`. The operator `or` is stronger binding than `==>`, i.e., `b1 or b2 ==> b3` = `(b1 or b2) ==> b3`. The operator `==>` is stronger binding than `<=>`, i.e., `b1 ==> b2 <=> b3` = `(b1 ==> b2) <=> b3`.

If in doubt, use brackets to make sure that the expression is parsed as desired.

In the conditional context of `if`, `repeat`, and `while` statements, Boolean expressions are evaluated via "lazy evaluation" (see `_lazy_and`, `_lazy_or`). In any other context, all operands are evaluated.

Examples

Example 1

Combinations of the Boolean constants `TRUE`, `FALSE`, and `UNKNOWN` are simplified automatically to one of these constants:

```
TRUE and not (FALSE or TRUE)
```

```
    FALSE
```

```
FALSE and UNKNOWN, TRUE and UNKNOWN
```

```
    FALSE, UNKNOWN
```

```
FALSE or UNKNOWN, TRUE or UNKNOWN
```

```
    UNKNOWN, TRUE
```

```
not UNKNOWN
```

```
    UNKNOWN
```

Example 2

Logical operators simplify subexpressions that evaluate to the constants TRUE, FALSE, UNKNOWN.

```
b1 or b2 and TRUE
```

$$b1 \vee b2$$

```
FALSE or ((not b1) and TRUE)
```

$$\neg b1$$

```
b1 and (b2 or FALSE) and UNKNOWN
```

$$UNKNOWN \wedge b1 \wedge b2$$

```
FALSE or (b1 and UNKNOWN) or x < 1
```

$$(UNKNOWN \wedge b1) \vee x < 1$$

```
TRUE and ((b1 and FALSE) or (b1 and TRUE))
```

$$b1$$

However, equalities and inequalities are not evaluated:

```
(x = x) and (1 < 2) and (2 < 3) and (3 < 4)
```

$$x = x \wedge 1 < 2 \wedge 2 < 3 \wedge 3 < 4$$

Boolean evaluation is enforced via `bool`:

```
bool(%)
```

$$TRUE$$

Example 3

Expressions involving symbolic Boolean subexpressions are not simplified by `and`, `or`, `not`. Simplification has to be requested explicitly via the function `simplify`:

```
(b1 and b2) or (b1 and (not b2)) and (1 < 2)
```

$$(b1 \wedge b2) \vee (b1 \wedge \neg b2 \wedge 1 < 2)$$

```
simplify(%, logic)
```

$$b1$$

Example 4

The Boolean functions `_and` and `_or` accept arbitrary sequences of Boolean expressions. The following call uses `isprime` to check whether *all* elements of the given set are prime:

```
set := {1987, 1993, 1997, 1999, 2001}:
_and(isprime(i) $ i in set)
```

FALSE

The following call checks whether *at least one* of the numbers is prime:

```
_or(isprime(i) $ i in set)
```

TRUE

```
delete set:
```

Parameters

b_1, b_2, …

Boolean expressions

Return Values

Boolean expression.

Overloaded By

b, b_1, b_2

See Also

MuPAD Functions

`<=>` | `==>` | `_lazy_and` | `_lazy_or` | `and` | `bool` | `FALSE` | `is` | `not` | `or` | `TRUE` | `UNKNOWN`

==>, _implies

Logical implication

Syntax

`b1 ==> b2`

`_implies(b1, b2)`

Description

`b1 ==> b2` represents the logical implication of the Boolean expressions `b1`, `b2`.

MuPAD uses a three state logic with the Boolean constants `TRUE`, `FALSE`, and `UNKNOWN`. These are processed as follows:

The operator `==>` is defined as follows: `a ==> b` is equivalent to `not a or b`.

Boolean expressions may be composed of these constants as well as of arbitrary arithmetical expressions. Typically, equations such as `x = y` and inequalities such as `x <> y`, `x < y`, `x <= y` etc. are used to construct Boolean expressions.

`_implies(a, b)` is equivalent to `a ==> b`.

Combinations of the constants `TRUE`, `FALSE`, `UNKNOWN` inside a Boolean expression are simplified automatically. However, symbolic Boolean subexpressions, equalities, and inequalities are not evaluated and simplified by logical operators. Use `bool` to evaluate such expressions to one of the Boolean constants. Note, however, that `bool` can evaluate inequalities `x < y`, `x <= y` etc. only if they are composed of numbers of type `Type::Real`. Cf. “Example 2” on page 1-162.

Use `simplify` with the option `logic` to simplify expressions involving symbolic Boolean subexpressions. Cf. “Example 3” on page 1-163.

The precedences of the logical operators are as follows: The operator `not` is stronger binding than `and`, i.e, `not b1 and b2` = `(not b1) and b2`. The operator `and` is stronger binding than `xor`, i.e., `b1 and b2 or b3` = `(b1 and b2) xor b3`. The operator `xor` is stronger binding than `or`, i.e., `b1 xor b2 or b3` = `(b1 xor b2) or`

b3. The operator or is stronger binding than ==>, i.e., b1 or b2 ==> b3 = (b1 or b2) ==> b3. The operator ==> is stronger binding than <=>, i.e., b1 ==> b2 <=> b3 = (b1 ==> b2) <=> b3.

If in doubt, use brackets to make sure that the expression is parsed as desired.

In the conditional context of if, repeat, and while statements, Boolean expressions are evaluated via "lazy evaluation" (see _lazy_and, _lazy_or). In any other context, all operands are evaluated.

Examples

Example 1

Combinations of the Boolean constants TRUE, FALSE, and UNKNOWN are simplified automatically to one of these constants:

```
TRUE and not (FALSE or TRUE)
```

```
    FALSE
```

```
FALSE and UNKNOWN, TRUE and UNKNOWN
```

```
    FALSE, UNKNOWN
```

```
FALSE or UNKNOWN, TRUE or UNKNOWN
```

```
    UNKNOWN, TRUE
```

```
not UNKNOWN
```

```
    UNKNOWN
```

Example 2

Logical operators simplify subexpressions that evaluate to the constants TRUE, FALSE, UNKNOWN.

```
b1 or b2 and TRUE
```

$$b1 \vee b2$$

```
FALSE or ((not b1) and TRUE)
```

$$\neg b1$$

```
b1 and (b2 or FALSE) and UNKNOWN
```

$$UNKNOWN \wedge b1 \wedge b2$$

```
FALSE or (b1 and UNKNOWN) or x < 1
```

$$(UNKNOWN \wedge b1) \vee x < 1$$

```
TRUE and ((b1 and FALSE) or (b1 and TRUE))
```

$$b1$$

However, equalities and inequalities are not evaluated:

```
(x = x) and (1 < 2) and (2 < 3) and (3 < 4)
```

$$x = x \wedge 1 < 2 \wedge 2 < 3 \wedge 3 < 4$$

Boolean evaluation is enforced via `bool`:

```
bool(%)
```

$$TRUE$$

Example 3

Expressions involving symbolic Boolean subexpressions are not simplified by `and`, `or`, `not`. Simplification has to be requested explicitly via the function `simplify`:

```
(b1 and b2) or (b1 and (not b2)) and (1 < 2)
```

$$(b1 \wedge b2) \vee (b1 \wedge \neg b2 \wedge 1 < 2)$$

```
simplify(%, logic)
```

$$b1$$

Example 4

The Boolean functions `_and` and `_or` accept arbitrary sequences of Boolean expressions. The following call uses `isprime` to check whether *all* elements of the given set are prime:

```
set := {1987, 1993, 1997, 1999, 2001}:
_and(isprime(i) $ i in set)
```

$$FALSE$$

The following call checks whether *at least one* of the numbers is prime:

```
_or(isprime(i) $ i in set)
```

$$TRUE$$

```
delete set:
```

Parameters

b_1, b_2, …

Boolean expressions

Return Values

Boolean expression.

Overloaded By

b, b_1, b_2

See Also

MuPAD Functions

<=> | _lazy_and | _lazy_or | and | bool | FALSE | is | not | or | TRUE | UNKNOWN | xor

<=>, _equiv

Logical equivalence

Syntax

`b1 <=> b2`

`_equiv(b1, b2)`

Description

`b1 <=> b2` represents the logical equivalence of the Boolean expressions `b1`, `b2`.

MuPAD uses a three state logic with the Boolean constants `TRUE`, `FALSE`, and `UNKNOWN`. These are processed as follows:

The operator `<=>` is defined as follows: `a <=> b` is equivalent to `(a => b) and (b => a)`.

Boolean expressions may be composed of these constants as well as of arbitrary arithmetical expressions. Typically, equations such as `x = y` and inequalities such as `x <> y`, `x < y`, `x <= y` etc. are used to construct Boolean expressions.

`_equiv(a, b)` is equivalent to `a <=> b`.

Combinations of the constants `TRUE`, `FALSE`, `UNKNOWN` inside a Boolean expression are simplified automatically. However, symbolic Boolean subexpressions, equalities, and inequalities are not evaluated and simplified by logical operators. Use `bool` to evaluate such expressions to one of the Boolean constants. Note, however, that `bool` can evaluate inequalities `x < y`, `x <= y` etc. only if they are composed of numbers of type `Type::Real`. Cf. “Example 2” on page 1-167.

Use `simplify` with the option `logic` to simplify expressions involving symbolic Boolean subexpressions. Cf. “Example 3” on page 1-168.

The precedences of the logical operators are as follows: The operator `not` is stronger binding than `and`, i.e, `not b1 and b2` = `(not b1) and b2`. The operator `and` is stronger binding than `xor`, i.e., `b1 and b2 or b3` = `(b1 and b2) xor b3`. The

operator `xor` is stronger binding than `or`, i.e., `b1 xor b2 or b3` = `(b1 xor b2) or b3`. The operator `or` is stronger binding than `==>`, i.e., `b1 or b2 ==> b3` = `(b1 or b2) ==> b3`. The operator `==>` is stronger binding than `<=>`, i.e., `b1 ==> b2 <=> b3` = `(b1 ==> b2) <=> b3`.

If in doubt, use brackets to make sure that the expression is parsed as desired.

In the conditional context of `if`, `repeat`, and `while` statements, Boolean expressions are evaluated via "lazy evaluation" (see `_lazy_and`, `_lazy_or`). In any other context, all operands are evaluated.

Examples

Example 1

Combinations of the Boolean constants `TRUE`, `FALSE`, and `UNKNOWN` are simplified automatically to one of these constants:

```
TRUE and not (FALSE or TRUE)
```

```
    FALSE
```

```
FALSE and UNKNOWN, TRUE and UNKNOWN
```

```
    FALSE, UNKNOWN
```

```
FALSE or UNKNOWN, TRUE or UNKNOWN
```

```
    UNKNOWN, TRUE
```

```
not UNKNOWN
```

```
    UNKNOWN
```

Example 2

Logical operators simplify subexpressions that evaluate to the constants `TRUE`, `FALSE`, `UNKNOWN`.

```
b1 or b2 and TRUE
```

$$b1 \vee b2$$

```
FALSE or ((not b1) and TRUE)
```

$$\neg b1$$

```
b1 and (b2 or FALSE) and UNKNOWN
```

$$UNKNOWN \wedge b1 \wedge b2$$

```
FALSE or (b1 and UNKNOWN) or x < 1
```

$$(UNKNOWN \wedge b1) \vee x < 1$$

```
TRUE and ((b1 and FALSE) or (b1 and TRUE))
```

$$b1$$

However, equalities and inequalities are not evaluated:

```
(x = x) and (1 < 2) and (2 < 3) and (3 < 4)
```

$$x = x \wedge 1 < 2 \wedge 2 < 3 \wedge 3 < 4$$

Boolean evaluation is enforced via `bool`:

```
bool(%)
```

$$TRUE$$

Example 3

Expressions involving symbolic Boolean subexpressions are not simplified by `and`, `or`, `not`. Simplification has to be requested explicitly via the function `simplify`:

```
(b1 and b2) or (b1 and (not b2)) and (1 < 2)
```

$$(b1 \wedge b2) \vee (b1 \wedge \neg b2 \wedge 1 < 2)$$

```
simplify(%, logic)
```

$$b1$$

Example 4

The Boolean functions `_and` and `_or` accept arbitrary sequences of Boolean expressions. The following call uses `isprime` to check whether *all* elements of the given set are prime:

```
set := {1987, 1993, 1997, 1999, 2001}:
_and(isprime(i) $ i in set)
```

$$FALSE$$

The following call checks whether *at least one* of the numbers is prime:

```
_or(isprime(i) $ i in set)
```

$$TRUE$$

```
delete set:
```

Parameters

b_1, b_2, …

Boolean expressions

Return Values

Boolean expression.

Overloaded By

b, b_1, b_2

See Also

MuPAD Functions
`==>` | `_lazy_and` | `_lazy_or` | `and` | `bool` | `FALSE` | `is` | `not` | `or` | `TRUE` | `UNKNOWN` | `xor`

append

Add elements to a list

Syntax

```
append(l, object1, object2, …)
```

Description

`append(l, object)` adds `object` to the list `l`.

`append(l, object1, object2, ...)` appends `object1`, `object2`, etc. to the list `l` and returns the new list as the result.

`append(f(x), object1, object2, ...)` appends `object1`, `object2`, etc. to the expression `f(x)` and returns the new expression as the result.

The call `append(l)` is legal and returns `l`.

`append(l, object1, object2, ...)` is equivalent to both `[op(l), object1, object2, ...]` and `l.[object1, object2, ...]`. However, `append` is more efficient.

The function `append` always returns a new object. The first argument remains unchanged. See “Example 3” on page 1-172.

Examples

Example 1

The function `append` adds new elements to the end of a list:

```
append([a, b], c, d)
```

$$[a, b, c, d]$$

If no new elements are given, the first argument is returned unmodified:

```
l := [a, b]: append(l)
```

$$[a, b]$$

The first argument may be an empty list:

```
append([ ], c)
```

$$[c]$$

Example 2

The function append adds new elements to the end of an expression:

```
append(f(a, b), c, d)
```

$$f(a, b, c, d)$$

Expressions can be written in operator notation:

```
append(a + b, c)
```

$$a + b + c$$

Example 3

The function append always returns a new object. The first argument remains unchanged:

```
l := [a, b]: append(l, c, d), l
```

$$[a, b, c, d], [a, b]$$

Example 4

Users can overload append for their own domains. For illustration, we create a new domain T and supply it with an "append" slot, which simply adds the remaining arguments to the internal operands of its first argument:

```
T := newDomain("T"):
T::append := x -> new(T, extop(x), args(2..args(0))):
```

If we now call `append` with an object of domain type `T`, the slot routine `T::append` is invoked:

```
e := new(T, 1, 2): append(e, 3)
```

$$\mathrm{new}(T, 1, 2, 3)$$

Parameters

`1`

A list or an expression

`object1, object2, …`

Arbitrary MuPAD objects

Return Values

Extended list or expression.

Overloaded By

l

See Also

MuPAD Domains
`DOM_EXPR` | `DOM_LIST`

MuPAD Functions
`_concat` | `_index` | `op`

arcsin

Inverse sine function

Syntax

```
arcsin(x)
```

Description

`arcsin(x)` represents the inverse of the sine function.

The angle returned by this function is measured in radians, not in degrees. For example, the result π represents an angle of 180^o.

`arcsin` is defined for complex arguments.

Floating-point values are returned for floating-point arguments. Floating-point intervals are returned for interval arguments. Unevaluated function calls are returned for most exact arguments.

If the argument is a rational multiple of `I`, the result is expressed in terms of hyperbolic functions. See “Example 2” on page 1-176.

The inverse sine function is multivalued. The MuPAD `arcsin` function returns the value on the main branch. The branch cuts are the real intervals (- ∞, - 1) and (1, ∞). Thus, `arcsin` returns values, such that $y = arcsin(x)$ satisfies $-\frac{\pi}{2} \le \Re(y) \le \frac{\pi}{2}$ for any finite complex x

The `sin` function returns explicit values for arguments that are certain rational multiples of π. For these values, `arcsin` returns an appropriate rational multiple of π on the main branch. See “Example 3” on page 1-176.

The values jump when the arguments cross a branch cut. See “Example 4” on page 1-177.

The float attributes are kernel functions. Thus, floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, `arcsin` is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
arcsin(1), arccos(1/sqrt(2)), arctan(5 + I), arccsc(1/3),
arcsec(I), arccot(1)
```

$$\frac{\pi}{2},\ \frac{\pi}{4},\ \arctan(5+i),\ \arcsin(3),\ \frac{\pi}{2}+\operatorname{arcsinh}(1)\,i,\ \frac{\pi}{4}$$

```
arcsin(-x), arccos(x + 1), arctan(1/x)
```

$$-\arcsin(x),\ \arccos(x+1),\ \arctan\left(\frac{1}{x}\right)$$

Floating-point values are computed for floating-point arguments:

```
arcsin(0.1234), arccos(5.6 + 7.8*I), arccot(1.0/10^20)
```

$$0.1237153458,\ 0.950687977-2.956002937\,i,\ 1.570796327$$

On input of floating-point intervals, these functions compute floating-point intervals containing the image sets:

```
arcsin(0...1), arccos(0...1)
```

$$0.0\ldots 1.570796327,\ -3.469446952\,10^{-18}\ldots 1.570796327$$

```
arcsin(2...3)
```

$$1.570796326\ldots 1.570796327+-1.762747175\ldots -1.316957896\,i$$

Note that certain types of input lead to severe overestimation, sometimes returning the whole image set of the function in question:

```
arccsc(-2...2);
csc(arccsc(-2...2))
```

$$-3.141592654 \ldots 2.382564905\,10^{-323228497} + -0.6931471806 \ldots \mathrm{RD\_INF}\,i$$

$$\cup -2.382564905\,10^{-323228497} \ldots 3.141592654 + \mathrm{RD\_NINF} \ldots 0.6931471806\,i$$

$$\mathrm{RD\_NINF} \ldots \mathrm{RD\_INF} + \mathrm{RD\_NINF} \ldots \mathrm{RD\_INF}\,i$$

Example 2

Arguments that are rational multiples of `I` are rewritten in terms of hyperbolic functions:

```
arcsin(5*I), arccos(5/4*I), arctan(-3*I)
```

$$\operatorname{arcsinh}(5)\,i,\ \frac{\pi}{2} - \operatorname{arcsinh}\left(\frac{5}{4}\right) i,\ -\operatorname{arctanh}(3)\,i$$

For other complex arguments unevaluated function calls without simplifications are returned:

```
arcsin(1/2^(1/2) + I), arccos(1 -3*I)
```

$$\arcsin\left(\frac{\sqrt{2}}{2} + i\right),\ \arccos(1 - 3\,i)$$

Example 3

Some special values are implemented:

```
arcsin(1/sqrt(2)), arccos((5^(1/2) - 1)/4), arctan(3^(1/2) - 2)
```

$$\frac{\pi}{4},\ \frac{2\,\pi}{5},\ -\frac{\pi}{12}$$

Such simplifications occur for arguments that are trigonometric images of rational multiples of π:

```
sin(9/10*PI), arcsin(sin(9/10*PI))
```

$$\frac{\sqrt{5}}{4} - \frac{1}{4},\ \frac{\pi}{10}$$

```
cos(PI/8)/sin(PI/8), arctan(cos(PI/8)/sin(PI/8))
```

$$\frac{\sqrt{\sqrt{2}+2}}{\sqrt{2-\sqrt{2}}},\ \frac{3\,\pi}{8}$$

Example 4

The values jump when crossing a branch cut:

```
arcsin(2.0 + I/10^10), arcsin(2.0 - I/10^10)
```

$$1.570796327 + 1.316957897\,i,\ 1.570796327 - 1.316957897\,i$$

On the branch cut, the values of `arcsin` coincide with the limit "from below" for real arguments $x > 1$. The values coincide with the limit "from above" for real $x < -1$:

```
arcsin(1.2), arcsin(1.2 - I/10^10), arcsin(1.2 + I/10^10)
```

$$1.570796327 - 0.6223625037\,i,\ 1.570796327 - 0.6223625037\,i,\ 1.570796327 + 0.6223625037\,i$$

```
arcsin(-1.2), arcsin(-1.2 + I/10^10), arcsin(-1.2 - I/10^10)
```

$$-1.570796327 + 0.6223625037\,i,\ -1.570796327 + 0.6223625037\,i,\ -1.570796327 - 0.6223625037\,i$$

Example 5

The inverse trigonometric functions can be rewritten in terms of the logarithm function with complex arguments:

```
rewrite(arcsin(x), ln), rewrite(arctan(x), ln)
```

$$-\ln\left(\sqrt{1-x^2}+x\,\mathrm{i}\right)\mathrm{i},\ \frac{\ln(1-x\,\mathrm{i})\,\mathrm{i}}{2}-\frac{\ln(1+x\,\mathrm{i})\,\mathrm{i}}{2}$$

Example 6

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the inverse trigonometric functions:

```
diff(arcsin(x^2), x), float(arccos(3)*arctan(5 + I))
```

$$\frac{2x}{\sqrt{1-x^4}},\ -0.06540673615+2.433548516\,\mathrm{i}$$

```
limit(arcsin(x^2)/arctan(x^2), x = 0)
```

$$1$$

```
series(arctan(sin(x)) - arcsin(tan(x)), x = 0, 10)
```

$$-x^3-\frac{83\,x^7}{120}-\frac{4\,x^9}{189}-\frac{22831\,x^{11}}{28800}+O(x^{13})$$

```
series(arccos(2 + x), x, 3)
```

$$-\mathrm{signIm}(x+2)\,\arccos(2)-\frac{\sqrt{3}\,x\,\mathrm{signIm}(x+2)\,\mathrm{i}}{3}+\frac{\sqrt{3}\,x^2\,\mathrm{signIm}(x+2)\,\mathrm{i}}{9}+O(x^3)$$

Example 7

When you call `arctan` with two arguments, MuPAD calls the `arg` function and computes the polar angle of a complex number:

```
arctan(y, x)
```

$$\arg(x+y\,\mathrm{i})$$

Parameters

x

Arithmetical expression or floating-point interval

Return Values

Arithmetical expression or floating-point interval.

Overloaded By

x

See Also

MuPAD Functions

`arccos` | `arccot` | `arccsc` | `arcsec` | `arctan` | `arg` | `cos` | `cot` | `csc` | `sec` | `sin` | `tan`

arccos

Inverse cosine function

Syntax

```
arccos(x)
```

Description

`arccos(x)` represents the inverse of the cosine function.

The angle returned by this function is measured in radians, not in degrees. For example, the result π represents an angle of 180^o.

`arccos` is defined for complex arguments.

Floating-point values are returned for floating-point arguments. Floating-point intervals are returned for interval arguments. Unevaluated function calls are returned for most exact arguments.

If the argument is a rational multiple of `I`, the result is expressed in terms of hyperbolic functions. See “Example 2” on page 1-182.

The inverse cosine function is multivalued. The MuPAD `arccos` function returns the value on the main branch. The branch cuts are the real intervals $(-\infty, -1)$ and $(1, \infty)$. Thus, `arccos` returns values, such that $y = arccos(x)$ satisfies $0 \le \Re(y) \le \pi$ for any finite complex x.

The `cos` function returns explicit values for arguments that are certain rational multiples of π. For these values, `arccos` returns an appropriate rational multiple of π on the main branch. See “Example 3” on page 1-182.

The values jump when the arguments cross a branch cut. See “Example 4” on page 1-183.

The float attributes are kernel functions. Thus, floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, `arccos` is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
arcsin(1), arccos(1/sqrt(2)), arctan(5 + I), arccsc(1/3),
arcsec(I), arccot(1)
```

$$\frac{\pi}{2},\ \frac{\pi}{4},\ \arctan(5+i),\ \arcsin(3),\ \frac{\pi}{2}+\operatorname{arcsinh}(1)\,i,\ \frac{\pi}{4}$$

```
arcsin(-x), arccos(x + 1), arctan(1/x)
```

$$-\arcsin(x),\ \arccos(x+1),\ \arctan\left(\frac{1}{x}\right)$$

Floating-point values are computed for floating-point arguments:

```
arcsin(0.1234), arccos(5.6 + 7.8*I), arccot(1.0/10^20)
```

$$0.1237153458,\ 0.950687977-2.956002937\,i,\ 1.570796327$$

On input of floating-point intervals, these functions compute floating-point intervals containing the image sets:

```
arcsin(0...1), arccos(0...1)
```

$$0.0\ldots 1.570796327,\ -3.469446952\,10^{-18}\ldots 1.570796327$$

```
arcsin(2...3)
```

$$1.570796326\ldots 1.570796327+-1.762747175\ldots -1.316957896\,i$$

Note that certain types of input lead to severe overestimation, sometimes returning the whole image set of the function in question:

```
arccsc(-2...2);
csc(arccsc(-2...2))
```

$$-3.141592654 \ldots 2.382564905\ 10^{-323228497} + -0.6931471806 \ldots \text{RD\_INF}\ i$$

$$\cup -2.382564905\ 10^{-323228497} \ldots 3.141592654 + \text{RD\_NINF} \ldots 0.6931471806\ i$$

$$\text{RD\_NINF} \ldots \text{RD\_INF} + \text{RD\_NINF} \ldots \text{RD\_INF}\ i$$

Example 2

Arguments that are rational multiples of `I` are rewritten in terms of hyperbolic functions:

```
arcsin(5*I), arccos(5/4*I), arctan(-3*I)
```

$$\operatorname{arcsinh}(5)\ i,\ \frac{\pi}{2} - \operatorname{arcsinh}\left(\frac{5}{4}\right) i,\ -\operatorname{arctanh}(3)\ i$$

For other complex arguments unevaluated function calls without simplifications are returned:

```
arcsin(1/2^(1/2) + I), arccos(1 -3*I)
```

$$\arcsin\left(\frac{\sqrt{2}}{2} + i\right),\ \arccos(1 - 3\ i)$$

Example 3

Some special values are implemented:

```
arcsin(1/sqrt(2)), arccos((5^(1/2) - 1)/4), arctan(3^(1/2) - 2)
```

$$\frac{\pi}{4},\ \frac{2\pi}{5},\ -\frac{\pi}{12}$$

Such simplifications occur for arguments that are trigonometric images of rational multiples of π:

```
sin(9/10*PI), arcsin(sin(9/10*PI))
```

$$\frac{\sqrt{5}}{4} - \frac{1}{4},\ \frac{\pi}{10}$$

```
cos(PI/8)/sin(PI/8), arctan(cos(PI/8)/sin(PI/8))
```

$$\frac{\sqrt{\sqrt{2}+2}}{\sqrt{2-\sqrt{2}}},\ \frac{3\,\pi}{8}$$

Example 4

The values jump when crossing a branch cut:

```
arcsin(2.0 + I/10^10), arcsin(2.0 - I/10^10)
```

$$1.570796327 + 1.316957897\,i,\ 1.570796327 - 1.316957897\,i$$

On the branch cut, the values of `arcsin` coincide with the limit "from below" for real arguments $x > 1$. The values coincide with the limit "from above" for real $x < -1$:

```
arcsin(1.2), arcsin(1.2 - I/10^10), arcsin(1.2 + I/10^10)
```

$$1.570796327 - 0.6223625037\,i,\ 1.570796327 - 0.6223625037\,i,\ 1.570796327 + 0.6223625037\,i$$

```
arcsin(-1.2), arcsin(-1.2 + I/10^10), arcsin(-1.2 - I/10^10)
```

$$-1.570796327 + 0.6223625037\,i,\ -1.570796327 + 0.6223625037\,i,\ -1.570796327 - 0.6223625037\,i$$

Example 5

The inverse trigonometric functions can be rewritten in terms of the logarithm function with complex arguments:

```
rewrite(arcsin(x), ln), rewrite(arctan(x), ln)
```

$$-\ln\left(\sqrt{1-x^2}+x\,\mathrm{i}\right)\mathrm{i},\ \frac{\ln(1-x\,\mathrm{i})\,\mathrm{i}}{2}-\frac{\ln(1+x\,\mathrm{i})\,\mathrm{i}}{2}$$

Example 6

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the inverse trigonometric functions:

```
diff(arcsin(x^2), x), float(arccos(3)*arctan(5 + I))
```

$$\frac{2x}{\sqrt{1-x^4}},\ -0.06540673615+2.433548516\,\mathrm{i}$$

```
limit(arcsin(x^2)/arctan(x^2), x = 0)
```

$$1$$

```
series(arctan(sin(x)) - arcsin(tan(x)), x = 0, 10)
```

$$-x^3-\frac{83\,x^7}{120}-\frac{4\,x^9}{189}-\frac{22831\,x^{11}}{28800}+O(x^{13})$$

```
series(arccos(2 + x), x, 3)
```

$$-\mathrm{signIm}(x+2)\,\arccos(2)-\frac{\sqrt{3}\,x\,\mathrm{signIm}(x+2)\,\mathrm{i}}{3}+\frac{\sqrt{3}\,x^2\,\mathrm{signIm}(x+2)\,\mathrm{i}}{9}+O(x^3)$$

Example 7

When you call `arctan` with two arguments, MuPAD calls the `arg` function and computes the polar angle of a complex number:

```
arctan(y, x)
```

$$\arg(x+y\,\mathrm{i})$$

Parameters

x

Arithmetical expression or floating-point interval

Return Values

Arithmetical expression or floating-point interval.

Overloaded By

x

See Also

MuPAD Functions

`arccot` | `arccsc` | `arcsec` | `arcsin` | `arctan` | `arg` | `cos` | `cot` | `csc` | `sec` | `sin` | `tan`

arctan

Inverse tangent function

Syntax

```
arctan(x)
```

```
arctan(y, x)
```

Description

`arctan(x)` represents the inverse of the tangent function.

`arctan(y, x)` is an alias for `arg(x, y)`.

The angle returned by this function is measured in radians, not in degrees. For example, the result π represents an angle of 180^o.

`arctan` is defined for complex arguments.

Floating-point values are returned for floating-point arguments. Floating-point intervals are returned for interval arguments. Unevaluated function calls are returned for most exact arguments.

If the argument is a rational multiple of `I`, the result is expressed in terms of hyperbolic functions. See “Example 2” on page 1-188.

The inverse tangent function is multivalued. The MuPAD `arctan` function returns the value on the main branch. The branch cuts are the intervals *interval*(- i ∞, [- i]) and *interval*([i], i ∞) on the imaginary axis. Thus, `arctan` returns values, such that y = *arctan*(x) satisfies $-\frac{\pi}{2} < \Re(y) < \frac{\pi}{2}$ for any finite complex x.

The `tan` function returns explicit values for arguments that are certain rational multiples of π. For these values, `arctan` returns an appropriate rational multiple of π on the main branch. See “Example 3” on page 1-189.

The values jump when the arguments cross a branch cut. See “Example 4” on page 1-189.

The float attributes are kernel functions. Thus, floating-point evaluation is fast.

If you call `arctan` with two arguments, `y` and `x`, MuPAD calls the `arg` function that computes the polar angle of a complex number `x + I*y`. See “Example 7” on page 1-191 and the `arg` help page.

Environment Interactions

When called with a floating-point argument, `arctan` is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
arcsin(1), arccos(1/sqrt(2)), arctan(5 + I), arccsc(1/3),
arcsec(I), arccot(1)
```

$$\frac{\pi}{2},\ \frac{\pi}{4},\ \arctan(5+i),\ \arcsin(3),\ \frac{\pi}{2}+\operatorname{arcsinh}(1)\,i,\ \frac{\pi}{4}$$

```
arcsin(-x), arccos(x + 1), arctan(1/x)
```

$$-\arcsin(x),\ \arccos(x+1),\ \arctan\left(\frac{1}{x}\right)$$

Floating-point values are computed for floating-point arguments:

```
arcsin(0.1234), arccos(5.6 + 7.8*I), arccot(1.0/10^20)
```

$$0.1237153458,\ 0.950687977-2.956002937\,i,\ 1.570796327$$

On input of floating-point intervals, these functions compute floating-point intervals containing the image sets:

```
arcsin(0...1), arccos(0...1)
```

$$0.0 \ldots 1.570796327,\ -3.469446952\ 10^{-18} \ldots 1.570796327$$

```
arcsin(2...3)
```

$$1.570796326 \ldots 1.570796327 + -1.762747175 \ldots -1.316957896\ i$$

Note that certain types of input lead to severe overestimation, sometimes returning the whole image set of the function in question:

```
arccsc(-2...2);
csc(arccsc(-2...2))
```

$$-3.141592654 \ldots 2.382564905\ 10^{-323228497} + -0.6931471806 \ldots \text{RD\_INF}\ i$$

$$\cup\ -2.382564905\ 10^{-323228497} \ldots 3.141592654 + \text{RD\_NINF} \ldots 0.6931471806\ i$$

$$\text{RD\_NINF} \ldots \text{RD\_INF} + \text{RD\_NINF} \ldots \text{RD\_INF}\ i$$

Example 2

Arguments that are rational multiples of `I` are rewritten in terms of hyperbolic functions:

```
arcsin(5*I), arccos(5/4*I), arctan(-3*I)
```

$$\operatorname{arcsinh}(5)\ i,\ \frac{\pi}{2} - \operatorname{arcsinh}\left(\frac{5}{4}\right) i,\ -\operatorname{arctanh}(3)\ i$$

For other complex arguments unevaluated function calls without simplifications are returned:

```
arcsin(1/2^(1/2) + I), arccos(1 -3*I)
```

$$\arcsin\left(\frac{\sqrt{2}}{2} + i\right),\ \arccos(1 - 3\ i)$$

Example 3

Some special values are implemented:

```
arcsin(1/sqrt(2)), arccos((5^(1/2) - 1)/4), arctan(3^(1/2) - 2)
```

$$\frac{\pi}{4}, \frac{2\,\pi}{5}, -\frac{\pi}{12}$$

Such simplifications occur for arguments that are trigonometric images of rational multiples of π:

```
sin(9/10*PI), arcsin(sin(9/10*PI))
```

$$\frac{\sqrt{5}}{4} - \frac{1}{4}, \frac{\pi}{10}$$

```
cos(PI/8)/sin(PI/8), arctan(cos(PI/8)/sin(PI/8))
```

$$\frac{\sqrt{\sqrt{2}+2}}{\sqrt{2-\sqrt{2}}}, \frac{3\,\pi}{8}$$

Example 4

The values jump when crossing a branch cut:

```
arcsin(2.0 + I/10^10), arcsin(2.0 - I/10^10)
```

$$1.570796327 + 1.316957897\,i,\ 1.570796327 - 1.316957897\,i$$

On the branch cut, the values of `arcsin` coincide with the limit "from below" for real arguments $x > 1$. The values coincide with the limit "from above" for real $x < -1$:

```
arcsin(1.2), arcsin(1.2 - I/10^10), arcsin(1.2 + I/10^10)
```

$$1.570796327 - 0.6223625037\,i,\ 1.570796327 - 0.6223625037\,i,\ 1.570796327 + 0.6223625037\,i$$

```
arcsin(-1.2), arcsin(-1.2 + I/10^10), arcsin(-1.2 - I/10^10)
```

$$-1.570796327 + 0.6223625037\,i,\ -1.570796327 + 0.6223625037\,i,\ -1.570796327 - 0.6223625037\,i$$

Example 5

The inverse trigonometric functions can be rewritten in terms of the logarithm function with complex arguments:

```
rewrite(arcsin(x), ln), rewrite(arctan(x), ln)
```

$$-\ln\left(\sqrt{1-x^2} + x\,i\right) i,\ \frac{\ln(1-x\,i)\,i}{2} - \frac{\ln(1+x\,i)\,i}{2}$$

Example 6

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the inverse trigonometric functions:

```
diff(arcsin(x^2), x), float(arccos(3)*arctan(5 + I))
```

$$\frac{2x}{\sqrt{1-x^4}},\ -0.06540673615 + 2.433548516\,i$$

```
limit(arcsin(x^2)/arctan(x^2), x = 0)
```

$$1$$

```
series(arctan(sin(x)) - arcsin(tan(x)), x = 0, 10)
```

$$-x^3 - \frac{83\,x^7}{120} - \frac{4\,x^9}{189} - \frac{22831\,x^{11}}{28800} + O(x^{13})$$

```
series(arccos(2 + x), x, 3)
```

$$-\operatorname{signIm}(x+2)\arccos(2) - \frac{\sqrt{3}\,x\,\operatorname{signIm}(x+2)\,i}{3} + \frac{\sqrt{3}\,x^2\,\operatorname{signIm}(x+2)\,i}{9} + O(x^3)$$

Example 7

When you call `arctan` with two arguments, MuPAD calls the `arg` function and computes the polar angle of a complex number:

```
arctan(y, x)
```

$$\arg(x + y\,\mathrm{i})$$

Parameters

x

Arithmetical expression or floating-point interval

y, x

Arithmetical expressions representing real numbers

Return Values

Arithmetical expression or floating-point interval.

Overloaded By

x

See Also

MuPAD Functions

`arccos` | `arccot` | `arccsc` | `arcsec` | `arcsin` | `arg` | `cos` | `cot` | `csc` | `sec` | `sin` | `tan`

arccsc

Inverse cosecant function

Syntax

```
arccsc(x)
```

Description

`arccsc(x)` represents the inverse of the cosecant function.

The angle returned by this function is measured in radians, not in degrees. For example, the result π represents an angle of 180^o.

`arccsc` is defined for complex arguments.

Floating-point values are returned for floating-point arguments. Floating-point intervals are returned for interval arguments. Unevaluated function calls are returned for most exact arguments.

If the argument is a rational multiple of `I`, the result is expressed in terms of hyperbolic functions. See “Example 2” on page 1-194.

MuPAD rewrites `arccsc` as `arccsc(x) = arcsin(1/x)`.

The inverse cosecant functions is multivalued. The MuPAD `arccsc` function returns values on the main branch defined as follows. The branch cut is the real interval (- 1, 1)

The `arccsc` function returns explicit values for arguments that are certain rational multiples of π. For these values, the inverse functions return an appropriate rational multiple of π on the main branch. See “Example 3” on page 1-194.

The values jump when the arguments cross a branch cut. See “Example 4” on page 1-195.

The float attributes are kernel functions. Thus, floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, `arccsc` is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
arcsin(1), arccos(1/sqrt(2)), arctan(5 + I), arccsc(1/3),
arcsec(I), arccot(1)
```

$$\frac{\pi}{2},\ \frac{\pi}{4},\ \arctan(5+i),\ \arcsin(3),\ \frac{\pi}{2}+\operatorname{arcsinh}(1)\,i,\ \frac{\pi}{4}$$

```
arcsin(-x), arccos(x + 1), arctan(1/x)
```

$$-\arcsin(x),\ \arccos(x+1),\ \arctan\left(\frac{1}{x}\right)$$

Floating-point values are computed for floating-point arguments:

```
arcsin(0.1234), arccos(5.6 + 7.8*I), arccot(1.0/10^20)
```

$$0.1237153458,\ 0.950687977-2.956002937\,i,\ 1.570796327$$

On input of floating-point intervals, these functions compute floating-point intervals containing the image sets:

```
arcsin(0...1), arccos(0...1)
```

$$0.0\ldots 1.570796327,\ -3.469446952\,10^{-18}\ldots 1.570796327$$

```
arcsin(2...3)
```

$$1.570796326\ldots 1.570796327+-1.762747175\ldots -1.316957896\,i$$

Note that certain types of input lead to severe overestimation, sometimes returning the whole image set of the function in question:

```
arccsc(-2...2);
csc(arccsc(-2...2))
```

$$-3.141592654 \ldots 2.382564905\ 10^{-323228497} + -0.6931471806 \ldots \text{RD\_INF}\ i$$

$$\cup -2.382564905\ 10^{-323228497} \ldots 3.141592654 + \text{RD\_NINF} \ldots 0.6931471806\ i$$

$$\text{RD\_NINF} \ldots \text{RD\_INF} + \text{RD\_NINF} \ldots \text{RD\_INF}\ i$$

Example 2

Arguments that are rational multiples of `I` are rewritten in terms of hyperbolic functions:

```
arcsin(5*I), arccos(5/4*I), arctan(-3*I)
```

$$\operatorname{arcsinh}(5)\ i,\ \frac{\pi}{2} - \operatorname{arcsinh}\left(\frac{5}{4}\right) i,\ -\operatorname{arctanh}(3)\ i$$

For other complex arguments unevaluated function calls without simplifications are returned:

```
arcsin(1/2^(1/2) + I), arccos(1 -3*I)
```

$$\arcsin\left(\frac{\sqrt{2}}{2} + i\right),\ \arccos(1 - 3\ i)$$

Example 3

Some special values are implemented:

```
arcsin(1/sqrt(2)), arccos((5^(1/2) - 1)/4), arctan(3^(1/2) - 2)
```

$$\frac{\pi}{4},\ \frac{2\pi}{5},\ -\frac{\pi}{12}$$

Such simplifications occur for arguments that are trigonometric images of rational multiples of π:

```
sin(9/10*PI), arcsin(sin(9/10*PI))
```

$$\frac{\sqrt{5}}{4} - \frac{1}{4},\ \frac{\pi}{10}$$

```
cos(PI/8)/sin(PI/8), arctan(cos(PI/8)/sin(PI/8))
```

$$\frac{\sqrt{\sqrt{2}+2}}{\sqrt{2-\sqrt{2}}},\ \frac{3\,\pi}{8}$$

Example 4

The values jump when crossing a branch cut:

```
arcsin(2.0 + I/10^10), arcsin(2.0 - I/10^10)
```

$$1.570796327 + 1.316957897\,i,\ 1.570796327 - 1.316957897\,i$$

On the branch cut, the values of `arcsin` coincide with the limit "from below" for real arguments $x > 1$. The values coincide with the limit "from above" for real $x < -1$:

```
arcsin(1.2), arcsin(1.2 - I/10^10), arcsin(1.2 + I/10^10)
```

$$1.570796327 - 0.6223625037\,i,\ 1.570796327 - 0.6223625037\,i,\ 1.570796327 + 0.6223625037\,i$$

```
arcsin(-1.2), arcsin(-1.2 + I/10^10), arcsin(-1.2 - I/10^10)
```

$$-1.570796327 + 0.6223625037\,i,\ -1.570796327 + 0.6223625037\,i,\ -1.570796327 - 0.6223625037\,i$$

Example 5

The inverse trigonometric functions can be rewritten in terms of the logarithm function with complex arguments:

```
rewrite(arcsin(x), ln), rewrite(arctan(x), ln)
```

$$-\ln\left(\sqrt{1-x^2}+x\,\mathrm{i}\right)\mathrm{i},\ \frac{\ln(1-x\,\mathrm{i})\,\mathrm{i}}{2}-\frac{\ln(1+x\,\mathrm{i})\,\mathrm{i}}{2}$$

Example 6

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the inverse trigonometric functions:

```
diff(arcsin(x^2), x), float(arccos(3)*arctan(5 + I))
```

$$\frac{2x}{\sqrt{1-x^4}},\ -0.06540673615+2.433548516\,\mathrm{i}$$

```
limit(arcsin(x^2)/arctan(x^2), x = 0)
```

$$1$$

```
series(arctan(sin(x)) - arcsin(tan(x)), x = 0, 10)
```

$$-x^3-\frac{83\,x^7}{120}-\frac{4\,x^9}{189}-\frac{22831\,x^{11}}{28800}+O(x^{13})$$

```
series(arccos(2 + x), x, 3)
```

$$-\mathrm{signIm}(x+2)\,\arccos(2)-\frac{\sqrt{3}\,x\,\mathrm{signIm}(x+2)\,\mathrm{i}}{3}+\frac{\sqrt{3}\,x^2\,\mathrm{signIm}(x+2)\,\mathrm{i}}{9}+O(x^3)$$

Example 7

When you call `arctan` with two arguments, MuPAD calls the `arg` function and computes the polar angle of a complex number:

```
arctan(y, x)
```

$$\arg(x+y\,\mathrm{i})$$

Parameters

x

Arithmetical expression or floating-point interval

Return Values

Arithmetical expression or floating-point interval.

Overloaded By

x

See Also

MuPAD Functions

`arccos` | `arccot` | `arcsec` | `arcsin` | `arctan` | `arg` | `cos` | `cot` | `csc` | `sec` | `sin` | `tan`

arcsec

Inverse secant function

Syntax

```
arcsec(x)
```

Description

`arcsec(x)` represents the inverse of the secant function.

The angle returned by this function is measured in radians, not in degrees. For example, the result π represents an angle of 180^o.

`arcsec` is defined for complex arguments.

Floating-point values are returned for floating-point arguments. Floating-point intervals are returned for interval arguments. Unevaluated function calls are returned for most exact arguments.

If the argument is a rational multiple of `I`, the result is expressed in terms of hyperbolic functions. See “Example 2” on page 1-200.

MuPAD rewrites `arcsec` as `arcsec(x) = arccos(1/x)`.

The inverse secant function is multivalued. The MuPAD `arccsc` function returns values on the main branch defined as follows. The branch cut is the real interval (- 1, 1).

The `arcsec` function returns explicit values for arguments that are certain rational multiples of π. For these values, the inverse functions return an appropriate rational multiple of π on the main branch. See “Example 3” on page 1-200.

The values jump when the arguments cross a branch cut. See “Example 4” on page 1-201.

The float attributes are kernel functions. Thus, floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, `arcsec` is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
arcsin(1), arccos(1/sqrt(2)), arctan(5 + I), arccsc(1/3),
arcsec(I), arccot(1)
```

$$\frac{\pi}{2},\ \frac{\pi}{4},\ \arctan(5+i),\ \arcsin(3),\ \frac{\pi}{2}+\operatorname{arcsinh}(1)\,i,\ \frac{\pi}{4}$$

```
arcsin(-x), arccos(x + 1), arctan(1/x)
```

$$-\arcsin(x),\ \arccos(x+1),\ \arctan\left(\frac{1}{x}\right)$$

Floating-point values are computed for floating-point arguments:

```
arcsin(0.1234), arccos(5.6 + 7.8*I), arccot(1.0/10^20)
```

$$0.1237153458,\ 0.950687977-2.956002937\,i,\ 1.570796327$$

On input of floating-point intervals, these functions compute floating-point intervals containing the image sets:

```
arcsin(0...1), arccos(0...1)
```

$$0.0\ldots 1.570796327,\ -3.469446952\,10^{-18}\ldots 1.570796327$$

```
arcsin(2...3)
```

$$1.570796326\ldots 1.570796327+-1.762747175\ldots -1.316957896\,i$$

Note that certain types of input lead to severe overestimation, sometimes returning the whole image set of the function in question:

```
arccsc(-2...2);
csc(arccsc(-2...2))
```

$$-3.141592654 \ldots 2.382564905\ 10^{-323228497} + -0.6931471806 \ldots \mathrm{RD\_INF}\ i$$

$$\cup -2.382564905\ 10^{-323228497} \ldots 3.141592654 + \mathrm{RD\_NINF} \ldots 0.6931471806\ i$$

$$\mathrm{RD\_NINF} \ldots \mathrm{RD\_INF} + \mathrm{RD\_NINF} \ldots \mathrm{RD\_INF}\ i$$

Example 2

Arguments that are rational multiples of `I` are rewritten in terms of hyperbolic functions:

```
arcsin(5*I), arccos(5/4*I), arctan(-3*I)
```

$$\operatorname{arcsinh}(5)\ i,\ \frac{\pi}{2} - \operatorname{arcsinh}\left(\frac{5}{4}\right) i,\ -\operatorname{arctanh}(3)\ i$$

For other complex arguments unevaluated function calls without simplifications are returned:

```
arcsin(1/2^(1/2) + I), arccos(1 -3*I)
```

$$\arcsin\left(\frac{\sqrt{2}}{2} + i\right),\ \arccos(1 - 3\ i)$$

Example 3

Some special values are implemented:

```
arcsin(1/sqrt(2)), arccos((5^(1/2) - 1)/4), arctan(3^(1/2) - 2)
```

$$\frac{\pi}{4},\ \frac{2\pi}{5},\ -\frac{\pi}{12}$$

Such simplifications occur for arguments that are trigonometric images of rational multiples of π:

```
sin(9/10*PI), arcsin(sin(9/10*PI))
```

$$\frac{\sqrt{5}}{4} - \frac{1}{4},\ \frac{\pi}{10}$$

```
cos(PI/8)/sin(PI/8), arctan(cos(PI/8)/sin(PI/8))
```

$$\frac{\sqrt{\sqrt{2}+2}}{\sqrt{2-\sqrt{2}}},\ \frac{3\,\pi}{8}$$

Example 4

The values jump when crossing a branch cut:

```
arcsin(2.0 + I/10^10), arcsin(2.0 - I/10^10)
```

$$1.570796327 + 1.316957897\,i,\ 1.570796327 - 1.316957897\,i$$

On the branch cut, the values of `arcsin` coincide with the limit "from below" for real arguments $x > 1$. The values coincide with the limit "from above" for real $x < -1$:

```
arcsin(1.2), arcsin(1.2 - I/10^10), arcsin(1.2 + I/10^10)
```

$$1.570796327 - 0.6223625037\,i,\ 1.570796327 - 0.6223625037\,i,\ 1.570796327 + 0.6223625037\,i$$

```
arcsin(-1.2), arcsin(-1.2 + I/10^10), arcsin(-1.2 - I/10^10)
```

$$-1.570796327 + 0.6223625037\,i,\ -1.570796327 + 0.6223625037\,i,\ -1.570796327 - 0.6223625037\,i$$

Example 5

The inverse trigonometric functions can be rewritten in terms of the logarithm function with complex arguments:

```
rewrite(arcsin(x), ln), rewrite(arctan(x), ln)
```

$$-\ln\left(\sqrt{1-x^2}+x\,\mathrm{i}\right)\mathrm{i},\ \frac{\ln(1-x\,\mathrm{i})\,\mathrm{i}}{2}-\frac{\ln(1+x\,\mathrm{i})\,\mathrm{i}}{2}$$

Example 6

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the inverse trigonometric functions:

```
diff(arcsin(x^2), x), float(arccos(3)*arctan(5 + I))
```

$$\frac{2x}{\sqrt{1-x^4}},\ -0.06540673615+2.433548516\,\mathrm{i}$$

```
limit(arcsin(x^2)/arctan(x^2), x = 0)
```

$$1$$

```
series(arctan(sin(x)) - arcsin(tan(x)), x = 0, 10)
```

$$-x^3-\frac{83\,x^7}{120}-\frac{4\,x^9}{189}-\frac{22831\,x^{11}}{28800}+O(x^{13})$$

```
series(arccos(2 + x), x, 3)
```

$$-\mathrm{signIm}(x+2)\,\arccos(2)-\frac{\sqrt{3}\,x\,\mathrm{signIm}(x+2)\,\mathrm{i}}{3}+\frac{\sqrt{3}\,x^2\,\mathrm{signIm}(x+2)\,\mathrm{i}}{9}+O(x^3)$$

Example 7

When you call `arctan` with two arguments, MuPAD calls the `arg` function and computes the polar angle of a complex number:

```
arctan(y, x)
```

$$\arg(x+y\,\mathrm{i})$$

Parameters

x

Arithmetical expression or floating-point interval

Return Values

Arithmetical expression or floating-point interval.

Overloaded By

x

See Also

MuPAD Functions

`arccos` | `arccot` | `arccsc` | `arcsin` | `arctan` | `arg` | `cos` | `cot` | `csc` | `sec` | `sin` | `tan`

arccot

Inverse cotangent function

Syntax

```
arccot(x)
```

Description

`arccot(x)` represents the inverse of the cotangent function.

The angle returned by this function is measured in radians, not in degrees. E.g., the result π represents an angle of 180^o.

`arccot` is defined for complex arguments.

Floating-point values are returned for floating-point arguments. Floating-point intervals are returned for interval arguments. Unevaluated function calls are returned for most exact arguments.

If the argument is a rational multiple of `I`, the result is expressed in terms of hyperbolic functions. See “Example 2” on page 1-206.

The inverse cotangent function is multivalued. The MuPAD `arccot` function returns the value on the main branch. The branch cut is the interval [- *i*, *i*] on the imaginary axis. Thus, `arcsin` returns values, such that $y = arccot(x)$ satisfies $-\frac{\pi}{2} < \Re(y) \le \frac{\pi}{2}$ for any finite complex x.

The `cot` function returns explicit values for arguments that are certain rational multiples of π. For these values, `arccot` returns an appropriate rational multiple of π on the main branch. See “Example 3” on page 1-207.

The values jump when the arguments cross a branch cut. See “Example 4” on page 1-207.

Note: MuPAD defines `arccot` as `arccot(x) = arctan(1/x)`, although `arccot` may return an unevaluated function call and does not rewrite itself in terms of `arctan`. As

a consequence of this definition, the real line crosses the branch cut and `arccot` has a jump discontinuity at the origin!

The float attributes are kernel functions. Thus, floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, `arccot` is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
arcsin(1), arccos(1/sqrt(2)), arctan(5 + I), arccsc(1/3),
arcsec(I), arccot(1)
```

$$\frac{\pi}{2},\ \frac{\pi}{4},\ \arctan(5+i),\ \arcsin(3),\ \frac{\pi}{2}+\operatorname{arcsinh}(1)\,i,\ \frac{\pi}{4}$$

```
arcsin(-x), arccos(x + 1), arctan(1/x)
```

$$-\arcsin(x),\ \arccos(x+1),\ \arctan\left(\frac{1}{x}\right)$$

Floating-point values are computed for floating-point arguments:

```
arcsin(0.1234), arccos(5.6 + 7.8*I), arccot(1.0/10^20)
```

$$0.1237153458,\ 0.950687977-2.956002937\,i,\ 1.570796327$$

On input of floating-point intervals, these functions compute floating-point intervals containing the image sets:

```
arcsin(0...1), arccos(0...1)
```

$$0.0 \ldots 1.570796327,\ -3.469446952\ 10^{-18} \ldots 1.570796327$$

```
arcsin(2...3)
```

$$1.570796326 \ldots 1.570796327 + -1.762747175 \ldots -1.316957896\ i$$

Note that certain types of input lead to severe overestimation, sometimes returning the whole image set of the function in question:

```
arccsc(-2...2);
csc(arccsc(-2...2))
```

$$-3.141592654 \ldots 2.382564905\ 10^{-323228497} + -0.6931471806 \ldots \text{RD\_INF}\ i$$

$$\cup\ -2.382564905\ 10^{-323228497} \ldots 3.141592654 + \text{RD\_NINF} \ldots 0.6931471806\ i$$

$$\text{RD\_NINF} \ldots \text{RD\_INF} + \text{RD\_NINF} \ldots \text{RD\_INF}\ i$$

Example 2

Arguments that are rational multiples of `I` are rewritten in terms of hyperbolic functions:

```
arcsin(5*I), arccos(5/4*I), arctan(-3*I)
```

$$\operatorname{arcsinh}(5)\ i,\ \frac{\pi}{2} - \operatorname{arcsinh}\left(\frac{5}{4}\right) i,\ -\operatorname{arctanh}(3)\ i$$

For other complex arguments unevaluated function calls without simplifications are returned:

```
arcsin(1/2^(1/2) + I), arccos(1 -3*I)
```

$$\arcsin\left(\frac{\sqrt{2}}{2} + i\right),\ \arccos(1 - 3\ i)$$

Example 3

Some special values are implemented:

```
arcsin(1/sqrt(2)), arccos((5^(1/2) - 1)/4), arctan(3^(1/2) - 2)
```

$$\frac{\pi}{4}, \frac{2\,\pi}{5}, -\frac{\pi}{12}$$

Such simplifications occur for arguments that are trigonometric images of rational multiples of π:

```
sin(9/10*PI), arcsin(sin(9/10*PI))
```

$$\frac{\sqrt{5}}{4} - \frac{1}{4}, \frac{\pi}{10}$$

```
cos(PI/8)/sin(PI/8), arctan(cos(PI/8)/sin(PI/8))
```

$$\frac{\sqrt{\sqrt{2}+2}}{\sqrt{2-\sqrt{2}}}, \frac{3\,\pi}{8}$$

Example 4

The values jump when crossing a branch cut:

```
arcsin(2.0 + I/10^10), arcsin(2.0 - I/10^10)
```

$$1.570796327 + 1.316957897\,i,\ 1.570796327 - 1.316957897\,i$$

On the branch cut, the values of `arcsin` coincide with the limit "from below" for real arguments $x > 1$. The values coincide with the limit "from above" for real $x < -1$:

```
arcsin(1.2), arcsin(1.2 - I/10^10), arcsin(1.2 + I/10^10)
```

$$1.570796327 - 0.6223625037\,i,\ 1.570796327 - 0.6223625037\,i,\ 1.570796327 + 0.6223625037\,i$$

```
arcsin(-1.2), arcsin(-1.2 + I/10^10), arcsin(-1.2 - I/10^10)
```

$$-1.570796327 + 0.6223625037\,i,\ -1.570796327 + 0.6223625037\,i,\ -1.570796327 - 0.6223625037\,i$$

Example 5

The inverse trigonometric functions can be rewritten in terms of the logarithm function with complex arguments:

```
rewrite(arcsin(x), ln), rewrite(arctan(x), ln)
```

$$-\ln\left(\sqrt{1-x^2} + x\,i\right) i,\ \frac{\ln(1-x\,i)\,i}{2} - \frac{\ln(1+x\,i)\,i}{2}$$

Example 6

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the inverse trigonometric functions:

```
diff(arcsin(x^2), x), float(arccos(3)*arctan(5 + I))
```

$$\frac{2x}{\sqrt{1-x^4}},\ -0.06540673615 + 2.433548516\,i$$

```
limit(arcsin(x^2)/arctan(x^2), x = 0)
```

$$1$$

```
series(arctan(sin(x)) - arcsin(tan(x)), x = 0, 10)
```

$$-x^3 - \frac{83\,x^7}{120} - \frac{4\,x^9}{189} - \frac{22831\,x^{11}}{28800} + O(x^{13})$$

```
series(arccos(2 + x), x, 3)
```

$$-\operatorname{signIm}(x+2)\arccos(2) - \frac{\sqrt{3}\,x\,\operatorname{signIm}(x+2)\,i}{3} + \frac{\sqrt{3}\,x^2\,\operatorname{signIm}(x+2)\,i}{9} + O(x^3)$$

Example 7

When you call `arctan` with two arguments, MuPAD calls the `arg` function and computes the polar angle of a complex number:

```
arctan(y, x)
```

$$\arg(x + y\,\mathrm{i})$$

Parameters

x

Arithmetical expression or floating-point interval

Return Values

Arithmetical expression or floating-point interval.

Overloaded By

x

See Also

MuPAD Functions

`arccos` | `arccsc` | `arcsec` | `arcsin` | `arctan` | `arg` | `cos` | `cot` | `csc` | `sec` | `sin` | `tan`

arcsinh

Inverse of the hyperbolic sine function

Syntax

```
arcsinh(x)
```

Description

`arcsinh(x)` represents the inverse of the hyperbolic sine function.

`arcsinh` is defined for complex arguments.

Floating-point values are returned for floating-point arguments. Floating-point intervals are returned for floating-point interval arguments. Unevaluated function calls are returned for most exact arguments.

The following special value is implemented:

`arcsinh`(0) = 0

The inverse hyperbolic sine function is multivalued. The MuPAD implementation returns values on the main branch defined by the following restriction of the imaginary part. For any finite complex x,

$$-\frac{\pi}{2} \le \Im(\operatorname{arcsinh}(x)) \le \frac{\pi}{2},$$

The inverse hyperbolic sine function is implemented according to the following relation to the logarithm function: `arcsinh(x) = ln(x + sqrt(x^2 + 1))`. See “Example 2” on page 1-212.

Consequently, the branch cuts are the intervals $(-i\infty, -i)$ and $(i, i\infty)$ on the imaginary axis. The values jump when the argument crosses a branch cut. See “Example 3” on page 1-212.

The float attributes are kernel functions, and floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, `arcsinh` is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
arcsinh(1), arccosh(1/sqrt(3)), arctanh(5 + I), arccsch(1/3),
arcsech(I), arccoth(2)
```

$$\operatorname{arcsinh}(1),\ \operatorname{arccosh}\left(\frac{\sqrt{3}}{3}\right),\ \operatorname{arctanh}(5+i),\ \operatorname{arcsinh}(3),\ \operatorname{arccosh}(-i),\ \operatorname{arccoth}(2)$$

```
arcsinh(-x), arccosh(x + 1), arctanh(1/x)
```

$$-\operatorname{arcsinh}(x),\ \operatorname{arccosh}(x+1),\ \operatorname{arctanh}\left(\frac{1}{x}\right)$$

Floating-point values are computed for floating-point arguments:

```
arcsinh(0.1234), arccosh(5.6 + 7.8*I), arccoth(1.0/10^20)
```

$$0.1230889466,\ 2.956002937 + 0.950687977\,i,\ -1.570796327\,i$$

Floating-point intervals are returned for arguments of this type:

```
arccoth(0.5 ... 1.5), arcsinh(0.1234...0.12345)
```

$$0.2554128118 \ldots \mathrm{RD\_INF} + -1.570796327 \ldots -1.570796326\,i \cup 0.8047189562 \ldots \mathrm{RD\_INF},\ 0.1230889466 \ldots 0.1231385701$$

The inverse of the hyperbolic tangent function has real values only in the interval (- 1, 1):

```
arctanh(-1/2...0), arctanh(2...3)
```

$$-0.5493061444 \ldots 0.0,\ 0.202732554 \ldots 0.6931471806 + -1.570796327 \ldots -1.570796326\,i$$

Example 2

The inverse hyperbolic functions can be rewritten in terms of the logarithm function:

```
rewrite(arcsinh(x), ln), rewrite(arctanh(x), ln)
```

$$\ln\left(x+\sqrt{x^2+1}\right),\ \frac{\ln(x+1)}{2}-\frac{\ln(1-x)}{2}$$

Example 3

The values jump when crossing a branch cut:

```
arctanh(2.0 + I/10^10), arctanh(2.0 - I/10^10)
```

$$0.5493061443 + 1.570796327\,i,\ 0.5493061443 - 1.570796327\,i$$

On the branch cut, the values of `arctanh` coincide with the limit "from below" for real arguments $x > 1$. The values coincide with the limit "from above" for real $x < -1$:

```
arctanh(1.2), arctanh(1.2 - I/10^10), arctanh(1.2 + I/10^10)
```

$$1.198947636 - 1.570796327\,i,\ 1.198947636 - 1.570796327\,i,\ 1.198947636 + 1.570796327\,i$$

```
arctanh(-1.2), arctanh(-1.2 + I/10^10), arctanh(-1.2 - I/10^10)
```

$$-1.198947636 + 1.570796327\,i,\ -1.198947636 + 1.570796327\,i,\ -1.198947636 - 1.570796327\,i$$

Example 4

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the inverse hyperbolic functions:

```
diff(arcsinh(x^2), x), float(arccosh(3)*arctanh(5 + I))
```

$$\frac{2\,x}{\sqrt{x^4+1}},\ 0.3427241326 + 2.698556745\,i$$

```
limit(arcsinh(x)/arctanh(x), x = 0)
```

$$1$$

```
series(arctanh(sinh(x)) - arcsinh(tanh(x)), x = 0, 10)
```

$$x^3 + \frac{83\,x^7}{120} - \frac{4\,x^9}{189} + \frac{22831\,x^{11}}{28800} + O(x^{13})$$

Parameters

x

Arithmetical expression or floating-point interval

Return Values

Arithmetical expression or floating-point interval

Overloaded By

x

See Also

MuPAD Functions

`arccosh` | `arccoth` | `arccsch` | `arcsech` | `arctanh` | `cosh` | `coth` | `csch` | `sech` | `sinh` | `tanh`

arccosh

Inverse of the hyperbolic cosine function

Syntax

```
arccosh(x)
```

Description

`arccosh(x)` represents the inverse of the hyperbolic cosine function.

`arccosh` is defined for complex arguments.

Floating-point values are returned for floating-point arguments. Floating-point intervals are returned for floating-point interval arguments. Unevaluated function calls are returned for most exact arguments.

The following special values are implemented:

`arccosh`(1) = 0, `arccosh`$(0) = \frac{i\pi}{2}$, `arccosh`(- 1) = $i\,\pi$.

The inverse hyperbolic cosine function is multivalued. The MuPAD implementation returns values on the main branch defined by the following restriction of the imaginary part. For any finite complex x,

$$-\pi < \Im(\operatorname{arccosh}(x)) \le \pi$$

,

The inverse hyperbolic cosine function is implemented according to the following relation to the logarithm function: `arccosh(x) = ln(x + sqrt(x^2 - 1))`. See “Example 2” on page 1-216.

Consequently, the branch cuts are the real interval (-∞, 1) and the imaginary axis.

The values jump when the argument crosses a branch cut. See “Example 3” on page 1-216.

The float attributes are kernel functions, and floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, `arccosh` is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
arcsinh(1), arccosh(1/sqrt(3)), arctanh(5 + I), arccsch(1/3),
arcsech(I), arccoth(2)
```

$$\operatorname{arcsinh}(1),\ \operatorname{arccosh}\left(\frac{\sqrt{3}}{3}\right),\ \operatorname{arctanh}(5+i),\ \operatorname{arcsinh}(3),\ \operatorname{arccosh}(-i),\ \operatorname{arccoth}(2)$$

```
arcsinh(-x), arccosh(x + 1), arctanh(1/x)
```

$$-\operatorname{arcsinh}(x),\ \operatorname{arccosh}(x+1),\ \operatorname{arctanh}\left(\frac{1}{x}\right)$$

Floating-point values are computed for floating-point arguments:

```
arcsinh(0.1234), arccosh(5.6 + 7.8*I), arccoth(1.0/10^20)
```

$$0.1230889466,\ 2.956002937 + 0.950687977\,i,\ -1.570796327\,i$$

Floating-point intervals are returned for arguments of this type:

```
arccoth(0.5 ... 1.5), arcsinh(0.1234...0.12345)
```

$$0.2554128118 \ldots \mathrm{RD\_INF} + -1.570796327 \ldots -1.570796326\,i \cup 0.8047189562 \ldots \mathrm{RD\_INF},\ 0.1230889466 \ldots 0.1231385701$$

The inverse of the hyperbolic tangent function has real values only in the interval (- 1, 1):

```
arctanh(-1/2...0), arctanh(2...3)
```

$$-0.5493061444 \ldots 0.0,\ 0.202732554 \ldots 0.6931471806 + -1.570796327 \ldots -1.570796326\,i$$

Example 2

The inverse hyperbolic functions can be rewritten in terms of the logarithm function:

```
rewrite(arcsinh(x), ln), rewrite(arctanh(x), ln)
```

$$\ln\left(x+\sqrt{x^2+1}\right),\ \frac{\ln(x+1)}{2}-\frac{\ln(1-x)}{2}$$

Example 3

The values jump when crossing a branch cut:

```
arctanh(2.0 + I/10^10), arctanh(2.0 - I/10^10)
```

$$0.5493061443 + 1.570796327\,i,\ 0.5493061443 - 1.570796327\,i$$

On the branch cut, the values of `arctanh` coincide with the limit "from below" for real arguments $x > 1$. The values coincide with the limit "from above" for real $x < -1$:

```
arctanh(1.2), arctanh(1.2 - I/10^10), arctanh(1.2 + I/10^10)
```

$$1.198947636 - 1.570796327\,i,\ 1.198947636 - 1.570796327\,i,\ 1.198947636 + 1.570796327\,i$$

```
arctanh(-1.2), arctanh(-1.2 + I/10^10), arctanh(-1.2 - I/10^10)
```

$$-1.198947636 + 1.570796327\,i,\ -1.198947636 + 1.570796327\,i,\ -1.198947636 - 1.570796327\,i$$

Example 4

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the inverse hyperbolic functions:

```
diff(arcsinh(x^2), x), float(arccosh(3)*arctanh(5 + I))
```

$$\frac{2\,x}{\sqrt{x^4+1}},\ 0.3427241326 + 2.698556745\,i$$

```
limit(arcsinh(x)/arctanh(x), x = 0)
```

$$1$$

```
series(arctanh(sinh(x)) - arcsinh(tanh(x)), x = 0, 10)
```

$$x^3 + \frac{83\,x^7}{120} - \frac{4\,x^9}{189} + \frac{22831\,x^{11}}{28800} + O(x^{13})$$

Parameters

x

Arithmetical expression or floating-point interval

Return Values

Arithmetical expression or floating-point interval

Overloaded By

x

See Also

MuPAD Functions

`arccoth` | `arccsch` | `arcsech` | `arcsinh` | `arctanh` | `cosh` | `coth` | `csch` | `sech` | `sinh` | `tanh`

arctanh

Inverse of the hyperbolic tangent function

Syntax

```
arctanh(x)
```

Description

`arctanh(x)` represents the inverse of the hyperbolic tangent function.

`arctanh` is defined for complex arguments.

Floating-point values are returned for floating-point arguments. Floating-point intervals are returned for floating-point interval arguments. Unevaluated function calls are returned for most exact arguments.

The following special value is implemented:

`arctanh`(0) = 0

The inverse hyperbolic tangent function is multivalued. The MuPAD implementation returns values on the main branch defined by the following restriction of the imaginary part. For any finite complex x,

$$-\frac{\pi}{2} < \Im(\operatorname{arctanh}(x)) < \frac{\pi}{2},$$

The inverse hyperbolic tangent function is implemented according to the following relation to the logarithm function: `arctanh(x) = (ln(1 + x) - ln(1 - x))/2`. See “Example 2” on page 1-220.

Consequently, the branch cuts are the real intervals (-∞, -1) and (1, ∞).

The values jump when the argument crosses a branch cut. See “Example 3” on page 1-220.

The float attributes are kernel functions, and floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, `arctanh` is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
arcsinh(1), arccosh(1/sqrt(3)), arctanh(5 + I), arccsch(1/3),
arcsech(I), arccoth(2)
```

$$\operatorname{arcsinh}(1),\ \operatorname{arccosh}\left(\frac{\sqrt{3}}{3}\right),\ \operatorname{arctanh}(5+i),\ \operatorname{arcsinh}(3),\ \operatorname{arccosh}(-i),\ \operatorname{arccoth}(2)$$

```
arcsinh(-x), arccosh(x + 1), arctanh(1/x)
```

$$-\operatorname{arcsinh}(x),\ \operatorname{arccosh}(x+1),\ \operatorname{arctanh}\left(\frac{1}{x}\right)$$

Floating-point values are computed for floating-point arguments:

```
arcsinh(0.1234), arccosh(5.6 + 7.8*I), arccoth(1.0/10^20)
```

$$0.1230889466,\ 2.956002937 + 0.950687977\,i,\ -1.570796327\,i$$

Floating-point intervals are returned for arguments of this type:

```
arccoth(0.5 ... 1.5), arcsinh(0.1234...0.12345)
```

$$0.2554128118 \ldots \mathrm{RD\_INF} + -1.570796327 \ldots -1.570796326\,i \cup 0.8047189562 \ldots \mathrm{RD\_INF},\ 0.1230889466 \ldots 0.1231385701$$

The inverse of the hyperbolic tangent function has real values only in the interval (- 1, 1):

```
arctanh(-1/2...0), arctanh(2...3)
```

$$-0.5493061444 \ldots 0.0,\ 0.202732554 \ldots 0.6931471806 + -1.570796327 \ldots -1.570796326\,i$$

Example 2

The inverse hyperbolic functions can be rewritten in terms of the logarithm function:

```
rewrite(arcsinh(x), ln), rewrite(arctanh(x), ln)
```

$$\ln\left(x+\sqrt{x^2+1}\right),\ \frac{\ln(x+1)}{2}-\frac{\ln(1-x)}{2}$$

Example 3

The values jump when crossing a branch cut:

```
arctanh(2.0 + I/10^10), arctanh(2.0 - I/10^10)
```

$$0.5493061443 + 1.570796327\,i,\ 0.5493061443 - 1.570796327\,i$$

On the branch cut, the values of `arctanh` coincide with the limit "from below" for real arguments $x > 1$. The values coincide with the limit "from above" for real $x < -1$:

```
arctanh(1.2), arctanh(1.2 - I/10^10), arctanh(1.2 + I/10^10)
```

$$1.198947636 - 1.570796327\,i,\ 1.198947636 - 1.570796327\,i,\ 1.198947636 + 1.570796327\,i$$

```
arctanh(-1.2), arctanh(-1.2 + I/10^10), arctanh(-1.2 - I/10^10)
```

$$-1.198947636 + 1.570796327\,i,\ -1.198947636 + 1.570796327\,i,\ -1.198947636 - 1.570796327\,i$$

Example 4

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the inverse hyperbolic functions:

```
diff(arcsinh(x^2), x), float(arccosh(3)*arctanh(5 + I))
```

$$\frac{2\,x}{\sqrt{x^4+1}},\ 0.3427241326 + 2.698556745\,i$$

```
limit(arcsinh(x)/arctanh(x), x = 0)
```

$$1$$

```
series(arctanh(sinh(x)) - arcsinh(tanh(x)), x = 0, 10)
```

$$x^3 + \frac{83\,x^7}{120} - \frac{4\,x^9}{189} + \frac{22831\,x^{11}}{28800} + O(x^{13})$$

Parameters

x

Arithmetical expression or floating-point interval

Return Values

Arithmetical expression or floating-point interval

Overloaded By

x

See Also

MuPAD Functions

`arccosh` | `arccoth` | `arccsch` | `arcsech` | `arcsinh` | `cosh` | `coth` | `csch` | `sech` | `sinh` | `tanh`

arccsch

Inverse of the hyperbolic cosecant function

Syntax

```
arccsch(x)
```

Description

`arccsch(x)` represents the inverse of the hyperbolic cosecant function.

`arccsch` is defined for complex arguments.

Floating-point values are returned for floating-point arguments. Floating-point intervals are returned for floating-point interval arguments. Unevaluated function calls are returned for most exact arguments.

The inverse hyperbolic cosecant function is multivalued. MuPAD rewrites `arccsch` as `arccsch(x) = arcsinh(1/x)`. The MuPAD implementation for `arcsinh` returns values on the main branch defined by the following restriction of the imaginary part. For any finite complex x,

$$-\frac{\pi}{2} \le \Im(\operatorname{arcsinh}(x)) \le \frac{\pi}{2},$$

The inverse hyperbolic cosecant function is implemented according to the following relation to the logarithm function: `arccsch(x) = arcsinh(1/x)`. See “Example 2” on page 1-224.

Consequently, the branch cut is the interval (-i, i) on the imaginary axis.

The values jump when the argument crosses a branch cut. See “Example 3” on page 1-224.

The float attributes are kernel functions, and floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, `arccsch` is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
arcsinh(1), arccosh(1/sqrt(3)), arctanh(5 + I), arccsch(1/3),
arcsech(I), arccoth(2)
```

$$\operatorname{arcsinh}(1),\ \operatorname{arccosh}\left(\frac{\sqrt{3}}{3}\right),\ \operatorname{arctanh}(5+i),\ \operatorname{arcsinh}(3),\ \operatorname{arccosh}(-i),\ \operatorname{arccoth}(2)$$

```
arcsinh(-x), arccosh(x + 1), arctanh(1/x)
```

$$-\operatorname{arcsinh}(x),\ \operatorname{arccosh}(x+1),\ \operatorname{arctanh}\left(\frac{1}{x}\right)$$

Floating-point values are computed for floating-point arguments:

```
arcsinh(0.1234), arccosh(5.6 + 7.8*I), arccoth(1.0/10^20)
```

$$0.1230889466,\ 2.956002937 + 0.950687977\,i,\ -1.570796327\,i$$

Floating-point intervals are returned for arguments of this type:

```
arccoth(0.5 ... 1.5), arcsinh(0.1234...0.12345)
```

$$0.2554128118 \ldots \mathrm{RD\_INF} + -1.570796327 \ldots -1.570796326\,i \cup 0.8047189562 \ldots \mathrm{RD\_INF},\ 0.1230889466 \ldots 0.1231385701$$

The inverse of the hyperbolic tangent function has real values only in the interval (- 1, 1):

```
arctanh(-1/2...0), arctanh(2...3)
```

$$-0.5493061444 \ldots 0.0,\ 0.202732554 \ldots 0.6931471806 + -1.570796327 \ldots -1.570796326\,i$$

Example 2

The inverse hyperbolic functions can be rewritten in terms of the logarithm function:

```
rewrite(arcsinh(x), ln), rewrite(arctanh(x), ln)
```

$$\ln\left(x+\sqrt{x^2+1}\right),\ \frac{\ln(x+1)}{2}-\frac{\ln(1-x)}{2}$$

Example 3

The values jump when crossing a branch cut:

```
arctanh(2.0 + I/10^10), arctanh(2.0 - I/10^10)
```

$$0.5493061443 + 1.570796327\,i,\ 0.5493061443 - 1.570796327\,i$$

On the branch cut, the values of `arctanh` coincide with the limit "from below" for real arguments $x > 1$. The values coincide with the limit "from above" for real $x < -1$:

```
arctanh(1.2), arctanh(1.2 - I/10^10), arctanh(1.2 + I/10^10)
```

$$1.198947636 - 1.570796327\,i,\ 1.198947636 - 1.570796327\,i,\ 1.198947636 + 1.570796327\,i$$

```
arctanh(-1.2), arctanh(-1.2 + I/10^10), arctanh(-1.2 - I/10^10)
```

$$-1.198947636 + 1.570796327\,i,\ -1.198947636 + 1.570796327\,i,\ -1.198947636 - 1.570796327\,i$$

Example 4

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the inverse hyperbolic functions:

```
diff(arcsinh(x^2), x), float(arccosh(3)*arctanh(5 + I))
```

$$\frac{2\,x}{\sqrt{x^4+1}},\ 0.3427241326 + 2.698556745\,i$$

```
limit(arcsinh(x)/arctanh(x), x = 0)
```

$$1$$

```
series(arctanh(sinh(x)) - arcsinh(tanh(x)), x = 0, 10)
```

$$x^3 + \frac{83\,x^7}{120} - \frac{4\,x^9}{189} + \frac{22831\,x^{11}}{28800} + O(x^{13})$$

Parameters

x

Arithmetical expression or floating-point interval

Return Values

Arithmetical expression or floating-point interval

Overloaded By

x

See Also

MuPAD Functions

`arccosh` | `arccoth` | `arcsech` | `arcsinh` | `arctanh` | `cosh` | `coth` | `csch` | `sech` | `sinh` | `tanh`

arcsech

Inverse of the hyperbolic secant function

Syntax

```
arcsech(x)
```

Description

`arcsech(x)` represents the inverse of the hyperbolic secant function.

`arcsech` is defined for complex arguments.

Floating-point values are returned for floating-point arguments. Floating-point intervals are returned for floating-point interval arguments. Unevaluated function calls are returned for most exact arguments.

The inverse hyperbolic secant function is multivalued. MuPAD rewrites `arcsech` as `arcsech(x) = arccosh(1/x)`. The MuPAD implementation for `arccosh` returns values on the main branch defined by the following restriction of the imaginary part. For any finite complex x,

$$-\pi < \Im(\operatorname{arccosh}(x)) \le \pi$$,

The inverse hyperbolic secant function is implemented according to the following relation to the logarithm function: `arcsech(x) = arccosh(1/x)`. See “Example 2” on page 1-228.

Consequently, the branch cuts are the real intervals (-∞, 0) and (1, ∞) together with the imaginary axis.

The values jump when the argument crosses a branch cut. See “Example 3” on page 1-228.

The float attributes are kernel functions, and floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, `arcsech` is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
arcsinh(1), arccosh(1/sqrt(3)), arctanh(5 + I), arccsch(1/3),
arcsech(I), arccoth(2)
```

$$\operatorname{arcsinh}(1),\ \operatorname{arccosh}\left(\frac{\sqrt{3}}{3}\right),\ \operatorname{arctanh}(5+i),\ \operatorname{arcsinh}(3),\ \operatorname{arccosh}(-i),\ \operatorname{arccoth}(2)$$

```
arcsinh(-x), arccosh(x + 1), arctanh(1/x)
```

$$-\operatorname{arcsinh}(x),\ \operatorname{arccosh}(x+1),\ \operatorname{arctanh}\left(\frac{1}{x}\right)$$

Floating-point values are computed for floating-point arguments:

```
arcsinh(0.1234), arccosh(5.6 + 7.8*I), arccoth(1.0/10^20)
```

$$0.1230889466,\ 2.956002937 + 0.950687977\,i,\ -1.570796327\,i$$

Floating-point intervals are returned for arguments of this type:

```
arccoth(0.5 ... 1.5), arcsinh(0.1234...0.12345)
```

$$0.2554128118 \ldots \mathrm{RD\_INF} + -1.570796327 \ldots -1.570796326\,i \cup 0.8047189562 \ldots \mathrm{RD\_INF},\ 0.1230889466 \ldots 0.1231385701$$

The inverse of the hyperbolic tangent function has real values only in the interval (- 1, 1):

```
arctanh(-1/2...0), arctanh(2...3)
```

$$-0.5493061444 \ldots 0.0,\ 0.202732554 \ldots 0.6931471806 + -1.570796327 \ldots -1.570796326\,i$$

Example 2

The inverse hyperbolic functions can be rewritten in terms of the logarithm function:

```
rewrite(arcsinh(x), ln), rewrite(arctanh(x), ln)
```

$$\ln\left(x+\sqrt{x^2+1}\right),\ \frac{\ln(x+1)}{2}-\frac{\ln(1-x)}{2}$$

Example 3

The values jump when crossing a branch cut:

```
arctanh(2.0 + I/10^10), arctanh(2.0 - I/10^10)
```

$$0.5493061443 + 1.570796327\,i,\ 0.5493061443 - 1.570796327\,i$$

On the branch cut, the values of `arctanh` coincide with the limit "from below" for real arguments $x > 1$. The values coincide with the limit "from above" for real $x < -1$:

```
arctanh(1.2), arctanh(1.2 - I/10^10), arctanh(1.2 + I/10^10)
```

$$1.198947636 - 1.570796327\,i,\ 1.198947636 - 1.570796327\,i,\ 1.198947636 + 1.570796327\,i$$

```
arctanh(-1.2), arctanh(-1.2 + I/10^10), arctanh(-1.2 - I/10^10)
```

$$-1.198947636 + 1.570796327\,i,\ -1.198947636 + 1.570796327\,i,\ -1.198947636 - 1.570796327\,i$$

Example 4

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the inverse hyperbolic functions:

```
diff(arcsinh(x^2), x), float(arccosh(3)*arctanh(5 + I))
```

$$\frac{2\,x}{\sqrt{x^4+1}},\ 0.3427241326 + 2.698556745\,i$$

```
limit(arcsinh(x)/arctanh(x), x = 0)
```

$$1$$

```
series(arctanh(sinh(x)) - arcsinh(tanh(x)), x = 0, 10)
```

$$x^3 + \frac{83\,x^7}{120} - \frac{4\,x^9}{189} + \frac{22831\,x^{11}}{28800} + O(x^{13})$$

Parameters

x

Arithmetical expression or floating-point interval

Return Values

Arithmetical expression or floating-point interval

Overloaded By

x

See Also

MuPAD Functions

`arccosh` | `arccoth` | `arccsch` | `arcsinh` | `arctanh` | `cosh` | `coth` | `csch` | `sech` | `sinh` | `tanh`

arccoth

Inverse of the hyperbolic cotangent function

Syntax

```
arccoth(x)
```

Description

`arccoth(x)` represents the inverse of the hyperbolic cotangent function.

`arccoth` is defined for complex arguments.

Floating-point values are returned for floating-point arguments. Floating-point intervals are returned for floating-point interval arguments. Unevaluated function calls are returned for most exact arguments.

The following special value is implemented:

$\texttt{arccoth}(0) = \frac{i\pi}{2}$

The inverse hyperbolic cotangent function is multivalued. The MuPAD implementation returns values on the main branch defined by the following restriction of the imaginary part. For any finite complex x,

$$-\frac{\pi}{2} < \Im(\operatorname{arccoth}(x)) \le \frac{\pi}{2}.$$

The inverse hyperbolic cotangent function is implemented according to the following relation to the logarithm function: `arccoth(x) = arctanh(1/x)`. See “Example 2” on page 1-232.

Consequently, the branch cut is the real interval [-1, 1].

The values jump when the argument crosses a branch cut. See “Example 3” on page 1-232.

`arccoth` is defined by `arccoth(x) = arctanh(1/x)`. However, MuPAD does not automatically rewrite it in terms of `arctanh`.

The float attributes are kernel functions, and floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, `arccoth` is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
arcsinh(1), arccosh(1/sqrt(3)), arctanh(5 + I), arccsch(1/3),
arcsech(I), arccoth(2)
```

$$\operatorname{arcsinh}(1),\ \operatorname{arccosh}\left(\frac{\sqrt{3}}{3}\right),\ \operatorname{arctanh}(5+i),\ \operatorname{arcsinh}(3),\ \operatorname{arccosh}(-i),\ \operatorname{arccoth}(2)$$

```
arcsinh(-x), arccosh(x + 1), arctanh(1/x)
```

$$-\operatorname{arcsinh}(x),\ \operatorname{arccosh}(x+1),\ \operatorname{arctanh}\left(\frac{1}{x}\right)$$

Floating-point values are computed for floating-point arguments:

```
arcsinh(0.1234), arccosh(5.6 + 7.8*I), arccoth(1.0/10^20)
```

$$0.1230889466,\ 2.956002937 + 0.950687977\,i,\ -1.570796327\,i$$

Floating-point intervals are returned for arguments of this type:

```
arccoth(0.5 ... 1.5), arcsinh(0.1234...0.12345)
```

$$0.2554128118 \ldots \mathrm{RD\_INF} + -1.570796327 \ldots - 1.570796326\,i \cup 0.8047189562 \ldots \mathrm{RD\_INF},\ 0.1230889466 \ldots 0.1231385701$$

The inverse of the hyperbolic tangent function has real values only in the interval (- 1, 1):

```
arctanh(-1/2...0), arctanh(2...3)
```

$$-0.5493061444 \ldots 0.0,\ 0.202732554 \ldots 0.6931471806 + -1.570796327 \ldots - 1.570796326\,i$$

Example 2

The inverse hyperbolic functions can be rewritten in terms of the logarithm function:

```
rewrite(arcsinh(x), ln), rewrite(arctanh(x), ln)
```

$$\ln\left(x + \sqrt{x^2+1}\right),\ \frac{\ln(x+1)}{2} - \frac{\ln(1-x)}{2}$$

Example 3

The values jump when crossing a branch cut:

```
arctanh(2.0 + I/10^10), arctanh(2.0 - I/10^10)
```

$$0.5493061443 + 1.570796327\,i,\ 0.5493061443 - 1.570796327\,i$$

On the branch cut, the values of `arctanh` coincide with the limit “from below” for real arguments $x > 1$. The values coincide with the limit “from above” for real $x < -1$:

```
arctanh(1.2), arctanh(1.2 - I/10^10), arctanh(1.2 + I/10^10)
```

$$1.198947636 - 1.570796327\,i,\ 1.198947636 - 1.570796327\,i,\ 1.198947636 + 1.570796327\,i$$

```
arctanh(-1.2), arctanh(-1.2 + I/10^10), arctanh(-1.2 - I/10^10)
```

$$-1.198947636 + 1.570796327\,i,\ -1.198947636 + 1.570796327\,i,\ -1.198947636 - 1.570796327\,i$$

Example 4

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the inverse hyperbolic functions:

```
diff(arcsinh(x^2), x), float(arccosh(3)*arctanh(5 + I))
```

$$\frac{2\,x}{\sqrt{x^4+1}},\ 0.3427241326 + 2.698556745\,i$$

```
limit(arcsinh(x)/arctanh(x), x = 0)
```

$$1$$

```
series(arctanh(sinh(x)) - arcsinh(tanh(x)), x = 0, 10)
```

$$x^3 + \frac{83\,x^7}{120} - \frac{4\,x^9}{189} + \frac{22831\,x^{11}}{28800} + O(x^{13})$$

Parameters

x

Arithmetical expression or floating-point interval

Return Values

Arithmetical expression or floating-point interval

Overloaded By

x

See Also

MuPAD Functions

`arccosh` | `arccsch` | `arcsech` | `arcsinh` | `arctanh` | `cosh` | `coth` | `csch` | `sech` | `sinh` | `tanh`

arg

Argument (polar angle) of a complex number

Syntax

```
arg(z)
```

```
arg(x, y)
```

Description

`arg(z)` returns the argument of the complex number `z`.

`arg(x, y)` returns the argument of the complex number with real part `x` and imaginary part `y`.

This function is also known as `atan2` in other mathematical languages.

The argument of a non-zero complex number $z = x + i\,y = |z|\,e^{i\phi}$ is its real polar angle ϕ. `arg(x,y)` represents the principal value $\phi \in (-\pi, \pi]$. For $x \neq 0$, $y \neq 0$, it is given by

$$\arg(x, y) = \arctan\left(\frac{y}{x}\right) + \frac{\pi}{2}\,\mathrm{sign}(y)\,(1 - \mathrm{sign}(x))$$

An error occurs if `arg` is called with two arguments and either one of the arguments `x`, `y` is a non-real numerical value. Symbolic arguments are assumed to be real.

On the other hand, if `arg` is called with only one argument `x + I*y`, it is not assumed that `x` and `y` are real.

A floating-point number is returned if one argument is given which is a floating-point number; or if two arguments are given, both of them are numerical and at least one of them is a floating-point number.

If the sign of the arguments can be determined, then the result is expressed in terms of `arctan`. Cf. “Example 2” on page 1-237. Otherwise, a symbolic call of `arg` is returned.

Numerical factors are eliminated from the first argument. Cf. “Example 3” on page 1-237.

A symbolic call to `arg` returned has only one argument.

The call `arg(0,0)`, or equivalently `arg(0)`, returns `0`.

An alternative representation is $\arg(x, y) = -i \ln\left(\frac{z}{|z|}\right) = -i \ln(\operatorname{sign}(z))$. Cf. “Example 4” on page 1-238.

Environment Interactions

When called with floating-point arguments, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision. Properties of identifiers are taken into account.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
arg(2, 3), arg(x, 4), arg(4, y), arg(x, y), arg(10, y + PI)
```

$$\arctan\left(\frac{3}{2}\right),\ \arg(x+4\,i),\ \arctan\left(\frac{y}{4}\right),\ \arg(x+y\,i),\ \arctan\left(\frac{\pi}{10}+\frac{y}{10}\right)$$

If `arg` is called with two arguments, the arguments are implicitly assumed to be real, which allows some additional simplifications compared to a call with only one argument:

```
arg(1, y), arg(1 + I*y)
```

$$\arctan(y),\ \arg(1+y\,i)$$

```
arg(x, infinity), arg(-infinity, 3), arg(-infinity, -3)
```

$$\frac{\pi}{2},\ \pi,\ -\pi$$

Floating point values are computed for floating-point arguments:

```
arg(2.0, 3), arg(2, 3.0), arg(10.0^100, 10.0^(-100))
```

$$0.9827937232,\ 0.9827937232,\ 1.0\ 10^{-200}$$

Example 2

`arg` reacts to properties of identifiers set via `assume`:

```
assume(x > 0): assume(y < 0): arg(x, y)
```

$$\arctan\left(\frac{y}{x}\right)$$

```
assume(x < 0): assume(y > 0): arg(x, y)
```

$$\pi + \arctan\left(\frac{y}{x}\right)$$

```
assume(x <> 0): arg(x, 3)
```

$$\arctan\left(\frac{3}{x}\right) - \frac{\pi\ (\mathrm{sign}(x) - 1)}{2}$$

```
unassume(x), unassume(y):
```

Example 3

Certain simplifications may occur in unevaluated calls. In particular, numerical factors are eliminated from the first argument:

```
arg(3*x, 9*y), arg(-12*sqrt(2)*x, 12*y)
```

$$\mathrm{arg}(x + 3\ y\ \mathrm{i}),\ \mathrm{arg}(-\sqrt{2}\ x + y\ \mathrm{i})$$

Example 4

Use `rewrite` to convert symbolic calls of `arg` to the logarithmic representation:

```
rewrite(arg(x, y), ln)
```

$$-\ln\left(\frac{x+y\,\mathrm{i}}{|x+y\,\mathrm{i}|}\right)\mathrm{i}$$

Example 5

System functions such as `float`, `limit`, or `series` handle expressions involving `arg`:

```
limit(arg(x, x^2/(1+x)), x = infinity)
```

$$\frac{\pi}{4}$$

```
series(arg(x, x^2), x = 1, 4, Real)
```

$$\frac{\pi}{4}+\frac{x-1}{2}-\frac{(x-1)^2}{4}+\frac{(x-1)^3}{12}+O((x-1)^4)$$

Parameters

z

arithmetical expression

x, y

arithmetical expressions representing real numbers

Return Values

Arithmetical expression.

Overloaded By

x, z

See Also

MuPAD Functions

arctan | Im | Re | rectform

args

Access procedure parameters

Syntax

```
args()

args(0)

args(i)

args(i .. j)
```

Description

`args(0)` returns the number of parameters of the current procedure.

`args(i)` returns the value of the `i`th parameter of the current procedure.

`args` accesses the actual parameters of a procedure and can only be used in procedures. It is mainly intended for procedures with a variable number of arguments, since otherwise parameters can simply be accessed by their names.

`args()` returns an expression sequence of all actual parameters.

`args(i..j)` returns an expression sequence containing the `i`th up to the `j`th parameter.

In procedures with option `hold`, `args` returns the parameters without further evaluation. Use `context` or `eval` to enforce a subsequent evaluation. See “Example 2” on page 1-242.

`procname ( args())` returns a symbolic function call of the current procedure with evaluated arguments.

Assigning values to formal parameters of a procedure changes the result of `args`. Cf. “Example 4” on page 1-242. `args(0)` remains unchanged.

Examples

Example 1

This example demonstrates the various ways of using `args`:

```
f := proc() begin
  print(Unquoted, "number of arguments" = args(0)):
  print(Unquoted, "sequence of all arguments" = args()):
  if args(0) > 0 then
    print(Unquoted, "first argument" = args(1)):
  end_if:
  if args(0) >= 3 then
    print(Unquoted, "second, third argument" = args(2..3)):
  end_if:
end_proc:

f():

number of arguments = 0

sequence of all arguments =

f(42):

number of arguments = 1

sequence of all arguments = 42

first argument = 42

f(a, b, c, d):

number of arguments = 4

sequence of all arguments = (a, b, c, d)
```

```
first argument = a


second, third argument = (b, c)
```

Example 2

`args` does not evaluate the returned parameters in procedures with the option `hold`. Use `context` to achieve this:

```
f := proc()
  option hold;
begin
  args(1), context(args(1))
end_proc:

delete x, y: x := y: y := 2: f(x)
```

$$x, 2$$

Example 3

We use `args` to access parameters of a procedure with an arbitrary number of arguments:

```
f := proc() begin
  args(1) * _plus(args(2..args(0)))
end_proc:
f(2, 3), f(2, 3, 4)
```

$$6, 14$$

Example 4

Assigning values to formal parameters affects the behavior of `args`. In the following example, `args` returns the value `4`, which is assigned inside the procedure, and not the value `1`, which is the argument of the procedure call:

```
f := proc(a) begin a := 4; args() end_proc:
f(1)
```

```
4
```

Parameters

`i, j`

Positive integers

Return Values

`args(0)` returns a nonnegative integer. All other calls return an arbitrary MuPAD object or a `sequence` of such objects.

See Also

MuPAD Domains

`DOM_PROC` | `DOM_VAR`

MuPAD Functions

`context` | `Pref::typeCheck` | `proc` | `procname` | `testargs`

array

Create an array

Syntax

```
array(m1 .. n1, <m2 .. n2, …>)

array(m1 .. n1, <m2 .. n2, …>, index1 = entry1, index2 = entry2, …)

array(m1 .. n1, <m2 .. n2, …>, List)

array(<m1 .. n1, m2 .. n2, …>, ListOfLists)
```

Description

`array(...)` creates an *array,* which is an n-dimensional rectangular structure holding arbitrary data.

`array( m_1..n_1, m_2..n_2 , ...)` creates an array with uninitialized entries, where the first index runs from m_1 to n_1, the second index runs from m_2 to n_2, etc.

`array( m_1..n_1, m_2..n_2 , ..., List)` creates an array with entries initialized from `List`.

`array(ListOfLists)` creates an array with entries initialized from `ListOfLists`. The dimension of the array is the same as of `ListOfLists`.

Arrays are container objects for storing data. In contrast to `tables`, the indices must be sequences of integers. While `tables` may grow in size dynamically, the number of entries in an array created by array is fixed.

Arrays created via `array` are of domain type `DOM_ARRAY`. They may contain arbitrary MuPAD objects as entries.

For an array `A`, say, of type `DOM_ARRAY` or `DOM_HFARRAY` and a sequence of integers `index` forming a valid array index, an indexed call `A[index]` returns the corresponding entry. If the entry of an array of type `DOM_ARRAY` is uninitialized, then the indexed

expression `A[index]` is returned. See“Example 1” on page 1-247 and “Example 5” on page 1-250.

An indexed assignment of the form `A[index]:=entry` initializes or overwrites the entry corresponding to `index`. See “Example 1” on page 1-247 and “Example 5” on page 1-250.

The index boundaries must satisfy $m_1 \le n_1$, $m_2 \le n_2$, etc. The dimension of the resulting array is the number of given range arguments; at least one range argument must be specified. The total number of entries of the resulting array is $(n_1 - m_1 + 1)\,(n_2 - m_2 + 1)\,\ldots$.

If only index range arguments are given to `array`, then an array with uninitialized entries is created. Hardware float arrays created via `hfarray` cannot have uninitialized entries. Entries are automatically set to 0.0 if no values are specified. Cf. “Example 1” on page 1-247.

If equations of the form `index=entry` are present, then the array entry corresponding to `index` is initialized with `entry`. This is useful for selectively initializing some particular array entries.

Each index must be a valid array index of the form i_1 for 1-dimensional arrays and $(i_1, i_2, \ldots)$ for higher-dimensional arrays, where $i_1, i_2, \ldots$ are integers within the valid boundaries, satisfying $m_1 \le i_1 \le n_1$, $m_2 \le i_2 \le n_2$, etc., and the number of integers in `index` matches the dimension of the array.

If the argument `List` is present, then the resulting array is initialized with the entries from `List`. This is useful for initializing all array entries at once. The list must have $(n_1 - m_1 + 1)\,(n_2 - m_2 + 1)\,\ldots$ elements, each becoming an operand of the array to be created. In case of 2-dimensional arrays, regarded as a matrix, the list contains the entries row after row.

The argument `ListOfLists` must be a nested list matching the structure of the array exactly. The nesting depth of the list must be greater or equal to the dimension of the array. The number of list entries at the k-th nesting level must be equal to the size of the k-th index range, i.e., $n_k - m_k + 1$. Cf. “Example 7” on page 1-253.

A call of the form `delete A[index]` deletes the entry corresponding to `index`, so that it becomes uninitialized. For arrays of domain type `DOM_HFARRAY` this means that the corresponding entry is set to 0.0. Cf. “Example 5” on page 1-250.

Note: Internally, uninitialized entries of an array of domain type DOM_ARRAY have the value NIL. Thus assigning NIL to an array entry has the same effect as deleting it via delete. Afterwards, an indexed call of the form A[index] returns the symbolic expression A[index], and not NIL, as one might expect. Cf. "Example 5" on page 1-250.

A 1-dimensional array is printed as a row vector. The index corresponds to the column number.

A 2-dimensional array is printed as a matrix. The first index corresponds to the row number and the second index corresponds to the column number.

A 1- or 2-dimensional array that is so big that it would exceed the maximal output width TEXTWIDTH is printed in the form `array( m_1..n_1, m_2..n_2, dots, index_1 = entry_1, index_2 = entry_2, dots )`. See "Example 10" on page 1-256. The same is true for arrays of dimension greater than two. See "Example 6" on page 1-252 and "Example 7" on page 1-253.

Arithmetic operations are not defined for arrays of domain type DOM_ARRAY. Use matrix to create 1-dimensional vectors or 2-dimensional matrices in the mathematical sense.

Arithmetic operations are defined for arrays of domain type DOM_HFARRAY!

E.g., linear combination of arrays A, B can be computed via a*A + b*B if A, B have the same format and if the scalar factors a, b are numbers (floats, integers or rationals).

2-dimensional hfarrays A, B are processed like matrices: Operations such as A*B (matrix multiplication), A^n (matrix powers), or 1/A (matrix inversion) are possible wherever this is meaningful mathematically.

Cf. "Example 8" on page 1-254.

Note the following special feature of arrays of domain type DOM_ARRAY:

Note: If an array is evaluated, it is only returned. The evaluation does not map recursively on the array entries! This is due to performance reasons. You have to map the function eval explicitly on the array in order to fully evaluate its entries.

Cf. "Example 9" on page 1-255.

Examples

Example 1

We create an uninitialized 1-dimensional array with indices ranging from 2 to 4:

```
A := array(2..4)
```

$$(\text{NIL}\ \text{NIL}\ \text{NIL})$$

The `NIL`s in the output indicate that the array entries are not initialized. We set the middle entry to 5 and last entry to `"MuPAD"`:

```
A[3] := 5: A[4] := "MuPAD": A
```

$$(\text{NIL}\ 5\ \text{"MuPAD"})$$

You can access array entries via indexed calls. Since the entry `A[2]` is not initialized, the symbolic expression `A[2]` is returned:

```
A[2], A[3], A[4]
```

$$A_2,\ 5,\ \text{"MuPAD"}$$

We can initialize an array already when creating it by passing initialization equations to `array`:

```
A := array(2..4, 3 = 5, 4 = "MuPAD")
```

$$(\text{NIL}\ 5\ \text{"MuPAD"})$$

We can initialize all entries of an array when creating it by passing a list of initial values to `array`:

```
array(2..4, [PI, 5, "MuPAD"])
```

$$(\pi\ 5\ \text{"MuPAD"})$$

Hardware float arrays do not have uninitialized entries. If no initialization value is given, the corrsponding entry is set to 0.0:

```
hfarray(-1..5)
```

(0.0 0.0 0.0 0.0 0.0 0.0 0.0)

```
hfarray(-1..5, 2 = PI, 4 = sqrt(2)*exp(2))
```

(0.0 0.0 0.0 3.141592654 0.0 10.44970335 0.0)

```
hfarray(-1..5, [frandom() $ i = -1..5])
```

[[0.2703581656, 0.8310371787, 0.153156516, 0.9948127808, 0.2662729021, 0.1801642277, 0.452083055]]

```
hfarray(1..2, 1..3, [[1, 2, 3], [4, 5, 6]])
```

$$\begin{pmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \end{pmatrix}$$

```
hfarray(1..2, 1..3, [1, 2, 3, 4, 5, 6])
```

$$\begin{pmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \end{pmatrix}$$

Example 2

Array boundaries may be negative integers as well:

```
A := array(-1..1, [2, sin(x), FAIL])
```

(2 $\sin(x)$ FAIL)

```
A[-1], A[0], A[1]
```

2, $\sin(x)$, FAIL

```
A := hfarray(-1..2, -3..-1, [[-1, -1, -1],
                             [ 0,  0,  0],
                             [ 1,  1,  1],
                             [ 2,  2,  2]])
```

$$\begin{pmatrix} -1.0 & -1.0 & -1.0 \\ 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 1.0 \\ 2.0 & 2.0 & 2.0 \end{pmatrix}$$

```
A[-1, -2], A[0, -3], A[2, -3]
```

$$-1.0,\ 0.0,\ 2.0$$

```
delete A:
```

Example 3

If the dimension and size of the `array` or `hfarray` are not specified explicitly then both values are taken from the given list:

```
hfarray([1.0,2.0,3.0,4.0,5.0]) = hfarray(1..5, [1.0,2.0,3.0,4.0,5.0]);
bool(%)
```

$$(\ 1.0\ \ 2.0\ \ 3.0\ \ 4.0\ \ 5.0\ ) = (\ 1.0\ \ 2.0\ \ 3.0\ \ 4.0\ \ 5.0\ )$$

TRUE

```
array([[1,2],[3,4],[5,6]]) = array(1..3, 1..2, [[1,2],[3,4],[5,6]]);
bool(%)
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

TRUE

Note that all subfields of one dimension must have the same size and dimension. Therefore, the following input leads to an error:

```
array([[1],[3,4],[5,6]])
```

```
Error: The argument is invalid. [array]
```

Example 4

The $ operator may be used to create a sequence of initialization equations:

```
array(1..8, i = i^2 $ i = 1..8)
```

$$(1\ 4\ 9\ 16\ 25\ 36\ 49\ 64)$$

```
hfarray(1..4, 1..4, (i, i) = 1 $ i = 1..4)
```

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$$

Equivalently, you can use the $ operator to create an initialization list:

```
array(1..8, [i^2 $ i = 1..8])
```

$$(1\ 4\ 9\ 16\ 25\ 36\ 49\ 64)$$

```
hfarray(1..8, [i*PI $ i = 1..8])
```

$$[[3.141592654, 6.283185307, 9.424777961, 12.56637061, 15.70796327, 18.84955592, 21.99114858, 25.13274123]]$$

Example 5

We create a 2×2 matrix as a 2-dimensional array:

```
A := array(1..2, 1..2, (1, 2) = 42, (2, 1) = 1 + I)
```

$$\begin{pmatrix} \text{NIL} & 42 \\ 1+i & \text{NIL} \end{pmatrix}$$

Internally, array entries are stored in a linearized form. They can be accessed in this form via `op`. Uninitialized entries internally have the value `NIL`:

```
op(A, 1), op(A, 2), op(A, 3), op(A, 4)
```

$$\text{NIL}, 42, 1+i, \text{NIL}$$

Note the difference to the indexed access:

```
A[1, 1], A[1, 2], A[2, 1], A[2, 2]
```

$$A_{1,1}, 42, 1+i, A_{2,2}$$

We can modify an array entry by an indexed assignment:

```
A[1, 1] := 0: A[1, 2] := 5:
A
```

$$\begin{pmatrix} 0 & 5 \\ 1+i & \text{NIL} \end{pmatrix}$$

You can delete the value of an array entry via `delete`. Afterwards, it is uninitialized again:

```
delete A[2, 1]: A[2, 1], op(A, 3)
```

$$A_{2,1}, \text{NIL}$$

Assigning `NIL` to an array entry has the same effect as deleting it:

```
A[1, 2] := NIL: A[1, 2], op(A, 2)
```

$$A_{1,2}, \text{NIL}$$

Apart from initialization and deleting entries via NIL assignments, hfarrays behave similarly:

```
A := hfarray(1..2, 1..2, (1, 1) = 1.0, (2, 2) = 1.0)
```

$$\begin{pmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{pmatrix}$$

```
op(A, 1), op(A, 2), op(A, 3), op(A, 4)
```

1.0, 0.0, 0.0, 1.0

```
A[1, 1], A[1, 2], A[2, 1], A[2, 2]
```

1.0, 0.0, 0.0, 1.0

```
A[2, 2] := PI:
A[2, 2]
```

3.141592654

```
delete A[2, 2]:
```

```
Error: The argument is invalid. [delete]
```

```
A
```

$$\begin{pmatrix} 1.0 & 0.0 \\ 0.0 & 3.141592654 \end{pmatrix}$$

```
delete A:
```

Example 6

We define a three-dimensional array with index values between 1 and 8 in each of the three dimensions and initialize two of the entries via initialization equations:

```
A := array(1..8, 1..8, 1..8, (1, 1, 1) = 111, (8, 8, 8) = 888)
```

```
array(1..8, 1..8, 1..8,
  (1, 1, 1) =  111
  (8, 8, 8) =  888
)
```

```
A[1, 1, 1], A[1, 1, 2]
```

$$111, A_{1,1,2}$$

We create a 3-dimensional hfarray:

```
A := hfarray(1..2, 2..3, 3..4, (1, 2, 3) = 123, (2, 3, 4) = 234)
```

```
hfarray(1..2, 2..3, 3..4, [123.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 234.0])
```

```
delete A, B:
```

Example 7

A nested list may be used to initialize a 2-dimensional array. The inner lists are the rows of the created matrix:

```
array(1..2, 1..3, [[1, 2, 3], [4, 5, 6]])
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

We create a three-dimensional array and initialize it from a nested list of depth three. The outer list has two entries for the first dimension. Each of these entries is a list with three entries for the second dimension. Finally, the innermost lists each have one entry for the third dimension:

```
array(2..3, 1..3, 1..1,
      [
        [ [1], [2], [3] ],
        [ [4], [5], [6] ]
      ])
```

```
array(2..3, 1..3, 1..1,
  (2, 1, 1) =  1
  (2, 2, 1) =  2
  (2, 3, 1) =  3
  (3, 1, 1) =  4
  (3, 2, 1) =  5
  (3, 3, 1) =  6
)
```

```
hfarray(2..3, 1..3, 1..1,
        [
          [ [1], [2], [3] ],
          [ [4], [5], [6] ]
        ])
```

```
hfarray(2..3, 1..3, 1..1, [1.0, 2.0, 3.0, 4.0, 5.0, 6.0])
```

Example 8

Basic arithmetic is available for arrays of domain type DOM_HFARRAY:

```
A := hfarray(1..5, [1, 2, 3, 4, 5]):
B := hfarray(1..5, [5, 4, 3, 2, 1]):
A + B
```

```
( 6.0 6.0 6.0 6.0 6.0 )
```

```
2*A
```

```
( 2.0 4.0 6.0 8.0 10.0 )
```

```
2*A - 3* B
```

```
( -13.0 -8.0 -3.0 2.0 7.0 )
```

2-dimensional arrays of type DOM_ARRAY are regarded as matrices. They can be multiplied accordingly:

```
A := hfarray(1..3, 1..3, [frandom() $ i = 1..9]):
B := hfarray(1..3, 1..2, [frandom() $ i = 1..6]):
A, B, A * B, A^10 * B
```

$$\begin{pmatrix} 0.6787819563 & 0.3549849261 & 0.6818588132 \\ 0.7219186551 & 0.4738297742 & 0.7889814922 \\ 0.2115258358 & 0.8556871754 & 0.04489739417 \end{pmatrix}, \begin{pmatrix} 0.8791601269 & 0.9193848479 \\ 0.7350574234 & 0.7875450269 \\ 0.9371484273 & 0.2953238727 \end{pmatrix},$$

$$\begin{pmatrix} 1.49669525 & 1.104997644 \\ 1.722366954 & 1.269888425 \\ 0.8570198134 & 0.8816251003 \end{pmatrix}, \begin{pmatrix} 104.6209923 & 83.25909538 \\ 121.6963323 & 96.84792749 \\ 80.3865413 & 63.97259571 \end{pmatrix}$$

The following command computes the matrix inverse of A:

```
1/A
```

$$\begin{pmatrix} 15.13041048 & -13.13273828 & 0.9952407788 \\ -3.111891486 & 2.632361162 & 1.001982382 \\ -11.97546309 & 11.70303455 & -1.512395388 \end{pmatrix}$$

Some functions such as `norm` act on hfarrays:

```
norm(A)
```

$$1.984729921$$

Example 9

If an array is evaluated, it is only returned. The evaluation does not map recursively on the array entries. Here, the entries a and b are not evaluated:

```
A := array(1..2, [a, b]):
a := 1:  b := 2:
A, eval(A)
```

$$(a\ b), (a\ b)$$

Due to the special evaluation of arrays the index operator evaluates array entries after extracting them from the array:

```
A[1], A[2]
```

$$1, 2$$

You have to `map` the function `eval` explicitly on the array in order to fully evaluate its entries:

```
map(A, eval)
```

$$(1\ 2)$$

Example 10

A 2-dimensional array is usually printed in matrix form:

```
A := array(1..4, 1..4, (1, 1) = 11, (4, 4) = 44)
```

$$\begin{pmatrix} 11 & \text{NIL} & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 44 \end{pmatrix}$$

```
B := hfarray(1..2, 1..3, (1, 1) = 11, (2, 3) = 23)
```

$$\begin{pmatrix} 11.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 23.0 \end{pmatrix}$$

If the output does not fit into `TEXTWIDTH`, a more compact output is used in `print`:

```
PRETTYPRINT := FALSE:
TEXTWIDTH := 20:
print(Plain, A)
```

```
array(1..4, 1..4, (\
1, 1) = 11, (4, 4) \
= 44)
```

```
PRETTYPRINT := TRUE:
```

```
delete A, B, TEXTWIDTH:
```

Parameters

m_1, n_1, m_2, n_2, …

The index boundaries: integers

$index_1$, $index_2$, …

A sequence of integers defining a valid array index

$entry_1$, $entry_2$, …

Arbitrary objects

List

A plain list of entries for initializing the array

ListOfLists

A nested list (of lists of lists of …) of entries for initializing the array

Return Values

Object of type DOM_ARRAY or DOM_HFARRAY, respectively.

See Also

MuPAD Domains

DOM_ARRAY | DOM_HFARRAY | DOM_LIST | DOM_TABLE

MuPAD Functions

_assign | _index | assignElements | delete | hfarray | indexval | matrix | table

hfarray

Create an array of hardware floats

Syntax

```
hfarray(m1 .. n1, <m2 .. n2, …>)
hfarray(m1 .. n1, <m2 .. n2, …>, index1 = number1, index2 = number2, …)
hfarray(m1 .. n1, <m2 .. n2, …>, List)
hfarray(<m1 .. n1, m2 .. n2, …>, ListOfLists)
```

Description

`hfarray(...)` creates an array specialized to hold *hardware floating-point values.* These *do not* react to `DIGITS`, and no symbolic expressions can be placed into an `hfarray`. The values can be real or complex.

`hfarray( m_1..n_1, m_2..n_2 , ...)` creates an array of floating-point zeroes, where the first index runs from m_1 to n_1, the second index runs from m_2 to n_2, etc.

`hfarray( m_1..n_1, m_2..n_2 , ..., List)` creates an array of floating-point numbers with entries initialized from `List`.

`hfarray(ListOfLists)` creates an array of floating-point numbers with entries initialized from `ListOfLists`. The dimension of the `hfarray` is the same as of `ListOfLists`.

Arrays are container objects for storing data. In contrast to `tables`, the indices must be sequences of integers. While `tables` may grow in size dynamically, the number of entries in an array created by hfarray is fixed.

Arrays created via `hfarray` are of domain type `DOM_HFARRAY`. They can only contain floating-point numbers as entries. Internally, these floating-point numbers are stored as "hardware floats" with about 15 significant decimal digits ("double precision"). This data type serves for storing *large* amounts of numerical data. E.g., an array with 15 digits software floats created via `array` (using `DIGITS` = 15) takes nearly 10 times as much storage space as the corresponding hfarray.

On input, the entries passed to `hfarray` may be MuPAD floating-point numbers, integers or rational numbers, or exact numerical expressions such as `PI + sin(sqrt(2))` that can be converted to floating-point numbers. Exact input data are automatically converted to hardware floats of double precision. This conversion does not depend on the present value of `DIGITS`!

Note: Entries of absolute value smaller than about 10^{-308} are stored as 0.0 by `hfarray`!

An error is thrown if symbolic objects are passed to `hfarray`.

For an array `A`, say, of type `DOM_ARRAY` or `DOM_HFARRAY` and a sequence of integers `index` forming a valid array index, an indexed call `A[index]` returns the corresponding entry. If the entry of an array of type `DOM_ARRAY` is uninitialized, then the indexed expression `A[index]` is returned. See "Example 1" on page 1-261 and "Example 5" on page 1-265.

An indexed assignment of the form `A[index]:=entry` initializes or overwrites the entry corresponding to `index`. See "Example 1" on page 1-261 and "Example 5" on page 1-265.

The index boundaries must satisfy $m_1 \le n_1$, $m_2 \le n_2$, etc. The dimension of the resulting array is the number of given range arguments; at least one range argument must be specified. The total number of entries of the resulting array is $(n_1 - m_1 + 1)\,(n_2 - m_2 + 1)\,\ldots$.

If only index range arguments are given to `array`, then an array with uninitialized entries is created. Hardware float arrays created via `hfarray` cannot have uninitialized entries. Entries are automatically set to 0.0 if no values are specified. Cf. "Example 1" on page 1-261.

If equations of the form `index=entry` are present, then the array entry corresponding to `index` is initialized with `entry`. This is useful for selectively initializing some particular array entries.

Each index must be a valid array index of the form i_1 for 1-dimensional arrays and $(i_1, i_2, \ldots)$ for higher-dimensional arrays, where $i_1, i_2, \ldots$ are integers within the valid boundaries, satisfying $m_1 \le i_1 \le n_1$, $m_2 \le i_2 \le n_2$, etc., and the number of integers in `index` matches the dimension of the array.

If the argument `List` is present, then the resulting array is initialized with the entries from `List`. This is useful for initializing all array entries at once. The list must have (n_1 - m_1 + 1) (n_2 - m_2 + 1) … elements, each becoming an operand of the array to be created. In case of 2-dimensional arrays, regarded as a matrix, the list contains the entries row after row.

The argument `ListOfLists` must be a nested list matching the structure of the array exactly. The nesting depth of the list must be greater or equal to the dimension of the array. The number of list entries at the k-th nesting level must be equal to the size of the k-th index range, i.e., n_k - m_k + 1. Cf. “Example 7” on page 1-267.

A call of the form `delete A[index]` deletes the entry corresponding to `index`, so that it becomes uninitialized. For arrays of domain type `DOM_HFARRAY` this means that the corresponding entry is set to 0.0. Cf. “Example 5” on page 1-265.

Note: Internally, uninitialized entries of an array of domain type `DOM_ARRAY` have the value `NIL`. Thus assigning `NIL` to an array entry has the same effect as deleting it via `delete`. Afterwards, an indexed call of the form `A[index]` returns the symbolic expression `A[index]`, and not `NIL`, as one might expect. Cf. “Example 5” on page 1-265.

A 1-dimensional array is printed as a row vector. The index corresponds to the column number.

A 2-dimensional array is printed as a matrix. The first index corresponds to the row number and the second index corresponds to the column number.

Arrays of dimension greater than two are printed in the form `hfarray( m_1..n_1, m_2..n_2, dots, index_1 = entry_1, index_2 = entry_2, dots )`. See “Example 6” on page 1-267 and “Example 7” on page 1-267.

Arithmetic operations are not defined for arrays of domain type `DOM_ARRAY`. Use `matrix` to create 1-dimensional vectors or 2-dimensional matrices in the mathematical sense.

Arithmetic operations are defined for arrays of domain type `DOM_HFARRAY`!

E.g., linear combination of arrays `A, B` can be computed via `a*A + b*B` if `A, B` have the same format and if the scalar factors `a, b` are numbers (floats, integers or rationals).

2-dimensional hfarrays `A, B` are processed like matrices: Operations such as `A*B` (matrix multiplication), `A^n` (matrix powers), or `1/A` (matrix inversion) are possible wherever this is meaningful mathematically.

Cf. “Example 8” on page 1-268.

Note the following special feature of arrays of domain type `DOM_ARRAY`:

Note: If an array is evaluated, it is only returned. The evaluation does not map recursively on the array entries! This is due to performance reasons. You have to `map` the function `eval` explicitly on the array in order to fully evaluate its entries.

Cf. “Example 9” on page 1-270.

Examples

Example 1

We create an uninitialized 1-dimensional array with indices ranging from 2 to 4:

```
A := array(2..4)
```

$$(\text{NIL}\ \ \text{NIL}\ \ \text{NIL})$$

The `NIL`s in the output indicate that the array entries are not initialized. We set the middle entry to 5 and last entry to `"MuPAD"`:

```
A[3] := 5: A[4] := "MuPAD": A
```

$$(\text{NIL}\ \ 5\ \ \text{"MuPAD"})$$

You can access array entries via indexed calls. Since the entry `A[2]` is not initialized, the symbolic expression `A[2]` is returned:

```
A[2], A[3], A[4]
```

$$A_2,\ 5,\ \text{"MuPAD"}$$

We can initialize an array already when creating it by passing initialization equations to `array`:

```
A := array(2..4, 3 = 5, 4 = "MuPAD")
```

$$(\text{NIL}\ \ 5\ \ \text{"MuPAD"})$$

We can initialize all entries of an array when creating it by passing a list of initial values to `array`:

```
array(2..4, [PI, 5, "MuPAD"])
```

$$(\pi\ \ 5\ \ \text{"MuPAD"})$$

Hardware float arrays do not have uninitialized entries. If no initialization value is given, the corrsponding entry is set to 0.0:

```
hfarray(-1..5)
```

$$(0.0\ \ 0.0\ \ 0.0\ \ 0.0\ \ 0.0\ \ 0.0\ \ 0.0)$$

```
hfarray(-1..5, 2 = PI, 4 = sqrt(2)*exp(2))
```

$$(0.0\ \ 0.0\ \ 0.0\ \ 3.141592654\ \ 0.0\ \ 10.44970335\ \ 0.0)$$

```
hfarray(-1..5, [frandom() $ i = -1..5])
```

$$[[0.2703581656,\ 0.8310371787,\ 0.153156516,\ 0.9948127808,\ 0.2662729021,\ 0.1801642277,\ 0.452083055]]$$

```
hfarray(1..2, 1..3, [[1, 2, 3], [4, 5, 6]])
```

$$\begin{pmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \end{pmatrix}$$

```
hfarray(1..2, 1..3, [1, 2, 3, 4, 5, 6])
```

$$\begin{pmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \end{pmatrix}$$

Example 2

Array boundaries may be negative integers as well:

```
A := array(-1..1, [2, sin(x), FAIL])
```

$$\begin{pmatrix} 2 & \sin(x) & \text{FAIL} \end{pmatrix}$$

```
A[-1], A[0], A[1]
```

$$2,\ \sin(x),\ \text{FAIL}$$

```
A := hfarray(-1..2, -3..-1, [[-1, -1, -1],
                             [ 0,  0,  0],
                             [ 1,  1,  1],
                             [ 2,  2,  2]])
```

$$\begin{pmatrix} -1.0 & -1.0 & -1.0 \\ 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 1.0 \\ 2.0 & 2.0 & 2.0 \end{pmatrix}$$

```
A[-1, -2], A[0, -3], A[2, -3]
```

$$-1.0,\ 0.0,\ 2.0$$

```
delete A:
```

Example 3

If the dimension and size of the `array` or `hfarray` are not specified explicitly then both values are taken from the given list:

```
hfarray([1.0,2.0,3.0,4.0,5.0]) = hfarray(1..5, [1.0,2.0,3.0,4.0,5.0]);
bool(%)
```

$$(1.0\ 2.0\ 3.0\ 4.0\ 5.0) = (1.0\ 2.0\ 3.0\ 4.0\ 5.0)$$

TRUE

```
array([[1,2],[3,4],[5,6]]) = array(1..3, 1..2, [[1,2],[3,4],[5,6]]);
bool(%)
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

TRUE

Note that all subfields of one dimension must have the same size and dimension. Therefore, the following input leads to an error:

```
array([[1],[3,4],[5,6]])
```

```
Error: The argument is invalid. [array]
```

Example 4

The $ operator may be used to create a sequence of initialization equations:

```
array(1..8, i = i^2 $ i = 1..8)
```

$$(1\ 4\ 9\ 16\ 25\ 36\ 49\ 64)$$

```
hfarray(1..4, 1..4, (i, i) = 1 $ i = 1..4)
```

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$$

Equivalently, you can use the $ operator to create an initialization list:

```
array(1..8, [i^2 $ i = 1..8])
```

```
( 1  4  9  16  25  36  49  64 )
```

```
hfarray(1..8, [i*PI $ i = 1..8])
```

```
[[3.141592654, 6.283185307, 9.424777961, 12.56637061, 15.70796327, 18.84955592,
  21.99114858, 25.13274123]]
```

Example 5

We create a 2×2 matrix as a 2-dimensional array:

```
A := array(1..2, 1..2, (1, 2) = 42, (2, 1) = 1 + I)
```

$$\begin{pmatrix} \text{NIL} & 42 \\ 1+i & \text{NIL} \end{pmatrix}$$

Internally, array entries are stored in a linearized form. They can be accessed in this form via `op`. Uninitialized entries internally have the value `NIL`:

```
op(A, 1), op(A, 2), op(A, 3), op(A, 4)
```

$$\text{NIL},\ 42,\ 1+i,\ \text{NIL}$$

Note the difference to the indexed access:

```
A[1, 1], A[1, 2], A[2, 1], A[2, 2]
```

$$A_{1,1},\ 42,\ 1+i,\ A_{2,2}$$

We can modify an array entry by an indexed assignment:

```
A[1, 1] := 0: A[1, 2] := 5:
A
```

$$\begin{pmatrix} 0 & 5 \\ 1+i & \text{NIL} \end{pmatrix}$$

You can delete the value of an array entry via `delete`. Afterwards, it is uninitialized again:

```
delete A[2, 1]: A[2, 1], op(A, 3)
```

$$A_{2,1}, \text{NIL}$$

Assigning `NIL` to an array entry has the same effect as deleting it:

```
A[1, 2] := NIL: A[1, 2], op(A, 2)
```

$$A_{1,2}, \text{NIL}$$

Apart from initialization and deleting entries via `NIL` assignments, hfarrays behave similarly:

```
A := hfarray(1..2, 1..2, (1, 1) = 1.0, (2, 2) = 1.0)
```

$$\begin{pmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{pmatrix}$$

```
op(A, 1), op(A, 2), op(A, 3), op(A, 4)
```

1.0, 0.0, 0.0, 1.0

```
A[1, 1], A[1, 2], A[2, 1], A[2, 2]
```

1.0, 0.0, 0.0, 1.0

```
A[2, 2] := PI:
A[2, 2]
```

3.141592654

```
delete A[2, 2]:
```

```
Error: The argument is invalid. [delete]
```

```
A
```

$$\begin{pmatrix} 1.0 & 0.0 \\ 0.0 & 3.141592654 \end{pmatrix}$$

```
delete A:
```

Example 6

We define a three-dimensional array with index values between 1 and 8 in each of the three dimensions and initialize two of the entries via initialization equations:

```
A := array(1..8, 1..8, 1..8, (1, 1, 1) = 111, (8, 8, 8) = 888)
```

$$\begin{array}{l} \text{array}(1..8,\ 1..8,\ 1..8, \\ \quad (1,\ 1,\ 1) = \ 111 \\ \quad (8,\ 8,\ 8) = \ 888 \\ ) \end{array}$$

```
A[1, 1, 1], A[1, 1, 2]
```

$$111,\ A_{1,1,2}$$

We create a 3-dimensional hfarray:

```
A := hfarray(1..2, 2..3, 3..4, (1, 2, 3) = 123, (2, 3, 4) = 234)
```

```
hfarray(1..2, 2..3, 3..4, [123.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 234.0])
```

```
delete A, B:
```

Example 7

A nested list may be used to initialize a 2-dimensional array. The inner lists are the rows of the created matrix:

```
array(1..2, 1..3, [[1, 2, 3], [4, 5, 6]])
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

We create a three-dimensional array and initialize it from a nested list of depth three. The outer list has two entries for the first dimension. Each of these entries is a list with three entries for the second dimension. Finally, the innermost lists each have one entry for the third dimension:

```
array(2..3, 1..3, 1..1,
      [
        [ [1], [2], [3] ],
        [ [4], [5], [6] ]
      ])
```

```
array(2..3, 1..3, 1..1,
  (2, 1, 1) =  1
  (2, 2, 1) =  2
  (2, 3, 1) =  3
  (3, 1, 1) =  4
  (3, 2, 1) =  5
  (3, 3, 1) =  6
)
```

```
hfarray(2..3, 1..3, 1..1,
        [
          [ [1], [2], [3] ],
          [ [4], [5], [6] ]
        ])
```

```
hfarray(2..3, 1..3, 1..1, [1.0, 2.0, 3.0, 4.0, 5.0, 6.0])
```

Example 8

Basic arithmetic is available for arrays of domain type DOM_HFARRAY:

```
A := hfarray(1..5, [1, 2, 3, 4, 5]):
B := hfarray(1..5, [5, 4, 3, 2, 1]):
A + B
```

$$(\,6.0\ \ 6.0\ \ 6.0\ \ 6.0\ \ 6.0\,)$$

```
2*A
```

$$(\,2.0\ \ 4.0\ \ 6.0\ \ 8.0\ \ 10.0\,)$$

```
2*A - 3* B
```

$$(\,-13.0\ \ -8.0\ \ -3.0\ \ 2.0\ \ 7.0\,)$$

2-dimensional arrays of type `DOM_ARRAY` are regarded as matrices. They can be multiplied accordingly:

```
A := hfarray(1..3, 1..3, [frandom() $ i = 1..9]):
B := hfarray(1..3, 1..2, [frandom() $ i = 1..6]):
A, B, A * B, A^10 * B
```

$$\begin{pmatrix} 0.6787819563 & 0.3549849261 & 0.6818588132 \\ 0.7219186551 & 0.4738297742 & 0.7889814922 \\ 0.2115258358 & 0.8556871754 & 0.04489739417 \end{pmatrix}, \begin{pmatrix} 0.8791601269 & 0.9193848479 \\ 0.7350574234 & 0.7875450269 \\ 0.9371484273 & 0.2953238727 \end{pmatrix},$$

$$\begin{pmatrix} 1.49669525 & 1.104997644 \\ 1.722366954 & 1.269888425 \\ 0.8570198134 & 0.8816251003 \end{pmatrix}, \begin{pmatrix} 104.6209923 & 83.25909538 \\ 121.6963323 & 96.84792749 \\ 80.3865413 & 63.97259571 \end{pmatrix}$$

The following command computes the matrix inverse of `A`:

```
1/A
```

$$\begin{pmatrix} 15.13041048 & -13.13273828 & 0.9952407788 \\ -3.111891486 & 2.632361162 & 1.001982382 \\ -11.97546309 & 11.70303455 & -1.512395388 \end{pmatrix}$$

Some functions such as `norm` act on hfarrays:

```
norm(A)
```

$$1.984729921$$

Example 9

If an array is evaluated, it is only returned. The evaluation does not map recursively on the array entries. Here, the entries `a` and `b` are not evaluated:

```
A := array(1..2, [a, b]):
a := 1:  b := 2:
A, eval(A)
```

$$(a\ b), (a\ b)$$

Due to the special evaluation of arrays the index operator evaluates array entries after extracting them from the array:

```
A[1], A[2]
```

$$1, 2$$

You have to `map` the function `eval` explicitly on the array in order to fully evaluate its entries:

```
map(A, eval)
```

$$(1\ 2)$$

Parameters

m_1, n_1, m_2, n_2, …

The index boundaries: integers

$index_1$, $index_2$, …

A sequence of integers defining a valid array index

$number_1$, $number_2$, …

Real or complex floating-point numbers or numerical expressions that can be converted to real or complex floating-point numbers

`List`

A plain list of entries for initializing the array

`ListOfLists`

A nested list (of lists of lists of …) of entries for initializing the array

Return Values

Object of type DOM_ARRAY or DOM_HFARRAY, respectively.

See Also

MuPAD Domains
DOM_ARRAY | DOM_HFARRAY | DOM_LIST | DOM_TABLE

MuPAD Functions
_assign | _index | array | assignElements | delete | indexval | matrix | table

assert

Assertions for debugging

Syntax

```
assert(cond)
```

Description

The statement `assert(cond)` declares that the condition `cond` holds true at the moment when the statement is evaluated. By default, MuPAD does not care about assertions. After setting `testargs(TRUE)`, however, MuPAD checks every assertion and stops with an error if boolean evaluation of `cond` does not give `TRUE`.

Assertions are a major debugging tool for programmers: by stating frequently what they think to have achieved, programmers make it easy for themselves to detect the first unintended intermediate result.

Examples

Example 1

Suppose we want to write a function `f` that takes an integer as its argument and returns 0 if that integer is a multiple of 3, and 1 otherwise. One idea how to code this could be the following: given an integer n, n modulo 3 must be equal to one of - 1, 1, or 0. In any case, `abs(n mod 3)` should do what we want:

```
f := proc(n: DOM_INT): DOM_INT
local k: DOM_INT;
begin
  k := n mod 3;
  assert(k = 1 or k = -1 or k = 0);
  abs(k)
end_proc
```

```
proc f(n) ... end
```

Checking assertions is switched on or off using `testargs`:

```
oldtestargs := testargs(): testargs(FALSE): f(5)
```

$$2$$

The result does not equal 1. For debugging purposes, we switch on assertion checking:

```
testargs(TRUE): f(5)
```

```
Error: Assertion 'k = 1 or k = -1 or k = 0' has failed. [f]
```

This shows that the local variable `k` must have gotten a wrong value. Indeed, when writing our program we overlooked the difference between `mod` and the symmetric remainder given by `mods`.

```
testargs(oldtestargs):
```

Parameters

cond

A boolean expression

Return Values

`assert` returns `TRUE` or raises an error.

See Also

MuPAD Functions

`testargs`

assign

Perform assignments given as equations

Syntax

```
assign(L)
```

```
assign(L, S)
```

Description

For each equation in a list, a set, or a table of equations `L`, `assign(L)` evaluates both sides of the equation and assigns the evaluated right hand side to the evaluated left hand side.

`assign(L, S)` does the same, but only for those equations whose left hand side is in the set `S`.

Since the arguments of `assign` are evaluated, the *evaluation* of the left hand side of each equation in `L` must be an admissible left hand side for an assignment. See the help page of the assignment operator `:=` for details.

Several assignments are performed from left to right. See “Example 4” on page 1-276.

`assign` can be conveniently used after a call to `solve` to assign a particular solution of a system of equations to the unknowns. See “Example 5” on page 1-276.

Examples

Example 1

We assign values to the three identifiers `B1,B2,B3`:

```
delete B1, B2, B3:
assign([B1 = 42, B2 = 13, B3 = 666]): B1, B2, B3
```

$$42,\ 13,\ 666$$

We specify a second argument to carry out only those assignments with left hand side `B1`:

```
delete B1, B2, B3:
assign([B1 = 42, B2 = 13, B3 = 666], {B1}): B1, B2, B3
```

$$42,\ B2,\ B3$$

The first argument may also be a table of equations:

```
delete B1, B2, B3:
assign(table(B1 = 42, B2 = 13, B3 = 666)): B1, B2, B3
```

$$42,\ 13,\ 666$$

Example 2

Unlike `_assign`, `assign` evaluates the left hand sides:

```
delete a, b: a := b: assign({a = 3}): a, b
```

$$3,\ 3$$

```
delete a, b: a := b: a := 3: a, b
```

$$3,\ b$$

Example 3

The object assigned may also be a sequence:

```
assign([X=(2,7)])
```

$$[X = (2,\ 7)]$$

```
X
```

$$2, 7$$

Example 4

The assignments are carried out one after another, from left to right. Since the right hand side is evaluated, the identifier `C` gets the value `3` in the following example:

```
assign([B=3, C=B])
```

$$[B = 3, C = B]$$

```
level(C,1)
```

$$3$$

Example 5

When called for an algebraic system, `solve` often returns a set of lists of assignments. `assign` can then be used to assign the solutions to the variables of the system:

```
sys:={x^2+y^2=2, x+y=5}:
S:= solve(sys)
```

$$\left\{\left[x = \frac{5}{2} - \frac{\sqrt{21}\,i}{2}, y = \frac{5}{2} + \frac{\sqrt{21}\,i}{2}\right], \left[x = \frac{5}{2} + \frac{\sqrt{21}\,i}{2}, y = \frac{5}{2} - \frac{\sqrt{21}\,i}{2}\right]\right\}$$

We want to check whether the first solution is really a solution:

```
assign(S[1]): sys
```

$$\left\{5 = 5, \left(-\frac{5}{2} + \frac{\sqrt{21}\,i}{2}\right)^2 + \left(\frac{5}{2} + \frac{\sqrt{21}\,i}{2}\right)^2 = 2\right\}$$

Things become clearer if we use floating-point evaluation:

```
float(sys)
```

$$\{2.0 = 2.0,\ 5.0 = 5.0\}$$

Parameters

L

A list, a set, or a table of equations

S

A set

Return Values

L.

See Also

MuPAD Functions

`:=` | `_assign` | `assignElements` | `delete` | `evalassign`

assignElements

Assign values to entries of an array, a list, or a table

Syntax

```
assignElements(L, [index1] = value1, [index2] = value2, …)

assignElements(L, [[index1], value1], [[index2], value2], …)
```

Description

`assignElements(L, [index1] = value1, [index2] = value2, ...)` returns a copy of `L` with `value1` stored at `index1`, `value2` stored at `index2`, etc.

`R:= assignElements(L,[index1]=value1,[index2]=value2,...)` has the same effect as the sequence of assignments `R:=L: R[index1]:=value1: R[index2]:=value2: ... R`, but is more efficient.

`assignElements` returns a modified copy of its first argument, which remains unchanged. See “Example 1” on page 1-279.

The second variant of the `assignElements` call, with lists instead of equations, is equivalent to the first variant. In fact, both equations and lists may be mixed in a single call. See “Example 1” on page 1-279.

All assignments are performed simultaneously, i.e., the order of the arguments is irrelevant. See “Example 3” on page 1-280.

All rules for indexed assignments apply, in particular with respect to the validity of indices. If `L` is a list, the indices must be positive integers not exceeding the length of `L`. If `L` is an array, the indices must be (sequences of) integers matching the dimension and lying within the valid ranges of the array. If `L` is a table, the indices may be arbitrary objects.

Examples

Example 1

Assignments may given as equations or lists, and both forms may be mixed in a single call:

```
L := array(1..3, [3, 4, 5]);
assignElements(L, [1] = one, [2] = two, [3] = three);
assignElements(L, [[1], one], [[2], two], [[3], three]);
assignElements(L, [1] = one, [[2], two], [3] = three);
```

$$(3\ 4\ 5)$$

$$(\text{one}\ \text{two}\ \text{three})$$

$$(\text{one}\ \text{two}\ \text{three})$$

$$(\text{one}\ \text{two}\ \text{three})$$

The array `L` itself is not modified by `assignElements`:

```
L
```

$$(3\ 4\ 5)$$

Example 2

Sequences, too, may be assigned as values to array elements, but they must be put in parentheses:

```
R := assignElements(array(1..2), [1] = (1, 7), [2] = PI);
[R[1]], [R[2]]
```

$$((1, 7)\ \pi)$$

$[1, 7], [\pi]$

Example 3

The sequence generator $ is useful to create sequences of assignments:

```
L := [i $ i = 1..10];
assignElements(L, [i] = L[i] + L[i + 1] $ i = 1..9)
```

$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$

$[3, 5, 7, 9, 11, 13, 15, 17, 19, 10]$

The order of the arguments is irrelevant:

```
assignElements(L, [10 - i] = L[10 - i] + L[11 - i] $ i = 1..9)
```

$[3, 5, 7, 9, 11, 13, 15, 17, 19, 10]$

Example 4

The indices of a table may be arbitrary objects, for example, strings:

```
assignElements(table(), [expr2text(i)] = i^2 $ i = 1..4)
```

"1"	1
"2"	4
"3"	9
"4"	16

Example 5

For arrays of dimension greater than one, the indices are sequences of as many integers as determined by the dimension of the array:

```
assignElements(array(1..3, 1..3),
  ([i, j] = i + j $ i = 1..3) $ j = 1..3)
```

$$\begin{pmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \\ 4 & 5 & 6 \end{pmatrix}$$

Parameters

`L`

An array, an hfarray, a list, or a table

`index1, index2, …`

Valid indices for `L`

`value1, value2, …`

Any MuPAD objects

Return Values

Object of the same type as `L`.

See Also

MuPAD Domains
`DOM_ARRAY` | `DOM_HFARRAY` | `DOM_LIST` | `DOM_TABLE`

MuPAD Functions
`:=` | `_assign` | `_index` | `array` | `assign` | `delete` | `evalassign` | `table`

assume

Set permanent assumption

Syntax

```
assume(condition)
```

```
assume(expr, set)
```

Description

`assume(condition)` sets the assumption that condition `condition` is true for all further calculations. This call removes all previous assumptions containing identifiers used in `condition`.

`assume(expr, set)` attaches the property `set` to the identifier or expression `expr`. This call overwrites all previous assumptions containing identifiers used in `expr`.

Assumptions are mathematical conditions that are assumed to hold true for all calculations. By default, all MuPAD identifiers are independent of each other and can take any value in the complex plane. For example, `sign(1 + x^2)` cannot be simplified any more because MuPAD assumes that `x` is a complex number. If you set an assumption that `x` is a real number, then MuPAD can simplify `sign(1 + x^2)` to `1`.

For this reason, many MuPAD functions return very general or piecewise-defined results that depend on further conditions. For example, `solve` or `int` can return `piecewise` results.

Many mathematical theorems hold only under certain conditions. For example, `x^b*y^b = (x*y)^b` holds if `b` is an integer. But this equation is not true for all combinations of `x`, `y`, and `b`. For example, it is not true if `x = y = -1, b = 1/2`. In such cases, you can use assumptions to get more specific results.

If you use `assume` inside a function or procedure, MuPAD uses the new assumption and ignores existing assumptions only inside the function or procedure. After the function or procedure call, MuPAD removes the new assumption and restores the assumptions that were set before the function or procedure call.

If `condition` is a relation (for example, `x < y`), then MuPAD implicitly assumes that both sides of the relation are real. See “Example 4” on page 1-285.

To delete assumptions previously set on `x`, use `unassume(x)` or `delete x`.

When you assign a value to an identifier with assumptions, the assigned value can be inconsistent with existing assumptions. Assignments overwrite all assumptions previously set on an identifier. See “Example 5” on page 1-286.

`assume` accepts any `condition` and Boolean combinations of `conditions`. See “Example 7” on page 1-288.

If `expr` is a list, vector, or matrix, use the syntax `assume(expr, set)`. Here `set` must be specified as one of `C_`, `R_`, `Q_`, `Z_`, `N_`, or an expression constructed with the set operations, such as `union`, `intersect`, or `minus`. `set` also can be a function of the `Type` library, for example, `Type::Real`, `Type::Integer`, `Type::PosInt`, and so on.

Do not use the syntaxes `assume(expr in set)` and `assume(condition)` for nonscalar `expr`.

Examples

Example 1

Set an assumption that identifier `n` is an integer. Here, `assume(n, Type::Integer)` is equivalent to `assume(n in Z_)` because `n` is a scalar.

```
assume(n, Type::Integer):
assume(n in Z_):
getprop(n);
```

$$\mathbb{Z}$$

Check if `n^2` is a nonnegative integer. MuPAD uses the assumption that you set on `n`.

```
is(n^2, Type::NonNegInt)
```

$$\text{TRUE}$$

Other system functions take this assumption into account:

```
abs(n^2 + 1);
simplify(sin(2*n*PI))
```

$$n^2 + 1$$

$$0$$

For further computations, delete the identifier `n`:

```
delete n
```

Example 2

To keep the existing assumptions and combine them with the new ones, use `assumeAlso`:

```
assume(n, Type::Integer):
getprop(n);
```

$$\mathbb{Z}$$

```
assumeAlso(n, Type::Positive):
getprop(n);
```

$$\mathbb{Z} \cap [1, \infty)$$

For further computations, delete the identifier `n`:

```
delete n
```

Alternatively, set multiple assumptions in one function call:

```
assume(n, Type::Integer and Type::Positive):
getprop(n);
```

$$\mathbb{Z} \cap [1, \infty)$$

For further computations, delete the identifier `n`:

```
delete n
```

Example 3

You can set separate assumptions on the real and imaginary parts of an identifier:

```
assume(Re(z) > 0);
assumeAlso(Im(z) < 0):

abs(Re(z));
sign(Im(z))
```

$$\Re(z)$$

$$-1$$

```
is(z, Type::Real), is(z > 0)
```

FALSE, FALSE

For further computations, delete the identifier `z`:

```
delete z
```

Example 4

Using `assume`, set the assumption `x > y`. Assumptions set as relations affect the properties of both identifiers.

```
assume(x > y)
```

To see the assumptions set on identifiers, use `getprop`:

```
getprop(x);
getprop(y);
```

$$(y, \infty)$$

$$(-\infty, x)$$

To keep an existing assumption on `y` and add a new one, use `assumeAlso`. For example, add the new assumption that `y` is greater than `0` while keeping the assumption that `y` is less than `x`:

```
assumeAlso(y > 0)
```

```
is(x^2 >= y^2)
```

$$\text{TRUE}$$

Relations, such as `x > y`, imply that the involved identifiers are real:

```
is(x, Type::Real), is(y, Type::Real)
```

$$\text{TRUE}, \text{TRUE}$$

You also can set a relational assumption where one side is not an identifier, but an expression:

```
assume(x > 1/y)
```

```
getprop(x);
getprop(y)
```

$$\left(\frac{1}{y}, \infty\right)$$

$$\mathbb{R}$$

For further computations, delete the identifiers `x` and `y`:

```
delete x, y
```

Example 5

`_assign` and `:=` do not check if an identifier has any assumptions. The assignment operation overwrites all assumptions:

```
assume(a > 0):
a := -2:
a, getprop(a)
```

$$-2,\ \{-2\}$$

For further computations, delete the identifier `a`:

```
delete a
```

Example 6

Set the assumption that x is positive and find the absolute value of `x`, the sign of `x`, and the real and imaginary parts of `x`. These system functions take assumptions set on identifiers into account:

```
assume(x > 0):
abs(x), sign(x), Re(x), Im(x)
```

$$x,\ 1,\ x,\ 0$$

Try expanding the expression `ln(z1*z2)` without additional assumptions. It does not expand because `ln(z1*z2) = ln(z1) + ln(z2)` is not true for arbitrary `z1`, `z2` in the complex plane:

```
expand(ln(z1*z2))
```

$$\ln(z1\ z2)$$

Now, set the assumption that one number is real and positive. Expand the same expression:

```
assume(z1 > 0): expand(ln(z1*z2))
```

$$\ln(z1) + \ln(z2)$$

For further computations, remove the assumptions on `x` and `z1`:

```
unassume(x); unassume(z1)
```

Example 7

Set these two assumptions on the identifier `a`. To combine the assumptions, use the Boolean operator `and`:

```
assume(a > 0 and a in Z_):
is(a = 0);
is(a = 1/2);
is(a = 2);
```

```
FALSE

FALSE

UNKNOWN
```

Parameters

`expr`

Identifier, mathematical expression, list, vector, or matrix containing identifiers.

If `expr` is a list, vector, or matrix, then only the syntax `assume(expr, set)` is valid.

`set`

Property representing a set of numbers or a set returned by `solve`.

For example, this set can be an element of `Dom::Interval`, `Dom::ImageSet`, `piecewise`, or one of `C_`, `R_`, `Q_`, `Z_`, `N_`. It also can be an expression constructed with the set operations, such as `union`,, `intersect` or `minus`. For more examples, see "Properties".

`condition`

Equality, inequality, element of relation, or Boolean combination (with the operators `and` or `or`).

Return Values

Void object `null()` of type `DOM_NULL`.

See Also

MuPAD Functions

`assumeAlso` | `assuming` | `assumingAlso` | `getprop` | `is` | `property::hasprop` | `property::showprops` | `unassume`

assumeAlso

Add permanent assumption

Syntax

```
assumeAlso(condition)
```

```
assumeAlso(expr, set)
```

Description

`assumeAlso(condition)` adds the assumption that `condition` is true for all further calculations. It does not remove previous assumptions containing identifiers used in `condition`.

`assumeAlso(expr, set)` attaches the property `set` to the identifier or expression `x`. It does not remove previous assumptions containing identifiers used in `expr`.

Assumptions are mathematical conditions that are assumed to hold true for all calculations. By default, all MuPAD identifiers are independent of each other and can take any value in the complex plane. For example, `sign(1 + x^2)` cannot be simplified any more because `x` MuPAD assumes that `x` is a complex number. If you set an assumption that `x` is a real number, then MuPAD can simplify `sign(1 + x^2)` to `1`.

For this reason, many MuPAD functions return very general or piecewise-defined results depending on further conditions. For example, `solve` or `int` can return `piecewise` results.

Many mathematical theorems hold only under certain conditions. For example, `x^b*y^b=(x*y)^b` holds if `b` is an integer. But this equation is not true for all combinations of `x`, `y`, and `b`. For example, it is not true if `x = y = -1, b = 1/2`. In such cases, you can use assumptions to get more specific results.

If you use `assumeAlso` inside a function or procedure, MuPAD uses that assumption only inside the function or procedure. After the function or procedure call, MuPAD removes that assumption and only keeps the assumptions that were set before the function or procedure call.

If `condition` is a relation (for example, `x < y`), then MuPAD implicitly assumes that both sides of the relation are real. See “Example 4” on page 1-293.

To delete assumptions previously set on `x`, use `unassume(x)` or `delete x`.

When assigning a value to an identifier with assumptions, the assigned value can be inconsistent with existing assumptions. Assignments overwrite all assumptions previously set on an identifier. See “Example 5” on page 1-295.

`assumeAlso` accepts any `condition` and Boolean combinations of `condition`s. See “Example 7” on page 1-295.

If `expr` is a list, vector, or matrix, use the syntax `assumeAlso(expr, set)`. Here, `set` must be specified as one of `C_`, `R_`, `Q_`, `Z_`, `N_`, or an expression constructed with the set operations, such as `union`, `intersect`, or `minus`. `set` also can be a function of the `Type` library, for example, `Type::Real`, `Type::Integer`, `Type::PosInt`, and so on.

Do not use the syntaxes `assumeAlso(expr in set)` and `assumeAlso(condition)` for nonscalar `expr`.

Examples

Example 1

Solve this equation without any assumptions on the variable `x`:

```
solve(x^5 - x, x)
```

$$\{-1, 0, 1, -i, i\}$$

Suppose, your computations deal with real numbers only. In this case, use the `assume` function to set the permanent assumption that `x` is real:

```
assume(x in R_)
```

If you solve the same equation now, you will get three real solutions:

```
solve(x^5 - x, x)
```

$$\{-1, 0, 1\}$$

If you also want to get only nonzero solutions, use `assumeAlso` to add the corresponding assumption:

```
assumeAlso(x <> 0);
solve(x^5 - x, x)
```

$$\{-1, 1\}$$

MuPAD keeps both assumptions for further computations:

```
getprop(x)
```

$$\mathbb{R} \setminus \{0\}$$

For further computations, delete the identifier `x`:

```
delete x
```

Example 2

When you use `assumeAlso`, MuPAD does not remove existing assumptions. Instead, it combines them with new assumptions. For example, assume that `n` is an integer:

```
assume(n, Type::Integer):
getprop(n);
```

$$\mathbb{Z}$$

Add the assumption that `n` is positive:

```
assumeAlso(n, Type::Positive):
getprop(n);
```

$$\mathbb{Z} \cap [1, \infty)$$

For further computations, delete the identifier `n`:

```
delete n
```

Alternatively, set multiple assumptions in one function call using the Boolean operator `and`:

```
assume(n, Type::Integer and Type::Positive):
getprop(n);
```

$$\mathbb{Z} \cap [1, \infty)$$

For further computations, delete the identifier `n`:

```
delete n
```

Example 3

You can set separate assumptions on the real and imaginary parts of an identifier:

```
assume(Re(z) > 0);
assumeAlso(Im(z) < 0):

abs(Re(z));
sign(Im(z))
```

$$\Re(z)$$

$$-1$$

For further computations, delete the identifier `z`:

```
delete z
```

Example 4

Using `assume`, set the assumption `x > y`. Assumptions set as relations affect the properties of both identifiers.

```
assume(x > y)
```

To see the assumptions set on identifiers, use `getprop`:

```
getprop(x);
```

```
getprop(y);
```

$$(y, \infty)$$

$$(-\infty, x)$$

To keep an existing assumption on `y` and add a new one, use `assumeAlso`. For example, add the new assumption that `y` is greater than `0` while keeping the assumption that `y` is less than `x`:

```
assumeAlso(y > 0)
```

```
is(x^2 >= y^2)
```

TRUE

Relations, such as `x > y`, imply that the involved identifiers are real:

```
is(x, Type::Real), is(y, Type::Real)
```

TRUE, TRUE

```
delete x, y:
```

You also can add a relational assumption where one side is not an identifier, but an expression:

```
assumeAlso(x > 1/y)
```

```
getprop(x);
getprop(y)
```

$$\left(\frac{1}{y}, \infty\right)$$

$$\mathbb{R}$$

For further computations, delete the identifiers `x` and `y`:

```
delete x, y
```

Example 5

_assign and := do not check if an identifier has any assumptions. The assignment operation overwrites all assumptions:

```
assume(a > 0):
a := -2:
a, getprop(a)
```

$$-2, \{-2\}$$

For further computations, delete the identifier a:

```
delete a
```

Example 7

Use assume to set the assumption that the identifier a is positive:

```
assume(a > 0)
```

Now, add two new assumptions using one call to assumeAlso. To combine the assumptions, use the Boolean operator and:

```
assumeAlso(a in Z_ and a < 5):
is(a = 0);
is(a = 1/2);
is(a = 2);
is(a = 6);
```

FALSE

FALSE

UNKNOWN

FALSE

Parameters

expr

Identifier, mathematical expression, list, vector, or matrix containing identifiers.

If `expr` is a list, vector, or matrix, then only the syntax `assumeAlso(expr, set)` is valid.

set

Property representing a set of numbers or a set returned by `solve`. This set can be an element of `Dom::Interval`, `Dom::ImageSet`, `piecewise`, or one of `C_`, `R_`, `Q_`, `Z_`, `N_`. It also can be an expression constructed with the set operations, such as `union`, `intersect` or `minus`. For more examples, see "Properties".

condition

Equality, inequality, element of relation, or Boolean combination (with the operators `and` or `or`).

Return Values

Void object `null()` of type `DOM_NULL`.

See Also

MuPAD Functions

`assume` | `assuming` | `assumingAlso` | `getprop` | `is` | `property::hasprop` | `property::showprops` | `unassume`

assuming, _assuming

Set temporary assumption

Syntax

```
calculation assuming condition
```

```
calculation assuming (expr, set)
```

```
_assuming(calculation, condition)
```

```
_assuming(calculation, (expr, set))
```

Description

`calculation assuming condition` evaluates `calculation` under the assumption that `condition` is true for that calculation.

`calculation assuming(expr, set)` temporarily attaches the property `set` to the identifier or expression `expr` and evaluates `calculation`. This call ignores all previous assumptions containing identifiers used in `expr`.

`calculation assuming condition` is equivalent to `_assuming(calculation, condition)`.

`calculation assuming (expr, set)` is equivalent to `_assuming(calculation, (expr, set))`.

`assuming` sets temporary assumptions. Temporary assumptions hold true only while the argument `calculation` is evaluated. After this evaluation, MuPAD removes these assumptions. Therefore, they do not affect further computations. MuPAD also removes temporary assumptions if the evaluation stops with an error.

`assuming` temporarily overwrites existing assumptions. If you have permanent assumptions, MuPAD ignores them while evaluating `calculation`. Instead, it uses temporary assumptions set by `assumimg`.

If assumptions contain linear equations with one variable, `assuming` solves these equations. Then the command inserts the solutions into `calculation` and evaluates the result. See “Example 6” on page 1-300.

If `expr` is a list, vector, or matrix, use the syntaxes `calculation assuming(expr, set)` and `_assuming(calculation, (expr, set))`. Here, `set` is specified as one of `C_`, `R_`, `Q_`, `Z_`, `N_`, or an expression constructed with the set operations, such as `union`, `intersect`, or `minus`. `set` also can be a function of the `Type` library, for example, `Type::Real`, `Type::Integer`, `Type::PosInt`, and so on.

Do not use the syntaxes `calculation assuming (expr in set)` (or its equivalent `_assuming(calculation, expr in set)`) and `calculation assuming condition` (or its equivalent `_assuming(calculation, condition)`) for nonscalar `expr`.

Examples

Example 1

Find the sign of the expression `x^2 + 1` assuming that the identifier `x` represents a real number:

```
sign(x^2+1) assuming (x, Type::Real)
```

```
    1
```

Example 2

Simplify this sine function assuming that `n` is an integer:

```
simplify(sin(n*PI)) assuming n in Z_
```

```
    0
```

Example 3

Additional assumptions let you simplify some expressions. For example, compute the right limit of x^p:

```
limit(x^p, x = 0, Right)
```

$$\begin{cases} 1 & \text{if } p = 0 \\ \infty & \text{if } p < 0 \\ 0 & \text{if } 0 < \Re(p) \\ \lim_{x \to 0^+} x^p & \text{if } \Re(p) < 0 \wedge \neg\, p \le 0 \end{cases}$$

Compute the right limit of the same expression for negative, positive, and zero values of p:

```
limit(x^p, x = 0, Right) assuming p < 0
```

$$\infty$$

```
limit(x^p, x = 0, Right) assuming p > 0
```

$$0$$

```
limit(x^p, x = 0, Right) assuming p = 0
```

$$1$$

Example 4

Assumptions set by `assuming` are temporary. They do not affect any previous or future computations:

```
getprop(a);
getprop(a) assuming a > 0;
getprop(a)
```

$$\mathbb{C}$$

$$(0, \infty)$$

$$\mathbb{C}$$

Example 5

If you already use a permanent assumption and want to add a temporary assumption on the same object, do not use `assuming`. It temporarily overwrites the permanent assumption:

```
assume(x in Z_):
solve(x^3 - (44*x^2)/3 + (148*x)/3 - 80/3 = 0, x) assuming x < 5
```

$$\left\{\frac{2}{3}, 4\right\}$$

Instead, use `assumingAlso`:

```
solve(x^3 - (44*x^2)/3 + (148*x)/3 - 80/3 = 0, x) assumingAlso x < 5
```

$$\{4\}$$

Example 6

If assumptions contain linear equations with one variable, `assuming` solves the equations, inserts the solutions into the expression, and then evaluates the expression:

```
a^2 + 1 assuming a - 2 = 1;
```

$$10$$

`assume` and `assumeAlso` do not solve equations:

```
assume(a - 2 = 1) ;
a^2 + 1
```

$$a^2 + 1$$

Parameters

`calculation`

Any MuPAD command or expression that you want to evaluate under the temporary assumption.

`condition`

Equality, inequality, element of relation, or Boolean combination (with the operators `and` or `or`).

`expr`

Identifier, mathematical expression, list, vector, or matrix containing identifiers.

If `expr` is a list, vector, or matrix, then only the syntaxes `calculation assuming(expr, set)` and `_assuming(calculation, (expr, set))` are valid.

`set`

Property representing a set of numbers or a set returned by `solve`.

For example, this set can be an element of `Dom::Interval`, `Dom::ImageSet`, `piecewise`, or one of `C_`, `R_`, `Q_`, `Z_`, `N_`. It also can be an expression constructed with the set operations, such as `union`, `intersect` or `minus`. For more examples, see "Properties".

Return Values

`assuming` returns the result of evaluating `calculation`.

See Also

MuPAD Functions

`assume` | `assumeAlso` | `assumingAlso` | `getprop` | `is` | `property::hasprop` | `property::showprops` | `unassume`

assumingAlso, _assumingAlso

Add temporary assumption

Syntax

```
calculation assumingAlso condition

calculation assumingAlso (expr, set)

_assumingAlso(calculation, condition)

_assumingAlso(calculation, (expr, set))
```

Description

`calculation assumingAlso condition` evaluates `calculation` under all existing assumptions along with the new assumption that `condition` is true for that calculation.

`calculation assumingAlso(expr, set)` temporarily attaches the property `set` to the identifier or expression `expr` and evaluates `calculation`. This call takes into account all previous assumptions containing identifiers used in `expr`.

`calculation assumingAlso condition` is equivalent to `_assumingAlso(calculation, condition)`.

`calculation assumingAlso (expr, set)` is equivalent to `_assumingAlso(calculation, (expr, set))`.

`assumingAlso` sets temporary assumptions in addition to existing permanent assumptions. Here, `condition` holds true only while the argument `calculation` is evaluated. After this evaluation, `condition` is removed. Therefore, it does not affect further computations. `condition` is also removed if the evaluation stops with an error.

If assumptions contain linear equations with one variable, `assumingAlso` solves these equations. Then the command inserts the solutions into `calculation` and evaluates the result. See “Example 3” on page 1-304.

If `expr` is a list, vector, or matrix, use the syntaxes `calculation assumingAlso(expr, set)` and `_assumingAlso(calculation, (expr, set))`.

Here, `set` is specified as one of `C_`, `R_`, `Q_`, `Z_`, `N_`, or an expression constructed with the set operations, such as `union`, `intersect`, or `minus`. `set` also can be a function of the `Type` library, for example, `Type::Real`, `Type::Integer`, `Type::PosInt`, and so on.

Do not use the syntaxes `calculation assumingAlso (expr in set)` (or its equivalent `_assumingAlso(calculation, expr in set)`) and `calculation assumingAlso condition` (or its equivalent `_assumingAlso(calculation, condition)`) for nonscalar `expr`.

Examples

Example 1

Solve this equation without any assumptions on the variable `x`:

```
solve(x^5 - x, x)
```

$$\{-1, 0, 1, -i, i\}$$

Suppose your computations deal with real numbers only. In this case, use the `assume` function to set the permanent assumption that `x` is real:

```
assume(x in R_)
```

If you solve the same equation now, you will get three real solutions:

```
solve(x^5 - x, x)
```

$$\{-1, 0, 1\}$$

To get only nonzero solutions for this particular equation, use `assumingAlso` to temporarily add the corresponding assumption:

```
solve(x^5 - x, x) assumingAlso x <> 0
```

$$\{-1, 1\}$$

After solving this equation, MuPAD discards the temporary assumption, but keeps the permanent one:

```
getprop(x)
```

$$\mathbb{R}$$

Example 2

If you already use a permanent assumption and want to add a temporary assumption on the same object, do not use `assuming`. It temporarily overwrites the permanent assumption:

```
assume(x in Z_):
solve(x^3 - (44*x^2)/3 + (148*x)/3 - 80/3 = 0, x) assuming x < 5
```

$$\left\{\frac{2}{3}, 4\right\}$$

Instead, use `assumingAlso`:

```
solve(x^3 - (44*x^2)/3 + (148*x)/3 - 80/3 = 0, x) assumingAlso x < 5
```

$$\{4\}$$

Example 3

If assumptions contain linear equations with one variable, `assumingAlso` solves the equations, inserts the solutions into the expression, and then evaluates the expression:

```
a^2 + 1 assumingAlso a - 2 = 1;
```

$$10$$

`assume` and `assumeAlso` do not solve equations:

```
assumeAlso(a - 2 = 1) ;
a^2 + 1
```

$$a^2 + 1$$

Parameters

`calculation`

Any MuPAD command or expression that you want to evaluate under the temporary assumption.

`condition`

Equality, inequality, element of relation, or Boolean combination (with the operators `and` or `or`).

`expr`

Identifier, mathematical expression, list, vector, or matrix containing identifiers.

If `expr` is a list, vector, or matrix, then only the syntaxes `calculation assumingAlso(expr, set)` and `_assumingAlso(calculation, (expr, set))` are valid.

`set`

Property representing a set of numbers or a set returned by `solve`. This set can be an element of `Dom::Interval`, `Dom::ImageSet`, `piecewise`, or one of `C_`, `R_`, `Q_`, `Z_`, `N_`. It also can be an expression constructed with the set operations, such as `union`, `intersect` or `minus`. For more examples, see "Properties".

Return Values

`assumingAlso` returns the result of evaluating `calculation`.

See Also

MuPAD Functions

`assume` | `assumeAlso` | `assuming` | `getprop` | `is` | `property::hasprop` | `property::showprops` | `unassume`

asympt

Compute an asymptotic series expansion

Syntax

```
asympt(f, x)

asympt(f, x, <order>, <dir>)

asympt(f, x = x0, <order>, <Left | Right>)
```

Description

`asympt(f, x)` computes the first terms of an asymptotic series expansion of `f` with respect to the variable `x` around the point `infinity`.

`asympt` is used to compute an asymptotic expansion of `f` when x tends to x_0. If such an expansion can be computed, a series object of domain type `Series::gseries` or `Series::Puiseux` is returned.

In contrast to the default behavior of `series`, `asympt` computes directed expansions that may be valid along the real line only.

`asympt` can compute more general types of asymptotic expansions than the related function `series`. Cf. “Example 5” on page 1-309.

If x_0 is a regular point of `f`, a pole, or an algebraic branch point, then `asympt` returns a Puiseux expansion. In this case it is recommended to use the faster function `series` instead.

If `asympt` cannot compute an asymptotic expansion, then a symbolic expression of type `"asympt"` is returned. Cf. “Example 4” on page 1-308.

The number of requested terms for the expansion is `order` if specified. Otherwise, the value of the environment variable `ORDER` is used. You can change the default value 6 by assigning a new value to `ORDER`.

The number of terms is counted from the lowest degree term on for finite expansion points, and from the highest degree term on for expansions around infinity, i.e., "`order`" has to be regarded as a "relative truncation order".

Note: The actual number of terms in the resulting series expansion may differ from the requested number of terms. See `series` for details.

The function `asympt` returns an object of domain type `Series::gseries` or `Series::Puiseux`. It can be manipulated via the standard arithmetic operations and various system functions. For example, `coeff` returns the coefficients; `expr` converts the series to an expression, removing the error term; `lmonomial` returns the leading monomial; `lterm` returns the leading term; `lcoeff` returns the leading coefficient; `map` applies a function to the coefficients; `nthcoeff` returns the n-th coefficient, `nthterm` the n-th term, and `nthmonomial` the n-th monomial.

Environment Interactions

The function is sensitive to the environment variable `ORDER`, which determines the default number of terms in series computations.

Examples

Example 1

We compute an asymptotic expansion for $x \to \infty$:

```
s := asympt(sin(1/x + exp(-x)) - sin(1/x), x)
```

$$e^{-x} - \frac{e^{-x}}{2\,x^2} + \frac{e^{-x}}{24\,x^4} - \frac{e^{-x}}{720\,x^6} + \frac{e^{-x}}{40320\,x^8} - \frac{e^{-x}}{3628800\,x^{10}} + O\left(\frac{e^{-x}}{x^{11}}\right)$$

The leading term and the third term are extracted:

```
lmonomial(s), nthterm(s, 3)
```

$$e^{-x}, \frac{e^{-x}}{x^4}$$

In the following call, only 2 terms of the expansion are requested:

```
asympt(
  exp(sin(1/x + exp(-exp(x)))) - exp(sin(1/x)), x, 2
)
```

$$e^{-e^x} + \frac{e^{-e^x}}{x} + O\left(\frac{e^{-e^x}}{x^2}\right)$$

```
delete s:
```

Example 2

We compute a expansion around a finite real point. By default, the expansion is valid "to the right" of the expansion point:

```
asympt(abs(x/(1+x)), x = 0)
```

$$x - x^2 + x^3 - x^4 + x^5 - x^6 + O(x^7)$$

A different expansion is valid "to the left" of the expansion point:

```
asympt(abs(x)/(1 + x), x = 0, Left)
```

$$-x + x^2 - x^3 + x^4 - x^5 + x^6 + O(-x^7)$$

Example 3

The following expansion is exact. Therefore, it has no "error term":

```
asympt(exp(x), x = infinity)
```

$$e^x$$

Example 4

Here is an example where `asympt` cannot compute an asymptotic series expansion:

```
asympt(cos(gamma(x*s))/s, x = infinity)
```

$$\text{asympt}\left(\frac{\cos(\Gamma(s\,x))}{s}, x = \infty\right)$$

Example 5

If we apply the function `series` to the following expression, it essentially returns the expression itself:

```
series((ln(ln(x)+ln(ln(x))) - ln(ln(x))) /
       ln(ln(x)+ln(ln(ln(x))))*ln(x), x = infinity)
```

$$\frac{\ln(x)\,(\ln(\ln(\ln(x))+\ln(x))-\ln(\ln(x)))}{\ln(\ln(\ln(\ln(x)))+\ln(x))}+O\left(\frac{1}{x^6}\right)$$

In this example, `asympt` computes a more detailed series expansion:

```
asympt((ln(ln(x)+ln(ln(x))) - ln(ln(x))) /
       ln(ln(x)+ln(ln(ln(x))))*ln(x), x = infinity)
```

$$1-\frac{\ln(\ln(x))}{2\ln(x)}-\frac{\sigma_1}{\ln(\ln(x))\ln(x)}+\frac{\sigma_2}{3\ln(x)^2}+\frac{\sigma_1}{2\ln(x)^2}+\frac{{\sigma_1}^2}{2\ln(\ln(x))\ln(x)^2}+O\left(\frac{1}{\sigma_2\ln(x)^2}\right)$$

where

$$\sigma_1 = \ln(\ln(\ln(x)))$$

$$\sigma_2 = \ln(\ln(x))^2$$

Parameters

`f`

An arithmetical expression representing a function in `x`

x

An identifier

x_0

The expansion point: an arithmetical expression; if not specified, the default expansion point `infinity` is used

order

The number of terms to be computed: a nonnegative integer; the default order is given by the environment variable `ORDER` (default value 6)

Options

Left, Right

With `Left`, the expansion is valid for real $x < x_0$; with `Right`, it is valid for $x > x_0$. For finite expansion points x_0, the default is `Right`.

Return Values

Object of domain type `Series::gseries` or `Series::Puiseux`, or an expression of type `"asympt"`.

Overloaded By

f

See Also

MuPAD Functions

`limit` | `mtaylor` | `O` | `ORDER` | `series` | `Series::gseries` | `Series::Puiseux` | `taylor` | `Type::Series`

bernoulli

Bernoulli numbers and polynomials

Syntax

```
bernoulli(n)
```

```
bernoulli(n, x)
```

Description

`bernoulli(n)` returns the n-th Bernoulli number.

`bernoulli(n, x)` returns the n-th Bernoulli polynomial in x.

The Bernoulli polynomials are defined by the generating function

$$\frac{t\,e^{x\,t}}{e^t-1}=\sum_{n=0}^{\infty}\frac{\mathrm{bernoulli}(n,x)}{n!}\,t^n$$

.

The Bernoulli numbers are defined by `bernoulli(n)` = `bernoulli(n,0)`.

An error occurs if `n` is a numerical value not representing a nonnegative integer.

If `n` is an integer larger than the value returned by `Pref::autoExpansionLimit()`, then the call `bernoulli(n)` is returned symbolically. Use `expand(bernoulli(n))` to get an explicit numerical result for large integers `n`.

If `n` contains non-numerical symbolic identifiers, then the call `bernoulli(n)` is returned symbolically. In most cases, the same holds true for calls `bernoulli(n, x)`. Some simplifications are implemented for special numerical values such as `x = 0`, `x = 1/2`, `x = 1` etc. for symbolic `n` . Cf. “Example 3” on page 1-313.

Note: Note that floating-point evaluation for high degree polynomials may be numerically unstable. Cf. “Example 4” on page 1-314.

The floating-point evaluation on the standard interval $x \in [0, 1]$ is numerically stable for arbitrary n.

Environment Interactions

When called with a floating-point value x, the function is sensitive to the environment variable DIGITS which determines the numerical working precision.

Examples

Example 1

The first Bernoulli numbers are:

```
bernoulli(n) $ n = 0..11
```

$$1, -\frac{1}{2}, \frac{1}{6}, 0, -\frac{1}{30}, 0, \frac{1}{42}, 0, -\frac{1}{30}, 0, \frac{5}{66}, 0$$

The first Bernoulli polynomials:

```
bernoulli(n, x) $ n = 0..4
```

$$1, x - \frac{1}{2}, x^2 - x + \frac{1}{6}, x^3 - \frac{3\,x^2}{2} + \frac{x}{2}, x^4 - 2\,x^3 + x^2 - \frac{1}{30}$$

If n is symbolic, then a symbolic call is returned:

```
bernoulli(n, x), bernoulli(n + 3/2, x), bernoulli(n + 5*I, x)
```

$$\text{bernoulli}(n, x), \text{bernoulli}\left(n + \frac{3}{2}, x\right), \text{bernoulli}(n + 5\,\mathrm{i}, x)$$

Example 2

If x is not an indeterminate, then the evaluation of the Bernoulli polynomial at the point x is returned:

```
bernoulli(50, 1 + I)
```

$$\frac{1325499634525572673731793891 25}{66} + 25\,i$$

```
bernoulli(3, 1 - y), expand(bernoulli(3, 1 - y))
```

$$\frac{1}{2} - \frac{3\,(y-1)^2}{2} - (y-1)^3 - \frac{y}{2},\ -y^3 + \frac{3\,y^2}{2} - \frac{y}{2}$$

Example 3

Certain simplifications occur for some special numerical values of `x`, even if `n` is symbolic:

```
bernoulli(n, 0), bernoulli(n, 1/2), bernoulli(n, 1)
```

$$\text{bernoulli}(n),\ \text{bernoulli}(n)\left(2^{1-n} - 1\right),\ (-1)^n\,\text{bernoulli}(n)$$

Calls with numerical arguments between $\frac{1}{2}$ and 1 are automatically rewritten in terms of calls with arguments between 0 and $\frac{1}{2}$:

```
bernoulli(n, 2/3), bernoulli(n, 0.7)
```

$$(-1)^n\,\text{bernoulli}\left(n,\ \frac{1}{3}\right),\ (-1)^n\,\text{bernoulli}(n,\ 0.3)$$

Calls with negative numerical arguments are automatially rewritten in terms of calls with positive arguments:

```
bernoulli(n, -2)
```

$$(-1)^n\,\text{bernoulli}(n,\ 2) + (-1)^n\,2^{n-1}\,n$$

```
bernoulli(n, -12.345)
```

$$(-1)^n\,\text{bernoulli}(n,\ 12.345) + (-1)^n\,12.345^{n-1}\,n$$

Example 4

Float evaluation of high degree polynomials may be numerically unstable:

```
exact := bernoulli(50, 1 + I): float(exact);
```

$$2.00833278\ 10^{27} + 25.0\ i$$

```
bernoulli(50, float(1 + I))
```

$$2.00833278\ 10^{27} - 10835184500.0\ i$$

```
DIGITS := 40: bernoulli(50, float(1 + I))
```

$$2.0083327795842010208057483200757575758\ 10^{27}$$

$$+ 24.999999999999999999999167000202474718 52\ i$$

```
delete exact, DIGITS:
```

Example 5

Some system functions such as `diff` or `expand` handle symbolic calls of `bernoulli`:

Some system functions such as `diff` or `expand` handle symbolic calls of `bernoulli`:

```
diff(bernoulli(n, f(x)), x)
```

$$n\ \text{bernoulli}(n-1, f(x))\ \frac{\partial}{\partial x} f(x)$$

```
expand(bernoulli(n, x + 2))
```

$$\text{bernoulli}(n, x) + \frac{n\,(x+1)^n}{x+1} + \frac{n\,x^n}{x}$$

```
expand(bernoulli(n, -x))
```

$$(-1)^n \,\mathrm{bernoulli}(n, x) + \frac{n\,(-x)^n}{x}$$

```
expand(bernoulli(n, 3*x))
```

$$\frac{3^n \,\mathrm{bernoulli}(n, x)}{3} + \frac{3^n \,\mathrm{bernoulli}\left(n, x + \frac{1}{3}\right)}{3} + \frac{3^n \,\mathrm{bernoulli}\left(n, x + \frac{2}{3}\right)}{3}$$

Parameters

n

An arithmetical expression representing a nonnegative integer

x

An arithmetical expression

Return Values

Arithmetical expression.

References

M. Abramowitz and I. Stegun, “Handbook of Mathematical Functions”, Dover Publications Inc., New York (1965).

See Also

MuPAD Functions

`euler`

bernstein

Bernstein polynomials

Syntax

```
bernstein(f, n, t)
```

```
bernstein(g, <var>, n, t)
```

Description

`bernstein(n, f, t)` with a univariate function f returns the `n`th-order Bernstein polynomial `sum(binomial(n,k)*t^k*(1-t)^(n-k)*f(k/n),k = 0..n)` evaluated at the point `t`. This polynomial is an approximation of f over the interval `[0,1]`. See "Example 1" on page 1-316.

`bernstein(g, <var>, n, t)` with a symbolic expression g returns the `n`th-order Bernstein polynomial approximation of `g`, evaluated at the point `t`. This syntax regards `g` as a univariate function of the variable `var`. You can omit specifying the variable `var` if the expression `g` is univariate. If it is multivariate, you must specify `var`. See "Example 2" on page 1-317 and "Example 3" on page 1-318.

The Bernstein representation of a polynomial is numerically stable when substituting numerical values between `0` and `1` for t. Nevertheless, if you simplify a Bernstein polynomial, the result can be unstable when substituting numerical values for t. See "Example 4" on page 1-319.

Examples

Example 1

Define the function representing a linear ramp as a MuPAD function:

```
f := t -> triangularPulse(1/4, 3/4, infinity, t):
```

Approximate `f` by the fifth-order Bernstein polynomials in the variable `t`:

```
b5 := bernstein(f, 5, t)
```

$$7\,t^3\,(t-1)^2 - 3\,t^2\,(t-1)^3 - 5\,t^4\,(t-1) + t^5$$

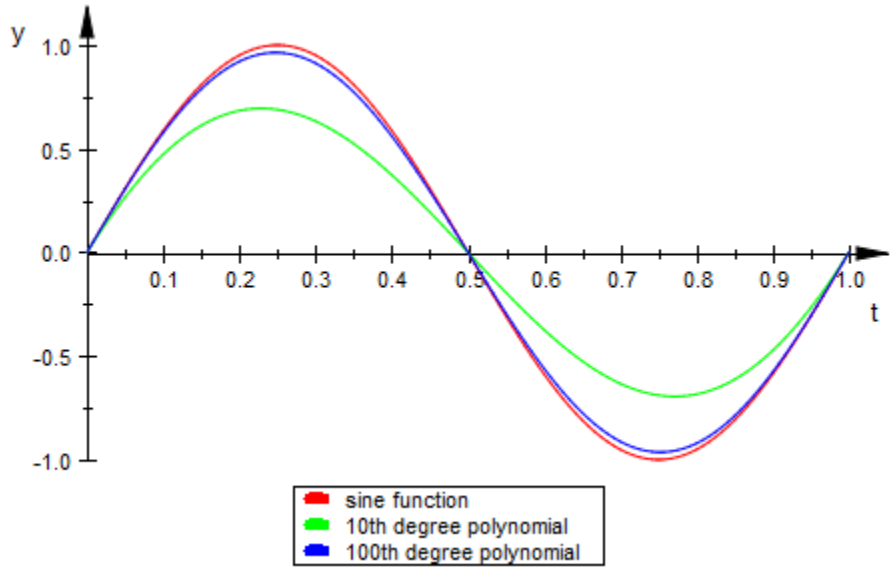
Example 2

Approximate the sine function `sin(2*PI*t)` by the 10th- and 100th-degree Bernstein polynomials:

```
b10 := bernstein(sin(2*PI*t), 10, t):
b100 := bernstein(sin(2*PI*t), 100, t):
```

Plot `sin(2*pi*t)` and its approximations:

```
p1 := plot::Function2d(sin(2*PI*t), t = 0..1,
                       LegendText = "sine function",
                       Color = RGB::Red):
p2 := plot::Function2d(b10, t = 0..1,
                       LegendText = "10th degree polynomial",
                       Color = RGB::Green):
p3 := plot::Function2d(b100, t = 0..1,
                       LegendText = "100th degree polynomial",
                       Color = RGB::Blue):
plot(p1, p2, p3, LegendVisible = TRUE)
```



Example 3

Approximate the exponential function by the second-order Bernstein polynomial in the variable `t`:

```
bernstein(exp(x), 2, t)
```

$$(t-1)^2 + t^2\,\mathrm{e} - 2\,t\,\sqrt{\mathrm{e}}\,(t-1)$$

Approximate the multivariate exponential function `y*exp(x*y)`. You must specify the variable because this expression contains more than one variable. `bernstein` regards the expression as a univariate function of that variable. For example, to treat this expression as a univariate function of `x`, enter:

```
bernstein(y*exp(x*y), x, 2, t)
```

$$y\,(t-1)^2 + t^2\,y\,\mathrm{e}^{y} - 2\,t\,y\,\mathrm{e}^{\frac{y}{2}}\,(t-1)$$

To treat this expression as a univariate function of `y`, enter:

```
bernstein(y*exp(x*y), y, 2, t)
```

$$t^2 e^x - t e^{\frac{x}{2}} (t-1)$$

Example 4

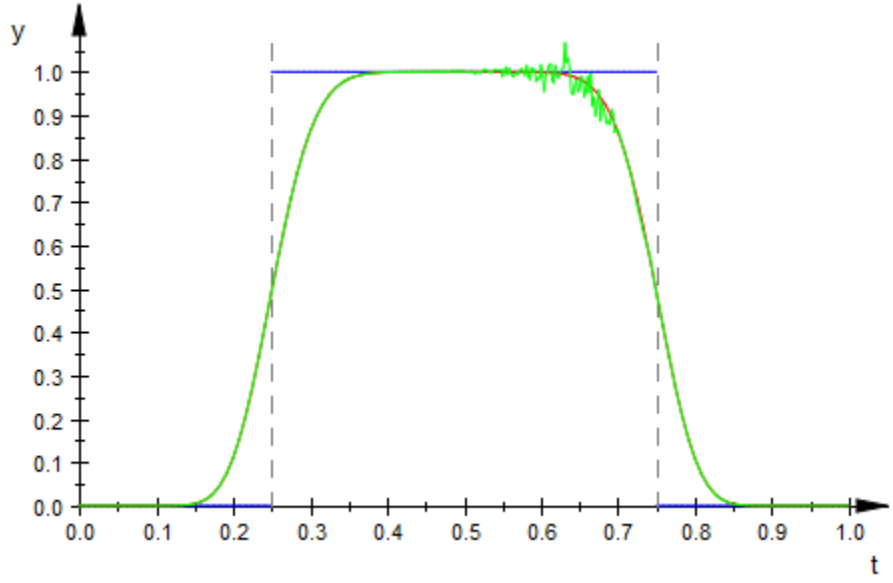
When you simplify a high-order symbolic Bernstein polynomial, the result often cannot be evaluated in a numerically stable way.

Approximate this rectangular pulse function by the 100th-degree Bernstein polynomial, and then simplify the result:

```
f := rectangularPulse(1/4,3/4,t):
f1 := bernstein(f, 100, t):
f2 := simplify(f1):
```

Compare the plot of the original rectangular pulse function `f`, its numerically stable Bernstein representation `f1`, and its simplified version `f2`. The simplified version shows a region of numerical instability.

```
plot(f, f1, f2, t = 0..1)
```



Parameters

`n`

Nonnegative integer.

`f`

Function accepting one input parameter and returning an arithmetical expression.

`g`

Arithmetical expression.

`t`

Arithmetical expression.

`var`

Indeterminate, specified as an identifier or indexed identifier.

Return Values

Arithmetical expression.

Algorithms

A Bernstein polynomial of degree `n` is defined as follows:

$$B(t) = \sum_{k=0}^{n} \beta_k \, b_{k,n}(t)$$

Here

$$b_{k,n}(t) = \binom{n}{k} t^k (1-t)^{n-k}$$

$k = 0,\ldots,n$

are the Bernstein basis polynomials, and

$$\binom{n}{k}$$

is a binomial coefficient.

The coefficients

$$\beta_k$$

are called Bernstein coefficients or Bezier coefficients.

If `f` is a continuous function on the interval `[0, 1]` and

$$B_n(f)(t) = \sum_{k=0}^{n} f\left(\frac{k}{n}\right) b_{k,n}(t)$$

is the approximating Bernstein polynomial, then

$$\lim_{n\to\infty} B_n(f)(t) = f(t)$$

uniformly in `t` on the interval `[0, 1]`.

See Also

MuPAD Functions

`bernsteinMatrix` | `binomial` | `fact` | `sum`

bernsteinMatrix

Bernstein matrix

Syntax

```
bernsteinMatrix(n, t)
```

Description

`bernsteinMatrix(n, t)`, where t is a vector of length `m`, returns the `m`-by-`(n+1)` Bernstein matrix M, such that `M(i,k+1) = binomial(n,k)*t[i]^k*(1-t[i])^(n-k)`. Here, the index `i` runs from 1 to `m`, and the index `k` runs from `0` to `n`. See "Example 2" on page 1-317 and "Example 3" on page 1-318.

The Bernstein matrix is also called the Bezier matrix.

Use Bernstein matrices to construct Bezier curves:

```
bezierCurve = bernsteinMatrix(n, t)*P
```
Here, the rows of matrix `P` specify the control points of the Bezier curve. For example, to construct the second-order 3-D Bezier curve, specify the control points as:

```
P = [p0x, p0y, p0z;  p1x, p1y, p1z;  p2x, p2y, p2z]
```

Examples

Example 1

Plot the fourth-order 2-D Bezier curve specified by the control points `p0 = [0, 1]`, `p1 = [4, 3]`, `p2 = [6, 2]`, `p3 = [3, 0]`, `p4 = [2, 4]`. Create a matrix with each row representing a control point:

```
P := matrix([[0, 1], [4, 3], [6, 2], [3, 0], [2, 4]]):
```

Construct the fourth-order Bernstein matrix `B`:

```
B := bernsteinMatrix(4, t)
```

$$\left((t-1)^4 \quad -4\,t\,(t-1)^3 \quad 6\,t^2\,(t-1)^2 \quad -4\,t^3\,(t-1) \quad t^4 \right)$$

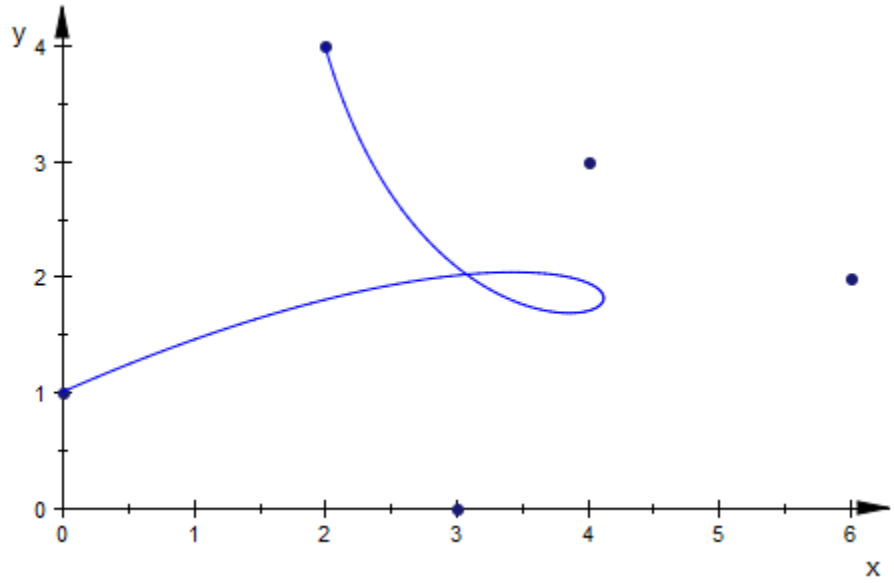
Construct the Bezier curve:

```
bezierCurve := simplify(B*P)
```

$$\left(-2\,t\,(-5\,t^3+6\,t^2+6\,t-8) \quad 5\,t^4+8\,t^3-18\,t^2+8\,t+1 \right)$$

Plot the curve and control points:

```
plot(plot::PointList2d(P),
       plot::Curve2d(bezierCurve, t = 0..a, a = 0..1))
```



Example 2

Construct the third-order 3-D Bezier curve specified by the 3-by-3 matrix `P` of control points. Each control point corresponds to a row of matrix `P`.

```
P := matrix([[0, 0, 0], [2, 2, 2], [2, -1, 1], [6, 1, 3]]):
```

Construct the Bernstein matrix:

```
B := bernsteinMatrix(3, t)
```

$$\left(-(t-1)^3 \;\; 3\,t\,(t-1)^2 \;\; -3\,t^2\,(t-1) \;\; t^3 \right)$$

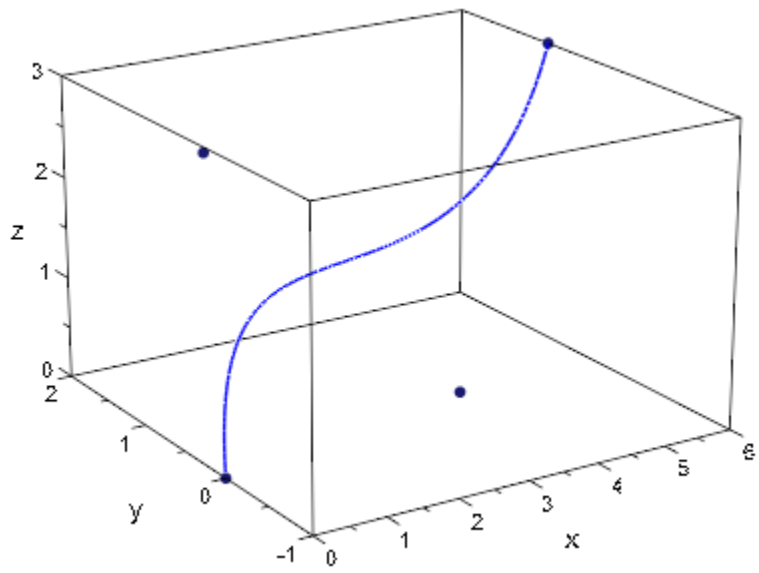
Construct the Bezier curve:

```
bezierCurve := simplify(B*P)
```

$$\left(6\,t\,(t^2-t+1) \;\; t\,(10\,t^2-15\,t+6) \;\; 3\,t\,(2\,t^2-3\,t+2) \right)$$

Plot the control points and Bezier curve:

```
plot(plot::PointList3d(P),
        plot::Curve3d(bezierCurve, t = 0..a, a = 0..1))
```



Parameters

`n`

Nonnegative integer.

`t`

Arithmetical expression, a list of expressions, or a vector of expressions. In MuPAD, a `1#m` or an `m#1` matrix represents a row or column vector, respectively.

Return Values

`nops(t)#(n+1)` matrix of the domain type `Dom::Matrix()`.

Algorithms

A Bernstein polynomial of degree `n` is defined as follows:

$$B(t) = \sum_{k=0}^{n} \beta_k \, b_{k,n}(t)$$

Here,

$$b_{k,n}(t) = \binom{n}{k} t^k (1-t)^{n-k}$$

$k = 0,\ldots,n$

are the Bernstein basis polynomials, and

$$\binom{n}{k}$$

is a binomial coefficient.

The coefficients

$$\beta_k$$

are called Bernstein coefficients or Bezier coefficients.

The Bernstein polynomial is given by the matrix multiplication of the Bernstein matrix and the vector of coefficients:

$$B(t) = b_{0n}(t)\ b_{1n}(t)\ \ldots\ b_{nn}(t) \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_n \end{pmatrix}$$

See Also

MuPAD Functions

`bernstein` | `binomial` | `fact` | `sum`

bessel

Modified Bessel functions of the first kind

Syntax

```
besselI(v, z)
```

Description

`besselI(v, z)` represents the modified Bessel functions of the first kind:

$$I_v(z) = \frac{\left(\frac{z}{2}\right)^v}{\sqrt{\pi}\,\Gamma\left(v+\frac{1}{2}\right)} \int_0^{\pi} e^{z\cos(t)} \sin(t)^{2v}\, dt$$

.

The Bessel functions are defined for complex arguments v and z.

A floating-point value is returned if either of the arguments is a floating-point number and the other argument is numerical. For most exact arguments the Bessel functions return an unevaluated function call. Special values at index $v = 0$ and/or argument $z = 0$ are implemented. Explicit symbolic expressions are returned, when the index v is a half integer. See “Example 2” on page 1-328.

For nonnegative integer indices v some of the Bessel functions have a branch cut along the negative real axis. A jump occurs when crossing this cut. See “Example 3” on page 1-328.

If floating-point approximations are desired for arguments that are exact numerical expressions, then we recommend to use `besselI(v, float(x))` rather than `float(besselI(v, x))`. In particular, for half integer indices the symbolic result `besselI(v,x)` is costly to compute. Further, floating-point evaluation of the resulting symbolic expression can be numerically unstable. See “Example 4” on page 1-329.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Unevaluated calls are returned for exact or symbolic arguments:

```
besselJ(2, 1 + I), besselK(0, x), besselY(v, x)
```

$$J_2(1+i),\ K_0(x),\ Y_v(x)$$

Floating point values are returned for floating-point arguments:

```
besselI(2, 5.0), besselK(3.2 + I, 10000.0)
```

$$17.50561497,\ 1.423757712\ 10^{-4345} + 4.555796986\ 10^{-4349}\ i$$

Example 2

Bessel functions can be expressed in terms of elementary functions if the index is an odd integer multiple of $\frac{1}{2}$:

```
besselJ(1/2, x), besselY(3/2, x)
```

$$\frac{\sqrt{2}\ \sin(x)}{\sqrt{\pi}\ \sqrt{x}},\ -\frac{\sqrt{2}\ \left(\sin(x)+\frac{\cos(x)}{x}\right)}{\sqrt{\pi}\ \sqrt{x}}$$

```
besselI(7/2, x), besselK(-7/2, x)
```

$$-\frac{\sqrt{2}\ \left(\sinh(x)\ \left(\frac{6}{x}+\frac{15}{x^3}\right)-\cosh(x)\ \left(\frac{15}{x^2}+1\right)\right)}{\sqrt{\pi}\ \sqrt{x}},\ \frac{\sqrt{2}\ \sqrt{\pi}\ e^{-x}\ \left(\frac{6}{x}+\frac{15}{x^2}+\frac{15}{x^3}+1\right)}{2\ \sqrt{x}}$$

Example 3

The negative real axis is a branch cut of the Bessel functions for non-integer indices v. A jump occurs when crossing this cut:

```
besselI(-3/4, -1.2), besselI(-3/4, -1.2 + I/10^10),
besselI(-3/4, -1.2 - I/10^10)
```

$$-0.76061492 - 0.76061492\,i,\ -0.76061492 - 0.7606149199\,i,\ -0.76061492 + 0.7606149199\,i$$

Example 4

The symbolic expressions returned by Bessel functions with half integer indices may be unsuitable for floating-point evaluation:

```
y := besselJ(51/2, PI)
```

$$\frac{\sqrt{2}\left(\frac{450675225}{\pi^4} - \frac{52650}{\pi^2} - \frac{1466947857375}{\pi^6} + \ldots + 1\right)}{\pi}$$

Floating point evaluation of this exact result is subject to numerical cancellation. The following result is dominated by round-off:

```
float(y)
```

$$-57.62024423$$

The numerical working precision has to be increased to obtain a more accurate result:

```
DIGITS:= 39: float(y)
```

$$1.1601295742121116493771032350748218236 1\,10^{-21}$$

Direct floating-point evaluation via the Bessel function yields a correct result within working precision:

```
DIGITS := 5: besselJ(51/2, float(PI))
```

$$1.1601\,10^{-21}$$

```
delete y, DIGITS:
```

Example 5

The functions `diff`, `float`, `limit`, and `series` handle expressions involving the Bessel functions:

```
diff(besselJ(0, x), x, x), float(ln(3 + besselI(17, sqrt(PI))))
```

$$\frac{J_1(x)}{x} - J_0(x),\ 1.098612289$$

```
limit(besselJ(2, x^2 + 1)*sqrt(x), x = infinity)
```

$$0$$

```
series(besselY(3, x)/x, x = infinity, 3)
```

$$\frac{\sqrt{2}\,\sigma_1}{\sqrt{\pi}\,x^{3/2}} + \frac{35\sqrt{2}\cos\left(x-\frac{7\pi}{4}\right)}{8\sqrt{\pi}\,x^{5/2}} - \frac{945\sqrt{2}\,\sigma_1}{128\sqrt{\pi}\,x^{7/2}} + O\left(\frac{1}{x^{9/2}}\right)$$

where

$$\sigma_1 = \sin\left(x - \frac{7\pi}{4}\right)$$

Parameters

v, z

arithmetical expressions

Return Values

Arithmetical expression.

Overloaded By

z

Algorithms

The modified Bessel functions $I_v(z)$ and $K_v(z)$ satisfy the modified Bessel equation:

$$z^2 \frac{\partial^2}{\partial z^2} w + z \frac{\partial}{\partial z} w - (z^2 + v^2)\, w = 0$$

.

When the index v is an integer, the modified Bessel functions of the first kind are governed by reflection formulas:

$$I_{-v}(z) = I_v(z)$$

.

See Also

MuPAD Functions

`besselJ` | `besselK` | `besselY`

besselJ

Bessel functions of the first kind

Syntax

```
besselJ(v, z)
```

Description

`besselJ(v, z)` represents the Bessel functions of the first kind:

$$J_v(z) = \frac{\left(\frac{z}{2}\right)^v}{\sqrt{\pi}\,\Gamma\left(v+\frac{1}{2}\right)} \int_0^{\pi} \cos(z\cos(t))\,\sin(t)^{2v}\,dt$$

.

The Bessel functions are defined for complex arguments v and z.

A floating-point value is returned if either of the arguments is a floating-point number and the other argument is numerical. For most exact arguments the Bessel functions return an unevaluated function call. Special values at index $v = 0$ and/or argument $z = 0$ are implemented. Explicit symbolic expressions are returned, when the index v is a half integer. See “Example 2” on page 1-333.

For nonnegative integer indices v some of the Bessel functions have a branch cut along the negative real axis. A jump occurs when crossing this cut. See “Example 3” on page 1-333.

If floating-point approximations are desired for arguments that are exact numerical expressions, then we recommend to use `besselJ(v, float(x))` rather than `float(besselJ(v, x))`. In particular, for half integer indices the symbolic result `besselJ(v,x)` is costly to compute. Further, floating-point evaluation of the resulting symbolic expression may be numerically unstable. Cf. “Example 4” on page 1-334.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Unevaluated calls are returned for exact or symbolic arguments:

```
besselJ(2, 1 + I), besselK(0, x), besselY(v, x)
```

$$J_2(1+i),\ K_0(x),\ Y_v(x)$$

Floating point values are returned for floating-point arguments:

```
besselI(2, 5.0), besselK(3.2 + I, 10000.0)
```

$$17.50561497,\ 1.423757712\ 10^{-4345} + 4.555796986\ 10^{-4349}\ i$$

Example 2

Bessel functions can be expressed in terms of elementary functions if the index is an odd integer multiple of $\frac{1}{2}$:

```
besselJ(1/2, x), besselY(3/2, x)
```

$$\frac{\sqrt{2}\,\sin(x)}{\sqrt{\pi}\,\sqrt{x}},\ -\frac{\sqrt{2}\left(\sin(x)+\frac{\cos(x)}{x}\right)}{\sqrt{\pi}\,\sqrt{x}}$$

```
besselI(7/2, x), besselK(-7/2, x)
```

$$-\frac{\sqrt{2}\left(\sinh(x)\left(\frac{6}{x}+\frac{15}{x^3}\right)-\cosh(x)\left(\frac{15}{x^2}+1\right)\right)}{\sqrt{\pi}\,\sqrt{x}},\ \frac{\sqrt{2}\,\sqrt{\pi}\,e^{-x}\left(\frac{6}{x}+\frac{15}{x^2}+\frac{15}{x^3}+1\right)}{2\,\sqrt{x}}$$

Example 3

The negative real axis is a branch cut of the Bessel functions for non-integer indices v. A jump occurs when crossing this cut:

```
besselI(-3/4, -1.2), besselI(-3/4, -1.2 + I/10^10),
besselI(-3/4, -1.2 - I/10^10)
```

$$-0.76061492 - 0.76061492\,i,\ -0.76061492 - 0.7606149199\,i,\ -0.76061492 + 0.7606149199\,i$$

Example 4

The symbolic expressions returned by Bessel functions with half integer indices may be unsuitable for floating-point evaluation:

```
y := besselJ(51/2, PI)
```

$$\frac{\sqrt{2}\left(\frac{450675225}{\pi^4} - \frac{52650}{\pi^2} - \frac{1466947857375}{\pi^6} + \ldots + 1\right)}{\pi}$$

Floating point evaluation of this exact result is subject to numerical cancellation. The following result is dominated by round-off:

```
float(y)
```

$$-57.62024423$$

The numerical working precision has to be increased to obtain a more accurate result:

```
DIGITS:= 39: float(y)
```

$$1.1601295742121116493771032350748218236 1\,10^{-21}$$

Direct floating-point evaluation via the Bessel function yields a correct result within working precision:

```
DIGITS := 5: besselJ(51/2, float(PI))
```

$$1.1601\,10^{-21}$$

```
delete y, DIGITS:
```

Example 5

The functions `diff`, `float`, `limit`, and `series` handle expressions involving the Bessel functions:

```
diff(besselJ(0, x), x, x), float(ln(3 + besselI(17, sqrt(PI))))
```

$$\frac{J_1(x)}{x} - J_0(x),\ 1.098612289$$

```
limit(besselJ(2, x^2 + 1)*sqrt(x), x = infinity)
```

$$0$$

```
series(besselY(3, x)/x, x = infinity, 3)
```

$$\frac{\sqrt{2}\,\sigma_1}{\sqrt{\pi}\,x^{3/2}} + \frac{35\sqrt{2}\cos\left(x-\frac{7\pi}{4}\right)}{8\sqrt{\pi}\,x^{5/2}} - \frac{945\sqrt{2}\,\sigma_1}{128\sqrt{\pi}\,x^{7/2}} + O\left(\frac{1}{x^{9/2}}\right)$$

where

$$\sigma_1 = \sin\left(x - \frac{7\pi}{4}\right)$$

Parameters

v, z

arithmetical expressions

Return Values

Arithmetical expression.

Overloaded By

z

Algorithms

The Bessel functions are regular (holomorphic) functions of z throughout the z-plane cut along the negative real axis, and for fixed $z \neq 0$, each is an entire (integral) function of v.

$J_v(z)$ and $Y_v(z)$ satisfy Bessel's equation in $w(v, z)$:

$$z^2 \frac{\partial^2}{\partial z^2} w + z \frac{\partial}{\partial z} w + (z^2 - v^2)\, w = 0$$

.

When the index v is an integer, the Bessel functions of the first kind are governed by reflection formulas:

$$J_{-v}(z) = (-1)^v J_v(z)$$

.

See Also

MuPAD Functions

`besselI` | `besselK` | `besselY`

besselK

Modified Bessel functions of the second kind

Syntax

```
besselK(v, z)
```

Description

`besselK(v, z)` represents the modified Bessel functions of the second kind:

$$K_v(z) = \frac{\frac{\pi}{2}\left(I_{-v}(z) - I_v(z)\right)}{\sin(v\,\pi)}$$

.

Here $I_v(z)$ are the modified Bessel functions of the first kind:

$$I_v(z) = \frac{\left(\frac{z}{2}\right)^v}{\sqrt{\pi}\,\Gamma\left(v + \frac{1}{2}\right)} \int_0^{\pi} e^{z\cos(t)} \sin(t)^{2v}\, dt$$

.

The Bessel functions are defined for complex arguments v and z.

A floating-point value is returned if either of the arguments is a floating-point number and the other argument is numerical. For most exact arguments the Bessel functions return an unevaluated function call. Special values at index $v = 0$ and/or argument $z = 0$ are implemented. Explicit symbolic expressions are returned, when the index v is a half integer. See “Example 2” on page 1-338.

For nonnegative integer indices v some of the Bessel functions have a branch cut along the negative real axis. A jump occurs when crossing this cut. See “Example 3” on page 1-339.

If floating-point approximations are desired for arguments that are exact numerical expressions, then we recommend to use `besselK(v, float(x))` rather than `float(besselK(v, x))`. In particular, for half integer indices the symbolic result

`besselK(v,x)` is costly to compute. Further, floating-point evaluation of the resulting symbolic expression may be numerically unstable. See “Example 4” on page 1-339.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Unevaluated calls are returned for exact or symbolic arguments:

```
besselJ(2, 1 + I), besselK(0, x), besselY(v, x)
```

$$J_2(1+i),\ K_0(x),\ Y_v(x)$$

Floating point values are returned for floating-point arguments:

```
besselI(2, 5.0), besselK(3.2 + I, 10000.0)
```

$$17.50561497,\ 1.423757712\ 10^{-4345} + 4.555796986\ 10^{-4349}\ i$$

Example 2

Bessel functions can be expressed in terms of elementary functions if the index is an odd integer multiple of $\frac{1}{2}$:

```
besselJ(1/2, x), besselY(3/2, x)
```

$$\frac{\sqrt{2}\ \sin(x)}{\sqrt{\pi}\ \sqrt{x}},\ -\frac{\sqrt{2}\ \left(\sin(x) + \frac{\cos(x)}{x}\right)}{\sqrt{\pi}\ \sqrt{x}}$$

```
besselI(7/2, x), besselK(-7/2, x)
```

$$-\frac{\sqrt{2}\left(\sinh(x)\left(\frac{6}{x}+\frac{15}{x^3}\right)-\cosh(x)\left(\frac{15}{x^2}+1\right)\right)}{\sqrt{\pi}\sqrt{x}},\ \frac{\sqrt{2}\sqrt{\pi}\,e^{-x}\left(\frac{6}{x}+\frac{15}{x^2}+\frac{15}{x^3}+1\right)}{2\sqrt{x}}$$

Example 3

The negative real axis is a branch cut of the Bessel functions for non-integer indices v. A jump occurs when crossing this cut:

```
besselI(-3/4, -1.2), besselI(-3/4, -1.2 + I/10^10),
besselI(-3/4, -1.2 - I/10^10)
```

$$-0.76061492 - 0.76061492\,i,\ -0.76061492 - 0.7606149199\,i,\ -0.76061492 + 0.7606149199\,i$$

Example 4

The symbolic expressions returned by Bessel functions with half integer indices may be unsuitable for floating-point evaluation:

```
y := besselJ(51/2, PI)
```

$$\frac{\sqrt{2}\left(\frac{450675225}{\pi^4}-\frac{52650}{\pi^2}-\frac{1466947857375}{\pi^6}+\ldots+1\right)}{\pi}$$

Floating point evaluation of this exact result is subject to numerical cancellation. The following result is dominated by round-off:

```
float(y)
```

$$-57.62024423$$

The numerical working precision has to be increased to obtain a more accurate result:

```
DIGITS:= 39: float(y)
```

$$1.160129574212111649377103235074821823 61\ 10^{-21}$$

Direct floating-point evaluation via the Bessel function yields a correct result within working precision:

```
DIGITS := 5: besselJ(51/2, float(PI))
```

$$1.1601\ 10^{-21}$$

```
delete y, DIGITS:
```

Example 5

The functions `diff`, `float`, `limit`, and `series` handle expressions involving the Bessel functions:

```
diff(besselJ(0, x), x, x), float(ln(3 + besselI(17, sqrt(PI))))
```

$$\frac{J_1(x)}{x} - J_0(x),\ 1.098612289$$

```
limit(besselJ(2, x^2 + 1)*sqrt(x), x = infinity)
```

$$0$$

```
series(besselY(3, x)/x, x = infinity, 3)
```

$$\frac{\sqrt{2}\,\sigma_1}{\sqrt{\pi}\,x^{3/2}} + \frac{35\sqrt{2}\cos\left(x-\frac{7\pi}{4}\right)}{8\sqrt{\pi}\,x^{5/2}} - \frac{945\sqrt{2}\,\sigma_1}{128\sqrt{\pi}\,x^{7/2}} + O\left(\frac{1}{x^{9/2}}\right)$$

where

$$\sigma_1 = \sin\left(x - \frac{7\pi}{4}\right)$$

Parameters

v, z

arithmetical expressions

Return Values

Arithmetical expression.

Overloaded By

z

Algorithms

The modified Bessel functions $I_v(z)$ and $K_v(z)$ satisfy the modified Bessel equation:

$$z^2 \frac{\partial^2}{\partial z^2} w + z \frac{\partial}{\partial z} w - (z^2 + v^2)\, w = 0$$

.

When the index v is an integer, the modified Bessel functions of the second kind are governed by reflection formula:

$$K_{-v}(z) = K_v(z)$$

.

See Also

MuPAD Functions

`besselI` | `besselJ` | `besselY`

besselY

Bessel functions of the second kind

Syntax

```
besselY(v, z)
```

Description

`besselJ(v, z)` represent the Bessel functions of the second kind:

$$Y_v(z) = \frac{J_v(z)\cos(v\,\pi) - J_{-v}(z)}{\sin(v\,\pi)}$$

.

Here $J_v(z)$ are the Bessel functions of the first kind:

$$J_v(z) = \frac{\left(\frac{z}{2}\right)^v}{\sqrt{\pi}\,\Gamma\left(v + \frac{1}{2}\right)} \int_0^{\pi} \cos(z\cos(t))\,\sin(t)^{2v}\,dt$$

.

The Bessel functions are defined for complex arguments v and z.

A floating-point value is returned if either of the arguments is a floating-point number and the other argument is numerical. For most exact arguments the Bessel functions return an unevaluated function call. Special values at index $v = 0$ and/or argument $z = 0$ are implemented. Explicit symbolic expressions are returned, when the index v is a half integer. See “Example 2” on page 1-343.

For nonnegative integer indices v some of the Bessel functions have a branch cut along the negative real axis. A jump occurs when crossing this cut. See “Example 3” on page 1-344.

If floating-point approximations are desired for arguments that are exact numerical expressions, then we recommend to use `besselY(v, float(x))` rather than `float(besselY(v, x))`. In particular, for half integer indices the symbolic result

`besselY(v,x)` is costly to compute. Further, floating-point evaluation of the resulting symbolic expression may be numerically unstable. See “Example 4” on page 1-344.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Unevaluated calls are returned for exact or symbolic arguments:

```
besselJ(2, 1 + I), besselK(0, x), besselY(v, x)
```

$$J_2(1+i),\ K_0(x),\ Y_v(x)$$

Floating point values are returned for floating-point arguments:

```
besselI(2, 5.0), besselK(3.2 + I, 10000.0)
```

$$17.50561497,\ 1.423757712\ 10^{-4345} + 4.555796986\ 10^{-4349}\ i$$

Example 2

Bessel functions can be expressed in terms of elementary functions if the index is an odd integer multiple of $\frac{1}{2}$:

```
besselJ(1/2, x), besselY(3/2, x)
```

$$\frac{\sqrt{2}\ \sin(x)}{\sqrt{\pi}\ \sqrt{x}},\ -\frac{\sqrt{2}\ \left(\sin(x) + \frac{\cos(x)}{x}\right)}{\sqrt{\pi}\ \sqrt{x}}$$

```
besselI(7/2, x), besselK(-7/2, x)
```

$$-\frac{\sqrt{2}\left(\sinh(x)\left(\frac{6}{x}+\frac{15}{x^3}\right)-\cosh(x)\left(\frac{15}{x^2}+1\right)\right)}{\sqrt{\pi}\sqrt{x}},\ \frac{\sqrt{2}\sqrt{\pi}\,e^{-x}\left(\frac{6}{x}+\frac{15}{x^2}+\frac{15}{x^3}+1\right)}{2\sqrt{x}}$$

Example 3

The negative real axis is a branch cut of the Bessel functions for non-integer indices v. A jump occurs when crossing this cut:

```
besselI(-3/4, -1.2), besselI(-3/4, -1.2 + I/10^10),
besselI(-3/4, -1.2 - I/10^10)
```

$$-0.76061492-0.76061492\,i,\ -0.76061492-0.7606149199\,i,\ -0.76061492+0.7606149199\,i$$

Example 4

The symbolic expressions returned by Bessel functions with half integer indices may be unsuitable for floating-point evaluation:

```
y := besselJ(51/2, PI)
```

$$\frac{\sqrt{2}\left(\frac{450675225}{\pi^4}-\frac{52650}{\pi^2}-\frac{1466947857375}{\pi^6}+\ldots+1\right)}{\pi}$$

Floating point evaluation of this exact result is subject to numerical cancellation. The following result is dominated by round-off:

```
float(y)
```

$$-57.62024423$$

The numerical working precision has to be increased to obtain a more accurate result:

```
DIGITS:= 39: float(y)
```

$$1.160129574212111649377103235074821823 61\ 10^{-21}$$

Direct floating-point evaluation via the Bessel function yields a correct result within working precision:

```
DIGITS := 5: besselJ(51/2, float(PI))
```

$$1.1601\ 10^{-21}$$

```
delete y, DIGITS:
```

Example 5

The functions `diff`, `float`, `limit`, and `series` handle expressions involving the Bessel functions:

```
diff(besselJ(0, x), x, x), float(ln(3 + besselI(17, sqrt(PI))))
```

$$\frac{J_1(x)}{x} - J_0(x),\ 1.098612289$$

```
limit(besselJ(2, x^2 + 1)*sqrt(x), x = infinity)
```

$$0$$

```
series(besselY(3, x)/x, x = infinity, 3)
```

$$\frac{\sqrt{2}\,\sigma_1}{\sqrt{\pi}\,x^{3/2}} + \frac{35\sqrt{2}\cos\left(x-\frac{7\pi}{4}\right)}{8\sqrt{\pi}\,x^{5/2}} - \frac{945\sqrt{2}\,\sigma_1}{128\sqrt{\pi}\,x^{7/2}} + O\left(\frac{1}{x^{9/2}}\right)$$

where

$$\sigma_1 = \sin\left(x - \frac{7\pi}{4}\right)$$

Parameters

v, z

arithmetical expressions

Return Values

Arithmetical expression.

Overloaded By

z

Algorithms

The Bessel functions are regular (holomorphic) functions of z throughout the z-plane cut along the negative real axis, and for fixed $z \neq 0$, each is an entire (integral) function of v.

$J_v(z)$ and $Y_v(z)$ satisfy Bessel's equation in $w(v, z)$:

$$z^2 \frac{\partial^2}{\partial z^2} w + z \frac{\partial}{\partial z} w + (z^2 - v^2)\, w = 0$$

.

When the index v is an integer, the Bessel functions of the second kind are governed by reflection formulas:

$$Y_{-v}(z) = (-1)^v Y_v(z)$$

.

See Also

MuPAD Functions

`besselI` | `besselJ` | `besselK`

beta

Beta function

Syntax

```
beta(x, y)
```

Description

`beta(x, y)` represents the beta function $\frac{\Gamma(x)\,\Gamma(y)}{\Gamma(x+y)}$.

The beta function is defined for complex arguments x and y.

The result is expressed by calls to the gamma function if both arguments are of type `Type::Numeric`. Note that the beta function may have a regular value, even if $\Gamma(x)$ or $\Gamma(y)$ and $\Gamma(x + y)$ are singular. In such cases `beta` returns the limit of the quotients of the singular terms.

A floating-point value is returned if both arguments are numerical and at least one of them is a floating-point value.

An unevaluated call of `beta` is returned, if none of the arguments vanishes and at least one of the arguments does not evaluate to a number of type `Type::Numeric`.

Environment Interactions

When called with floating-point arguments, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
beta(1, 5), beta(I, 3/2), beta(1, y + 1), beta(x, y)
```

$$\frac{1}{5},\ \frac{\sqrt{\pi}\,\Gamma(\mathrm{i})}{2\,\Gamma\left(\frac{3}{2}+\mathrm{i}\right)},\ \frac{1}{y+1},\ \beta(x,\ y)$$

Floating point values are computed for floating-point arguments:

```
beta(3.5, sqrt(2)), beta(sqrt(2), 2.0 + 10.0*I)
```

$$0.1395855454,\ -0.01112350756-0.03108193098\,\mathrm{i}$$

Example 2

The gamma function is singular if its argument is a nonpositive integer. Nevertheless, `beta` has a regular value for the following arguments:

```
beta(-3, 2)
```

$$\frac{1}{6}$$

Example 3

The functions `diff`, `expand` and `float` handle expressions involving `beta`:

```
diff(beta(x^2, x), x)
```

$$\beta(x,\ x^2)\,(\psi(x)+2\,x\,\psi(x^2)-\psi(x^2+x)\,(2\,x+1))$$

```
expand(beta(x - 1, y + 1))
```

$$-\frac{y\,\Gamma(x)\,\Gamma(y)}{\Gamma(x+y)-x\,\Gamma(x+y)}$$

```
float(beta(100, 1000))
```

$$7.730325902\ 10^{-147}$$

Example 4

The functions `diff` and `series` can handle `beta`:

```
diff(beta(x, y), x);
diff(beta(x, y), y);
```

$$-\beta(x, y)\,(\psi(x+y)-\psi(x))$$

$$-\beta(x, y)\,(\psi(x+y)-\psi(y))$$

```
normal(series(beta(x, y), y = 0, 3))
```

$$\frac{1}{y}-\mathrm{EULER}-\psi(x)+y\left(\frac{\psi(x)^2}{2}+\mathrm{EULER}\,\psi(x)+\frac{\mathrm{EULER}^2}{2}+\frac{\pi^2}{12}-\frac{\psi'(x)}{2}\right)+O(y^2)$$

```
series(beta(x, x), x = infinity, 4)
```

$$\frac{2\left(\frac{1}{4}\right)^x\sqrt{\pi}}{\sqrt{x}}+\frac{\left(\frac{1}{4}\right)^x\sqrt{\pi}}{4\,x^{3/2}}+\frac{\left(\frac{1}{4}\right)^x\sqrt{\pi}}{64\,x^{5/2}}+O\left(\frac{\left(\frac{1}{4}\right)^x}{x^{7/2}}\right)$$

Parameters

x, y

arithmetical expressions or floating-point intervals

Return Values

Arithmetical expression or a floating-point interval.

Overloaded By

x

See Also

MuPAD Functions
`binomial` | `fact` | `gamma` | `psi`

binomial

Binomial coefficients

Syntax

```
binomial(n, k)
```

Description

`binomial(n, k)` represents the binomial coefficient $\binom{n}{k} = \frac{n!}{(n-k)!\,k!}$.

Binomial coefficients are defined for complex arguments via the `gamma` function:

$$\binom{n}{k} = \frac{\Gamma(n+1)}{\Gamma(k+1)\,\Gamma(n-k+1)}.$$

With $\Gamma(n+1) = n!$, this coincides with the usual binomial coefficients for integer arguments satisfying $0 \le k \le n$.

A symbolic function call is returned if one of the arguments cannot be evaluated to a number of type `Type::Numeric`. However, for $k = 0$, $k = 1$, $k = n - 1$, and $k = n$, simplified results are returned for any n.

Let n be a number of type `Type::Numerical`. If k evaluates to a nonnegative integer, then $\frac{n\,(n-1)\,\ldots\,(n-k+1)}{k!}$ is returned. If $n - k$ evaluates to a nonnegative integer, then $\frac{n\,(n-1)\,\ldots\,(k+1)}{(n-k)!}$ is returned. If k or $n - k$ evaluates to a negative integer, then 0 is returned. If k evaluates to a floating-point number, then a floating-point value is returned. In all other cases, a symbolic call of `binomial` is returned.

A floating-point value is returned if both arguments are numerical and at least one of them is a floating-point value.

Environment Interactions

When called with floating-point arguments, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
binomial(10, k) $ k=-2..12
```

$$0, 0, 1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1, 0, 0$$

```
binomial(-23/12, 3), binomial(1 + I, 3)
```

$$-\frac{37835}{10368}, -\frac{i}{3}$$

```
binomial(n, k), binomial(n, 1), binomial(n, 4)
```

$$\binom{n}{k}, n, \binom{n}{4}$$

Floating point values are computed for floating-point arguments:

```
binomial(-235/123, 3.0), binomial(3.0, 1 + I)
```

$$-3.624343742, 4.411293492 + 2.205646746\,i$$

Example 2

The `expand` function handles expressions involving `binomial`:

```
binomial(n, 3) = expand(binomial(n, 3))
```

$$\binom{n}{3} = \frac{n^3}{6} - \frac{n^2}{2} + \frac{n}{3}$$

```
binomial(2, k) = expand(binomial(2, k))
```

$$\binom{2}{k} = \frac{2\sin(\pi k)}{\pi k (k-1)(k-2)}$$

The `float` attribute handles `binomial` if all arguments can be converted to floating-point numbers:

```
binomial(sin(3), 5/4), float(binomial(sin(3), 5/4))
```

$$\binom{\sin(3)}{\frac{5}{4}},\ -0.08360571366$$

Example 3

The functions `diff` and `series` can handle `binomial`:

```
diff(binomial(n, k), n);
diff(binomial(n, k), k);
```

$$-(\psi(n-k+1) - \psi(n+1))\binom{n}{k}$$

$$(\psi(n-k+1) - \psi(k+1))\binom{n}{k}$$

```
normal(series(binomial(n, k), k = 0, 3))
```

$$1 + k(\mathrm{EULER} + \psi(n+1)) + k^2\left(\frac{\psi(n+1)^2}{2} + \mathrm{EULER}\,\psi(n+1) - \frac{\psi'(n+1)}{2} + \frac{\mathrm{EULER}^2}{2} - \frac{\pi^2}{12}\right) + O(k^3)$$

```
series(binomial(2*n, n), n = infinity, 4)
```

$$\frac{4^n}{\sqrt{\pi}\,\sqrt{n}} - \frac{4^n}{8\,\sqrt{\pi}\,n^{3/2}} + \frac{4^n}{128\,\sqrt{\pi}\,n^{5/2}} + O\left(\frac{4^n}{n^{7/2}}\right)$$

Parameters

n, k

arithmetical expressions

Return Values

Arithmetical expression.

See Also

MuPAD Functions
`beta` | `fact` | `gamma` | `psi`

block

Create an object protected against evaluation

Syntax

```
block(a)
```

Description

`block(a)` creates a block, i.e., an object of special type that contains an unevaluated copy of `a`. It is treated as atomic and remains unchanged by evaluation.

`block` and domains derived thereof form a hierarchy of data types designed to provide control over the evaluation of certain subexpressions. Any object may be put as content into any type of block.

`block` is a domain If `d` is any block domain, `d(a)` creates a block belonging to that domain, with content `a`.

`block(a)` puts its argument into a block, without evaluating it. In order to evaluate `a` normally before putting it into a block, use `eval(hold(block)(a))`.

Blocks are invariant under evaluation.

Blocks of type `block` are atomic: the only operand of a block is the block itself.

Sequences may be put into blocks as well; in the case of `block`, they are not flattened. See “Example 4” on page 1-357.

You can create further block domains using inheritance. This particularly useful for creating own `evaluate` or `testtype` methods. See “Example 6” on page 1-357.

In case of nested blocks, only the outermost block is removed by both `expr` and `unblock`.

Examples

Example 1

A block is a sort of container that protects its content against evaluation:

```
y:= 1: bl:= block(1+y)
```

$$1+y$$

Blocks are atomic; thus y and 1 are not visible as operands:

```
op(bl), nops(bl)
```

$$1+y,\ 1$$

Although blocks are not arithmetical expressions, some basic arithmetical operations are defined for them:

```
collect(x+bl+x*bl, bl);
delete y, bl:
```

$$(x+1)\,(1+y)+x$$

Example 2

Transparent blocks protect against evaluation, too:

```
y:= 2:
bl:=blockTransparent(x+y+z)
```

$$x+y+z$$

However, a transparent block allows access to the operands of its content:

```
op(bl,1), subs(bl, hold(y) = y)
```

$$x,\ x+2+z$$

```
delete bl, y:
```

Example 3

With a third kind of block, suppressing evaluation may be limited to certain identifiers:

```
x:= 3: y:=1:
blockIdents({hold(x)})(x+y)
```

$$x+1$$

```
delete x, y:
```

Example 4

A block may also contain a sequence; flattening is suppressed:

```
block((x, y),z)
```

$$x, y, z$$

Example 5

The content of a block may be extracted and evaluated using `unblock`:

```
y:= 1: unblock(block(y)); delete y:
```

$$1$$

Example 6

We want to create blocks that represent arithmetical expressions. To do this, we need our own block domain that overloads `testtype`:

```
domain myblock
  inherits block;
  category  Cat::BaseCategory;
  testtype:= (bl, T) -> if T = Type::Arithmetical or T = dom then
                          TRUE
```

```
                        else
                          block::testtype(bl, T)
                        end_if;
end_domain:
```

This allows us to make the number zero invisible for the evaluator by enclosing it into a block, but to retain the option to plug it into special functions:

```
f := sin(x+myblock(0))
```

$$\sin(x+0)$$

We can now manipulate this expression, without being disturbed by automatic simplification:

```
expand(f)
```

$$\cos(0)\,\sin(x)+\cos(x)\,\sin(0)$$

```
 eval(unblock(%))
```

$$\sin(x)$$

Parameters

a

Any object or sequence of objects

Return Values

`block` creates objects of its own type.

See Also

MuPAD Functions

`blockIdents` | `blockTransparent` | `freeze` | `hold` | `unblock`

blockIdents

Create a transparent block with some identifiers protected against evaluation

Syntax

```
blockIdents(S)(a)
```

Description

`blockIdents(S)(a)` creates a transparent block which is evaluated like `a`, except that the identifiers in `S` are neither substituted by their values nor are their properties used.

`blockIdents` is a parametrized family of domains, depending on a parameter `S` which must be a set. If `d` is any block domain, `d(a)` creates a block belonging to that domain, with content `a`.

`blockIdents(S)(a)` replaces all identifiers in `a` that belong to `S` by newly created identifiers, evaluates the result and substitutes back; the final result is put into a block. In order to evaluate `a` normally before putting it into a block, use`eval(hold(blockIdents(S))(a))`.

Blocks of type `blockIdents(S)` are evaluated in the same way as their contents at the time of creation (see above).

Blocks of type `blockIndents(S)` have the same operands as their content.

Sequences can also be put into blocks.

You can create further block domains using inheritance. This particularly useful for creating own `evaluate` or `testtype` methods. See “Example 6” on page 1-361.

The call `expr(b)` replaces all transparent blocks in `b` by their content, without evaluating that content.

In case of nested blocks, only the outermost block is removed by both `expr` and `unblock`.

Examples

Example 1

A block is a sort of container that protects its content against evaluation:

```
y:= 1: bl:= block(1+y)
```

$$1+y$$

Blocks are atomic; thus y and 1 are not visible as operands:

```
op(bl), nops(bl)
```

$$1+y,\ 1$$

Although blocks are not arithmetical expressions, some basic arithmetical operations are defined for them:

```
collect(x+bl+x*bl, bl);
delete y, bl:
```

$$(x+1)\,(1+y)+x$$

Example 2

Transparent blocks protect against evaluation, too:

```
y:= 2:
bl:=blockTransparent(x+y+z)
```

$$x+y+z$$

However, a transparent block allows access to the operands of its content:

```
op(bl,1), subs(bl, hold(y) = y)
```

$$x,\ x+2+z$$

```
delete bl, y:
```

Example 3

With a third kind of block, suppressing evaluation may be limited to certain identifiers:

```
x:= 3: y:=1:
blockIdents({hold(x)})(x+y)
```

$$x + 1$$

```
delete x, y:
```

Example 4

A block may also contain a sequence; flattening is suppressed:

```
block((x, y),z)
```

$$x, y, z$$

Example 5

The content of a block may be extracted and evaluated using `unblock`:

```
y:= 1: unblock(block(y)); delete y:
```

$$1$$

Example 6

We want to create blocks that represent arithmetical expressions. To do this, we need our own block domain that overloads `testtype`:

```
domain myblock
  inherits block;
  category  Cat::BaseCategory;
  testtype:= (bl, T) -> if T = Type::Arithmetical or T = dom then
```

```
                          TRUE
                        else
                          block::testtype(bl, T)
                        end_if;
end_domain:
```

This allows us to make the number zero invisible for the evaluator by enclosing it into a block, but to retain the option to plug it into special functions:

```
f := sin(x+myblock(0))
```

$$\sin(x+0)$$

We can now manipulate this expression, without being disturbed by automatic simplification:

```
expand(f)
```

$$\cos(0)\sin(x)+\cos(x)\sin(0)$$

```
 eval(unblock(%))
```

$$\sin(x)$$

Parameters

a

Any object or sequence of objects

s

Set of identifiers

Return Values

`blockIdents` creates objects of its own type.

See Also

MuPAD Functions

`block` | `blockTransparent` | `freeze` | `hold` | `unblock`

blockTransparent

Create a transparent block protected against evaluation

Syntax

```
blockTransparent(a)
```

Description

`blockTransparent(a)` creates a transparent block, which is left unchanged by evaluation, too, but treated as expression with the same operands as `a`.

`blockTransparent` is a domain. If `d` is any block domain, `d(a)` creates a block belonging to that domain, with content `a`.

`blockTransparent(a)` puts its argument into a block, without evaluating it. In order to evaluate `a` normally before putting it into a block of some kind, use `eval(hold(blockTransparent)(a))`.

Transparent blocks are invariant under evaluation.

Transparent blocks, on the other hand, have the same operands as their content..

Sequences can also be put into block; in the case of `blockTransparent`, they are not flattened. See “Example 4” on page 1-366.

You can create further block domains using inheritance. This particularly useful for creating own `evaluate` or `testtype` methods. See “Example 6” on page 1-366.

The call `expr(b)` replaces all transparent blocks in `b` by their content, without evaluating that content. Thus, `expr(blockTransparent(a))` is similar to `hold(a)`.

In case of nested blocks, only the outermost block is removed by both `expr` and `unblock`.

Examples

Example 1

A block is a sort of container that protects its content against evaluation:

```
y:= 1: bl:= block(1+y)
```

$$1+y$$

Blocks are atomic; thus `y` and `1` are not visible as operands:

```
op(bl), nops(bl)
```

$$1+y,\ 1$$

Although blocks are not arithmetical expressions, some basic arithmetical operations are defined for them:

```
collect(x+bl+x*bl, bl);
delete y, bl:
```

$$(x+1)\,(1+y)+x$$

Example 2

Transparent blocks protect against evaluation, too:

```
y:= 2:
bl:=blockTransparent(x+y+z)
```

$$x+y+z$$

However, a transparent block allows access to the operands of its content:

```
op(bl,1), subs(bl, hold(y) = y)
```

$$x,\ x+2+z$$

```
delete bl, y:
```

Example 3

With a third kind of block, suppressing evaluation may be limited to certain identifiers:

```
x:= 3: y:=1:
blockIdents({hold(x)})(x+y)
```

$$x + 1$$

```
delete x, y:
```

Example 4

A block may also contain a sequence; flattening is suppressed:

```
block((x, y),z)
```

$$x, y, z$$

Example 5

The content of a block may be extracted and evaluated using `unblock`:

```
y:= 1: unblock(block(y)); delete y:
```

$$1$$

Example 6

We want to create blocks that represent arithmetical expressions. To do this, we need our own block domain that overloads `testtype`:

```
domain myblock
  inherits block;
  category  Cat::BaseCategory;
  testtype:= (bl, T) -> if T = Type::Arithmetical or T = dom then
                          TRUE
```

```
                        else
                          block::testtype(bl, T)
                        end_if;
end_domain:
```

This allows us to make the number zero invisible for the evaluator by enclosing it into a block, but to retain the option to plug it into special functions:

```
f := sin(x+myblock(0))
```

$$\sin(x + 0)$$

We can now manipulate this expression, without being disturbed by automatic simplification:

```
expand(f)
```

$$\cos(0)\,\sin(x) + \cos(x)\,\sin(0)$$

```
 eval(unblock(%))
```

$$\sin(x)$$

Parameters

a

Any object or sequence of objects

Return Values

`blockTransparent` creates objects of its own type.

See Also

MuPAD Functions

`block` | `blockIdents` | `freeze` | `hold` | `unblock`

unblock

Replace blocks by their contents

Syntax

```
unblock(b, <blockdomain, <Recurse>>)
```

Description

`unblock(b)` replaces all blocks that appear as subexpressions in `b` by their contents.

`unblock(b)` replaces all blocks in `b` by the result of evaluating their content. Thus, `unblock(block(a))` should in most cases be equivalent to `a`. The behavior of `unblock` may be controlled by additional arguments. If a second argument `blockdomain` is given, only blocks belonging to a domain that inherits from `blockdomain` are replaced by their content. If `FALSE` is provided as a third argument, only `b` is replaced by its content if it is a block of suitable type itself.

The call `expr(b)` replaces all transparent blocks in `b` by their content, without evaluating that content. Thus, `expr(blockTransparent(a))` is similar to `hold(a)`.

In case of nested blocks, only the outermost block is removed by both `expr` and `unblock`.

Examples

Example 1

A block is a sort of container that protects its content against evaluation:

```
y:= 1: bl:= block(1+y)
```

$$1+y$$

Blocks are atomic; thus `y` and `1` are not visible as operands:

```
op(bl), nops(bl)
```

$$1+y,\ 1$$

Although blocks are not arithmetical expressions, some basic arithmetical operations are defined for them:

```
collect(x+bl+x*bl, bl);
delete y, bl:
```

$$(x+1)\,(1+y)+x$$

Example 2

Transparent blocks protect against evaluation, too:

```
y:= 2:
bl:=blockTransparent(x+y+z)
```

$$x+y+z$$

However, a transparent block allows access to the operands of its content:

```
op(bl,1), subs(bl, hold(y) = y)
```

$$x,\ x+2+z$$

```
delete bl, y:
```

Example 3

With a third kind of block, suppressing evaluation may be limited to certain identifiers:

```
x:= 3: y:=1:
blockIdents({hold(x)})(x+y)
```

$$x+1$$

```
delete x, y:
```

Example 4

A block may also contain a sequence; flattening is suppressed:

```
block((x, y),z)
```

$$x, y, z$$

Example 5

The content of a block may be extracted and evaluated using `unblock`:

```
y:= 1: unblock(block(y)); delete y:
```

$$1$$

Example 6

We want to create blocks that represent arithmetical expressions. To do this, we need our own block domain that overloads `testtype`:

```
domain myblock
  inherits block;
  category  Cat::BaseCategory;
  testtype:= (bl, T) -> if T = Type::Arithmetical or T = dom then
                          TRUE
                        else
                          block::testtype(bl, T)
                        end_if;
end_domain:
```

This allows us to make the number zero invisible for the evaluator by enclosing it into a block, but to retain the option to plug it into special functions:

```
f := sin(x+myblock(0))
```

$$\sin(x+0)$$

We can now manipulate this expression, without being disturbed by automatic simplification:

```
expand(f)
```

$$\cos(0)\,\sin(x) + \cos(x)\,\sin(0)$$

```
 eval(unblock(%))
```

$$\sin(x)$$

Parameters

`b`

Any object

`blockdomain`

Any domain that inherits from `block`

`Recurse`

TRUE or FALSE

Return Values

In most cases, the object b.

See Also

MuPAD Functions

`block` | `blockIdents` | `blockTransparent` | `freeze` | `hold`

bool

Boolean evaluation

Syntax

```
bool(b)
```

Description

`bool(b)` evaluates the Boolean expression `b`.

The function `bool` serves for reducing Boolean expressions to one of the Boolean constants `TRUE`, `FALSE`, or `UNKNOWN`.

Boolean expressions are expressions that are composed of equalities, inequalities, elementhood relations, and these constants, combined via the logical operators `and`, `or`, `not`.

The function `bool` evaluates all equalities and inequalities inside a Boolean expression to either `TRUE` or `FALSE`. The resulting logical combination of the Boolean constants is reduced according to the rules of the MuPAD three state logic (see `and`, `or`, `not`).

Note: Equations `x = y` and inequalities `x <> y` are evaluated *syntactically* by `bool`. It does not test equality in any mathematical sense.

Note: Inequalities `x < y`, `x <= y` etc. can be evaluated by `bool` if and only if `x` and `y` are real numbers of type `Type::Real`. Otherwise, an error occurs.

`bool` evaluates *all* subexpressions of a Boolean expression before simplifying the result. The functions `_lazy_and`, `_lazy_or` provide an alternative: “lazy Boolean evaluation”.

There is no need to use `bool` in the conditional part of `if`, `repeat`, and `while` statements. Internally, these statements enforce Boolean evaluation by `_lazy_and` and `_lazy_or`. Cf. “Example 5” on page 1-375.

Use `simplify` with the option `logic` to simplify expressions involving symbolic Boolean subexpressions. Cf. “Example 7” on page 1-376.

`bool` is overloadable not only for domains, but also for function environments. This means that, if `f` evaluates to a function environment, then `bool(f(x1, …, xn))` returns `f::bool( x1, …, xn )`, or an error if no slot `f::bool` exists.

The call `bool(x ~= y)` serves for comparing numerical values `x` and `y`. If both `x` and `y` can be converted to non-zero real or complex floating-point numbers, it is checked whether `float((x - y)/x)| < 10^(-DIGITS)` is satisfied. Thus, `TRUE` is returned if `x` and `y` coincide within the relative numerical precision set by `DIGITS`. For `x = 0`, the criterion is `|float(y)| < 10^(-DIGITS)`. For `y = 0`, the criterion is `|float(x)| < 10^(-DIGITS)`. If either `x` or `y` contains a symbolic object that cannot be converted to a real or complex floating point number, the function `bool` returns the value `UNKNOWN`.

Examples

Example 1

MuPAD realizes that 1 is less than 2:

```
1 < 2 = bool(1 < 2)
```

$$1 < 2 = \text{TRUE}$$

Note that `bool` can fail to compare real numbers expressed symbolically:

```
bool(sqrt(14) <= sqrt(2)*sqrt(7))
```

```
Error: Cannot evaluate to Boolean. [_leequal]
```

You can compare floating-point approximations. Alternatively, you can use `is`:

```
bool(float(sqrt(14)) <= float(sqrt(2)*sqrt(7))),
is(sqrt(14) <= sqrt(2)*sqrt(7))
```

$$\text{TRUE}, \text{TRUE}$$

Example 2

The Boolean operators `and`, `or`, `not` do not evaluate equations and inequalities logically, and return a symbolic Boolean expression. Boolean evaluation and simplification is enforced by `bool`:

```
a = a and 3 < 4
```

$$a = a \wedge 3 < 4$$

```
bool(a = a and 3 < 4)
```

TRUE

Example 3

`bool` handles the special Boolean constant `UNKNOWN`:

```
bool(UNKNOWN and 1 < 2), bool(UNKNOWN or 1 < 2),
bool(UNKNOWN and 1 > 2), bool(UNKNOWN or 1 > 2)
```

UNKNOWN, TRUE, FALSE, UNKNOWN

Example 4

`bool` must be able to reduce all parts of a composite Boolean expression to one of the Boolean constants. No symbolic Boolean subexpressions may be involved:

```
b := b1 and b2 or b3: bool(b)
```

```
Error: Cannot evaluate to Boolean. [bool]
```

```
b1 := 1 < 2: b2 := x = x: b3 := FALSE: bool(b)
```

TRUE

```
delete b, b1, b2, b3:
```

Example 5

There is no need to use `bool` explicitly in the conditional parts of `if`, `repeat`, and `while` statements. Note, however, that these structures internally use "lazy evaluation" via `_lazy_and` and `_lazy_or` rather than "complete Boolean evaluation" via `bool`:

```
x := 0: if x <> 0 and sin(1/x) = 0 then 1 else 2 end
```

```
    2
```

In contrast to "lazy evaluation", `bool` evaluates *all* conditions. Consequently, a division by zero occurs in the evaluation of `sin(1/x) = 0`:

```
bool(x <> 0 and sin(1/x) = 0)
```

```
Error: Division by zero. [_invert]
```

```
delete x:
```

Example 6

Note that `bool` does not operate recursively. The following calls are completely different, the first one comparing the expression `TRUE = TRUE` and the constant `TRUE` (syntactically), the second one comparing the result of another `bool`-call with `TRUE`:

```
bool((TRUE = TRUE) = TRUE);
bool(bool(TRUE = TRUE) = TRUE)
```

```
    FALSE
```

```
    TRUE
```

Since `if`, `while` and similar constructs use the same Boolean evaluation internally, this also effects conditions in such clauses:

```
if (is(a < b) = TRUE) or (3 = 3) then YES else NO end;
if (is(a < b) or (3 = 3)) = TRUE then YES else NO end
```

```
    YES
```

NO

Example 7

Expressions involving symbolic Boolean subexpressions cannot be processed by `bool`. However, `simplify` with the option `logic` can be used for simplification:

```
(b1 and b2) or (b1 and (not b2)) and (1 < 2)
```

$$(b1 \wedge b2) \vee (b1 \wedge \neg b2 \wedge 1 < 2)$$

```
simplify(%, logic)
```

b1

Parameters

b

A Boolean expression

Return Values

`TRUE`, `FALSE`, or `UNKNOWN`.

Overloaded By

b

See Also

MuPAD Functions

`_lazy_and` | `_lazy_or` | `FALSE` | `if` | `is` | `repeat` | `TRUE` | `UNKNOWN` | `while`

break, _break

Terminate a loop or a Case switch prematurely

Syntax

```
break
```

```
_break()
```

Description

`break` terminates `for`, `repeat`, `while` loops, and `case` statements.

The `break` statement is equivalent to the function call `_break()`. The return value is the void object of type `DOM_NULL`.

Inside `for`, `repeat`, `while`, and `case` statements, the `break` statement exits from the loop/switch. Execution proceeds with the next statement after the `end` clause of the loop/switch.

In nested loops, only the innermost loop is terminated by `break`.

`break` also terminates a statement sequence `_stmtseq(..., break, ...)`.

Outside `for`, `repeat`, `while`, `case`, and `_stmtseq`, the `break` statement has no effect.

Examples

Example 1

Loops are exited prematurely by `break`:

```
for i from 1 to 10 do
  print(i);
  if i = 2 then break end_if
end_for
```

```
    1

    2

delete i:
```

Example 2

In a `case` statement, all commands starting with the first matching branch are executed:

```
x := 2:
case x
 of 1 do print(1); x^2;
 of 2 do print(2); x^2;
 of 3 do print(3); x^2;
 otherwise print(UNKNOWN)
end_case:
```

```
    2

    3

    UNKNOWN
```

In the next version, `break` ensures that only the statements in the matching branch are evaluated:

```
case x
 of 1 do print(1); x^2; break;
 of 2 do print(2); x^2; break;
 of 3 do print(3); x^2; break;
 otherwise print(UNKNOWN)
end_case:
```

```
    2

delete x:
```

See Also

MuPAD Functions
`case` | `for` | `next` | `repeat` | `return` | `while`

buildnumber

The Build number of

Syntax

```
buildnumber()
```

Description

`buildnumber()` returns the "build number" of the MuPAD library.

`buildnumber` is a natural number increasing over time which enables the MuPAD developers to exactly identify the version of the library used. Its primary use is for cooperating partners with access to development versions.

Examples

Example 1

At the time of this writing, the MuPAD build number was 42703:

```
buildnumber()
```

```
42703
```

Return Values

Integer.

See Also

MuPAD Functions

`Pref::kernel | version`

bytes

Memory used by the current session

Syntax

```
bytes()
```

Description

`bytes()` returns the current memory consumption.

`bytes` returns the following three numbers:

- The number of bytes used logically; this is the amount of memory which is actually used for storing MuPAD data.
- The number of bytes physically allocated by the memory management; this is the amount of memory MuPAD has allocated from the operating system. The difference between the physical and the logical bytes is the amount of memory which has already been reserved for future calculations.
- The maximum number of bytes ever allocated from the operating system during the current session. This value *never* decreases, not even on a call to `reset`.

Examples

Example 1

In a freshly started MuPAD session, `bytes` may return the following data on the memory consumption of the session:

```
bytes()
```

```
837168, 1572864, 1703936
```

Each computation increases the memory usage:

```
sin(PI): bytes()

     970204, 1638400, 1703936

solve(x-1=0, x): bytes()

     2361864, 3014656, 3014656
```

Return Values

Sequence of three integers.

See Also

MuPAD Functions
`rtime` | `time`

card

Cardinality of a set

Syntax

```
card(set)
```

```
card(d)
```

Description

`card(set)` returns the cardinality of `set`.

If `set` is a `DOM_SET`, the number of operands is returned; `card` does not attempt to investigate whether the members of `set` really represent pairwise different mathematical objects.

`card` does not distinguish different infinite cardinals; it just returns `infinity` if `set` is infinite.

`card` returns a symbolic call to itself if it cannot determine the cardinality.

If applied to a domain `d`, `card` returns the domain entry `d::size`. A domain that does not have this entry is not regarded as a set.

Examples

Example 1

The cardinality of a finite set equals the number of its operands:

```
card({1, 2, 3})
```

```
3
```

This holds true even if there exist two operands of the set that represent the same mathematical object:

```
card({1, 1.0})
```

$$2$$

Example 2

`card` does not distinguish different sizes of infinite sets:

```
card(R_), card(Z_)
```

$$\infty, \infty$$

Example 3

Set-theoretic expressions containing symbols are legal input, but usually `card` will not be able to determine their cardinality:

```
card(S union {3})
```

$$|\{3\} \cup S|$$

Example 4

Domains that have a `"size"` entry are regarded as sets:

```
card(Dom::IntegerMod(7))
```

$$7$$

Parameters

`set`

A set of type `DOM_SET`, or a set-theoretic expression

d

A domain representing a set

Return Values

Nonnegative integer, or `infinity`.

Overloaded By

d, set

See Also

MuPAD Functions
nops

case, of, otherwise, end_case, _case

Switch statement

Syntax

```
case x
  of match1 do
    statements1
  of match2 do
    statements2
  ...
  otherwise
    otherstatements
end_case
```

```
case x
  of match1 do
    statements1
  of match2 do
    statements2
  ...
end_case
```

```
_case(x, match1, statements1, match2, statements2, , …, <otherstatements>)
```

Description

A `case-end_case` statement allows to switch between various branches in a program.

The `case` statement is a control structure that extends the functionality of the `if` statement. In a `case` statement, an object is compared with a number of given values and one or more statement sequences are executed.

If the value of `x` equals one of the values `match1`, `match2` etc., the first matching branch *and all its following branches (including `otherwise`)* are executed, until the execution is terminated by a `break` or a `return` statement, or the `end_case`.

If the value of `x` does not equal any of the values `match1, match2, ...`, only the `otherwise` branch is executed. If no `otherwise` branch exists, the `case` statement terminates and returns the void object of type `DOM_NULL`.

The keyword `end_case` may be replaced by the keyword `end`.

Examples

Example 1

All statements after the first match are executed:

```
x := 2:
case x
  of 1 do print(1)
  of 2 do print(4)
  of 3 do print(9)
  otherwise print("otherwise")
end_case:
```

```
    4


    9


    "otherwise"
```

`break` may be used to ensure that only one matching branch is executed:

```
case x
  of 1 do print(1); 1; break
  of 2 do print(4); 4; break
  of 3 do print(9); 9; break
  otherwise print("otherwise")
end_case:
```

```
    4
```

```
delete x:
```

Example 2

The functionality of the `case` statement allows to share code that is to be used in several branches. The following function uses the statement `print(x, "is a real number")` for the three branches that correspond to real MuPAD numbers:

```
isReal := proc(x)
begin
   case domtype(x)
     of DOM_INT do
     of DOM_RAT do
     of DOM_FLOAT do print(x, "is a real number"); break
     of DOM_COMPLEX do print(x, "is not a real number"); break
     otherwise print(x, "cannot decide");
   end_case
end_proc:
isReal(3), isReal(3/7), isReal(1.23), isReal(3 + I), isReal(z)
```

3, "is a real number"

$\frac{3}{7}$, "is a real number"

1.23, "is a real number"

$3 + i$, "is not a real number"

z, "cannot decide"

```
delete isReal:
```

Example 3

The correspondence between the functional and the imperative form of the `case` statement is demonstrated:

```
hold(_case(x, match1, (1; break), match2, (4; break),
           print("otherwise")))
```

```
case x
  of match1 do
    1;
    break
  of match2 do
    4;
    break
  otherwise
    print("otherwise")
end_case
```

```
hold(_case(x, match1, (1; break), match2, (4; break)))
```

```
case x
  of match1 do
    1;
    break
  of match2 do
    4;
    break
end_case
```

Parameters

`x, match1, match2, …`

Arbitrary MuPAD objects

`statements1, otherstatements, …`

Arbitrary sequences of statements

Return Values

Result of the last command executed inside the `case` statement. The void object of type `DOM_NULL` is returned if no matching branch was found and no `otherwise` branch exists. `NIL` is returned if a matching branch was encountered, but no command was executed inside this branch.

Algorithms

The functionality of the `case` statement corresponds to the `switch` statement of the C programming language.

See Also

MuPAD Functions
`break` | `if` | `return`

ceil

Rounding up to the next integer

Syntax

```
ceil(x)
```

Description

`ceil` rounds a number to the next larger integer.

For complex arguments, rounding is applied separately to the real and the imaginary parts.

Integers are returned for real numbers and exact expressions representing real numbers.

Unevaluated function calls are returned for arguments that contain symbolic identifiers.

For floating-point intervals, the result will be a floating-point interval containing all the results of applying the rounding function to the real or complex numbers inside the interval.

If you think of `x` as a floating-point number, then `trunc(x)` truncates the digits after the decimal point. Thus, `trunc` coincides with `floor` for real positive arguments and with `ceil` for real negative arguments, respectively.

Note: If the argument is a floating-point number of absolute value larger than 10^{DIGITS}, the resulting integer is affected by internal non-significant digits! Cf. “Example 3” on page 1-393.

Note: Internally, exact numerical expressions that are neither integers nor rational numbers are approximated by floating-point numbers before rounding. Thus, the resulting integer may depend on the present value of `DIGITS`! Cf. “Example 4” on page 1-394.

Environment Interactions

The functions are sensitive to the environment variable DIGITS which determines the numerical working precision.

Examples

Example 1

We demonstrate the rounding of real and complex numbers:

```
ceil(3.5), floor(3.5), round(3.5), trunc(3.5)
```

$$4,\ 3,\ 4,\ 3$$

```
ceil(-7/2), floor(-7/2), round(-7/2), trunc(-7/2)
```

$$-3,\ -4,\ -3,\ -3$$

```
ceil(3 + 5/2*I), floor(4.3 + 7*I), round(I/2), trunc(I/2)
```

$$3+3\,i,\ 4+7\,i,\ i,\ 0$$

Also symbolic expressions representing numbers can be rounded:

```
x := PI*I + 7*sin(exp(2)): ceil(x), floor(x), round(x), trunc(x)
```

$$7+4\,i,\ 6+3\,i,\ 6+3\,i,\ 6+3\,i$$

Rounding of expressions with symbolic identifiers produces unevaluated function calls:

```
delete x: ceil(x), floor(x - 1), round(x + 1), trunc(x^2 + 3)
```

$$\lceil x \rceil,\ \lfloor x-1 \rfloor,\ \mathrm{round}(x+1),\ \mathrm{trunc}(x^2+3)$$

Example 2

The call `round(x, n)` serves for rounding the `n`-th decimal digit of the floating-point representation of `x`:

```
round(123.456, 1), round(123.456, 2), round(123.456, 3),
round(123.456, 4), round(123.456, 5)
```

```
123.5, 123.46, 123.456, 123.456, 123.456
```

```
float(exp(5)*PI), round(exp(5)*PI, 3)
```

```
466.2536903, 466.254
```

The second argument may also be negative, leading to rounding of the digits to the left of the decimal point:

```
round(123.45, 1), round(123.45, 0), round(123.45, -1),
round(123.45, -2), round(123.45, -3)
```

```
123.4, 123.0, 120.0, 100.0, 0.0
```

Example 3

Care should be taken when rounding floating-point numbers of large absolute value:

```
x := 10^30/3.0
```

$3.333333333\ 10^{29}$

Note that only the first 10 decimal digits are “significant”. Further digits are subject to round-off effects caused by the internal binary representation. These “insignificant” digits are part of the integer produced by rounding:

```
floor(x), ceil(x)
```

```
333333333333333332997967970304, 333333333333333332997967970304
```

```
delete x:
```

Example 4

Exact numerical expressions are internally converted to floating point numbers before rounding. Consequently, the present setting of `DIGITS` can affect the result:

```
x := 10^30 - exp(30)^ln(10)
```

$$10000000000000000000000000000 - (e^{30})^{\ln(10)}$$

Note that the exact value of this number is 0. Floating point evaluation is subject to severe cancellations:

```
DIGITS := 10: float(x), floor(x), ceil(x)
```

$$-6.157265116\ 10^{13},\ 671088640,\ 671088640$$

The floating-point result is more accurate when a higher precision is used. The rounded values change accordingly:

```
DIGITS := 20: float(x), floor(x), ceil(x)
```

$$5504.0,\ 5504,\ 5504$$

```
DIGITS := 30: float(x), floor(x), ceil(x)
```

$$0.000000312924385070800781 25,\ 0,\ 1$$

```
delete x, DIGITS:
```

Example 5

On floating-point intervals, `ceil` and `floor` behave as expected:

```
ceil(3.5...6.7); floor(3.5...6.7)
```

$$4.0 \ldots 7.0$$

$$3.0 \ldots 6.0$$

These intervals, as easily seen, contain the results of `ceil(x)` and `floor(x)` for all $x \in 3.5 \ldots 6.7$, respectively.

Because there are finite numbers represented as `RD_INF` and `RD_NINF`, respectively, `ceil` and `floor` return very small or large representable numbers in certain cases:

```
ceil(RD_NINF...RD_NINF)
```

$$\text{RD\_NINF} \ldots -2.098578716\ 10^{323228496}$$

Parameters

x

An arithmetical expression or a floating-point interval

Return Values

Arithmetical expression.

Overloaded By

x

See Also

MuPAD Functions

`floor` | `frac` | `round` | `trunc`

floor

Rounding down to the next integer

Syntax

```
floor(x)
```

Description

`floor` rounds a number to the next smaller integer.

For complex arguments, rounding is applied separately to the real and the imaginary parts.

Integers are returned for real numbers and exact expressions representing real numbers.

Unevaluated function calls are returned for arguments that contain symbolic identifiers.

For floating-point intervals, the result will be a floating-point interval containing all the results of applying the rounding function to the real or complex numbers inside the interval.

If you think of `x` as a floating-point number, then `trunc(x)` truncates the digits after the decimal point. Thus, `trunc` coincides with `floor` for real positive arguments and with `ceil` for real negative arguments, respectively.

Note: If the argument is a floating-point number of absolute value larger than 10^{DIGITS}, the resulting integer is affected by internal non-significant digits! Cf. “Example 3” on page 1-398.

Note: Internally, exact numerical expressions that are neither integers nor rational numbers are approximated by floating-point numbers before rounding. Thus, the resulting integer may depend on the present value of `DIGITS`! Cf. “Example 4” on page 1-399.

Environment Interactions

The functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate the rounding of real and complex numbers:

```
ceil(3.5), floor(3.5), round(3.5), trunc(3.5)
```

$$4,\ 3,\ 4,\ 3$$

```
ceil(-7/2), floor(-7/2), round(-7/2), trunc(-7/2)
```

$$-3,\ -4,\ -3,\ -3$$

```
ceil(3 + 5/2*I), floor(4.3 + 7*I), round(I/2), trunc(I/2)
```

$$3+3\,i,\ 4+7\,i,\ i,\ 0$$

Also symbolic expressions representing numbers can be rounded:

```
x := PI*I + 7*sin(exp(2)): ceil(x), floor(x), round(x), trunc(x)
```

$$7+4\,i,\ 6+3\,i,\ 6+3\,i,\ 6+3\,i$$

Rounding of expressions with symbolic identifiers produces unevaluated function calls:

```
delete x: ceil(x), floor(x - 1), round(x + 1), trunc(x^2 + 3)
```

$$\lceil x \rceil,\ \lfloor x-1 \rfloor,\ \mathrm{round}(x+1),\ \mathrm{trunc}(x^2+3)$$

Example 2

The call `round(x, n)` serves for rounding the `n`-th decimal digit of the floating-point representation of `x`:

```
round(123.456, 1), round(123.456, 2), round(123.456, 3),
round(123.456, 4), round(123.456, 5)
```

```
123.5, 123.46, 123.456, 123.456, 123.456
```

```
float(exp(5)*PI), round(exp(5)*PI, 3)
```

```
466.2536903, 466.254
```

The second argument may also be negative, leading to rounding of the digits to the left of the decimal point:

```
round(123.45, 1), round(123.45, 0), round(123.45, -1),
round(123.45, -2), round(123.45, -3)
```

```
123.4, 123.0, 120.0, 100.0, 0.0
```

Example 3

Care should be taken when rounding floating-point numbers of large absolute value:

```
x := 10^30/3.0
```

$3.333333333\ 10^{29}$

Note that only the first 10 decimal digits are “significant”. Further digits are subject to round-off effects caused by the internal binary representation. These “insignificant” digits are part of the integer produced by rounding:

```
floor(x), ceil(x)
```

```
333333333333333332997967970304, 333333333333333332997967970304
```

```
delete x:
```

Example 4

Exact numerical expressions are internally converted to floating point numbers before rounding. Consequently, the present setting of `DIGITS` can affect the result:

```
x := 10^30 - exp(30)^ln(10)
```

$$10000000000000000000000000000 - (e^{30})^{\ln(10)}$$

Note that the exact value of this number is 0. Floating point evaluation is subject to severe cancellations:

```
DIGITS := 10: float(x), floor(x), ceil(x)
```

$$-6.157265116\ 10^{13},\ 671088640,\ 671088640$$

The floating-point result is more accurate when a higher precision is used. The rounded values change accordingly:

```
DIGITS := 20: float(x), floor(x), ceil(x)
```

$$5504.0,\ 5504,\ 5504$$

```
DIGITS := 30: float(x), floor(x), ceil(x)
```

$$0.000000312924385070800781 25,\ 0,\ 1$$

```
delete x, DIGITS:
```

Example 5

On floating-point intervals, `ceil` and `floor` behave as expected:

```
ceil(3.5...6.7); floor(3.5...6.7)
```

$$4.0 \ldots 7.0$$

$$3.0 \ldots 6.0$$

These intervals, as easily seen, contain the results of `ceil(x)` and `floor(x)` for all $x \in 3.5 \ldots 6.7$, respectively.

Because there are finite numbers represented as `RD_INF` and `RD_NINF`, respectively, `ceil` and `floor` return very small or large representable numbers in certain cases:

```
ceil(RD_NINF...RD_NINF)
```

$$\text{RD\_NINF} \ldots -2.098578716\ 10^{323228496}$$

Parameters

x

An arithmetical expression or a floating-point interval

Return Values

Arithmetical expression.

Overloaded By

x

See Also

MuPAD Functions

`ceil` | `frac` | `round` | `trunc`

round

Rounding to the nearest integer

Syntax

```
round(x, <n>)
```

Description

`round` rounds a number to the nearest integer.

For complex arguments, rounding is applied separately to the real and the imaginary parts.

For the call `round(x, n)`, the result is a floating-point number with the `n`-th decimal digit after the decimal point being rounded. All further digits are set to zero. If the integer `n` is negative, the corresponding digit to the left of the decimal point is rounded. Cf. “Example 2” on page 1-403.

Unevaluated function calls are returned for arguments that contain symbolic identifiers.

For floating-point intervals, the result will be a floating-point interval containing all the results of applying the rounding function to the real or complex numbers inside the interval.

Note: If the argument is a floating-point number of absolute value larger than 10^{DIGITS}, the resulting integer is affected by internal non-significant digits! Cf. “Example 3” on page 1-403.

Note: Internally, exact numerical expressions that are neither integers nor rational numbers are approximated by floating-point numbers before rounding. Thus, the resulting integer may depend on the present value of `DIGITS`! Cf. “Example 4” on page 1-404.

Environment Interactions

The functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate the rounding of real and complex numbers:

```
ceil(3.5), floor(3.5), round(3.5), trunc(3.5)
```

$$4,\ 3,\ 4,\ 3$$

```
ceil(-7/2), floor(-7/2), round(-7/2), trunc(-7/2)
```

$$-3,\ -4,\ -3,\ -3$$

```
ceil(3 + 5/2*I), floor(4.3 + 7*I), round(I/2), trunc(I/2)
```

$$3+3\,i,\ 4+7\,i,\ i,\ 0$$

Also symbolic expressions representing numbers can be rounded:

```
x := PI*I + 7*sin(exp(2)): ceil(x), floor(x), round(x), trunc(x)
```

$$7+4\,i,\ 6+3\,i,\ 6+3\,i,\ 6+3\,i$$

Rounding of expressions with symbolic identifiers produces unevaluated function calls:

```
delete x: ceil(x), floor(x - 1), round(x + 1), trunc(x^2 + 3)
```

$$\lceil x \rceil,\ \lfloor x-1 \rfloor,\ \mathrm{round}(x+1),\ \mathrm{trunc}(x^2+3)$$

Example 2

The call `round(x, n)` serves for rounding the `n`-th decimal digit of the floating-point representation of `x`:

```
round(123.456, 1), round(123.456, 2), round(123.456, 3),
round(123.456, 4), round(123.456, 5)
```

123.5, 123.46, 123.456, 123.456, 123.456

```
float(exp(5)*PI), round(exp(5)*PI, 3)
```

466.2536903, 466.254

The second argument may also be negative, leading to rounding of the digits to the left of the decimal point:

```
round(123.45, 1), round(123.45, 0), round(123.45, -1),
round(123.45, -2), round(123.45, -3)
```

123.4, 123.0, 120.0, 100.0, 0.0

Example 3

Care should be taken when rounding floating-point numbers of large absolute value:

```
x := 10^30/3.0
```

$3.333333333\ 10^{29}$

Note that only the first 10 decimal digits are “significant”. Further digits are subject to round-off effects caused by the internal binary representation. These “insignificant” digits are part of the integer produced by rounding:

```
floor(x), ceil(x)
```

333333333333333332997967970304, 333333333333333332997967970304

```
delete x:
```

Example 4

Exact numerical expressions are internally converted to floating point numbers before rounding. Consequently, the present setting of `DIGITS` can affect the result:

```
x := 10^30 - exp(30)^ln(10)
```

$$10000000000000000000000000000 - (e^{30})^{\ln(10)}$$

Note that the exact value of this number is 0. Floating point evaluation is subject to severe cancellations:

```
DIGITS := 10: float(x), floor(x), ceil(x)
```

$$-6.157265116\ 10^{13},\ 671088640,\ 671088640$$

The floating-point result is more accurate when a higher precision is used. The rounded values change accordingly:

```
DIGITS := 20: float(x), floor(x), ceil(x)
```

$$5504.0,\ 5504,\ 5504$$

```
DIGITS := 30: float(x), floor(x), ceil(x)
```

$$0.000000312924385070800781 25,\ 0,\ 1$$

```
delete x, DIGITS:
```

Example 5

On floating-point intervals, `ceil` and `floor` behave as expected:

```
ceil(3.5...6.7); floor(3.5...6.7)
```

$$4.0 \ldots 7.0$$

$$3.0 \ldots 6.0$$

These intervals, as easily seen, contain the results of `ceil(x)` and `floor(x)` for all $x \in 3.5 \ldots 6.7$, respectively.

Because there are finite numbers represented as `RD_INF` and `RD_NINF`, respectively, `ceil` and `floor` return very small or large representable numbers in certain cases:

```
ceil(RD_NINF...RD_NINF)
```

$$\text{RD\_NINF} \ldots -2.098578716\ 10^{323228496}$$

Parameters

x

An arithmetical expression or a floating-point interval

n

An integer. If `n` is positive, the `n`-th digit after the decimal point is rounded. If `n` is negative, the `|n|`-th digit before the decimal point is rounded.

Return Values

Arithmetical expression.

Overloaded By

x

See Also

MuPAD Functions

`ceil` | `floor` | `frac` | `trunc`

trunc

Rounding towards zero

Syntax

```
trunc(x)
```

Description

`trunc` rounds a number to the next integer in the direction of 0.

For complex arguments, rounding is applied separately to the real and the imaginary parts.

Integers are returned for real numbers and exact expressions representing real numbers.

Unevaluated function calls are returned for arguments that contain symbolic identifiers.

For floating-point intervals, the result will be a floating-point interval containing all the results of applying the rounding function to the real or complex numbers inside the interval.

If you think of `x` as a floating-point number, then `trunc(x)` truncates the digits after the decimal point. Thus, `trunc` coincides with `floor` for real positive arguments and with `ceil` for real negative arguments, respectively.

Note: If the argument is a floating-point number of absolute value larger than 10^{DIGITS}, the resulting integer is affected by internal non-significant digits! Cf. “Example 3” on page 1-408.

Note: Internally, exact numerical expressions that are neither integers nor rational numbers are approximated by floating-point numbers before rounding. Thus, the resulting integer may depend on the present value of `DIGITS`! Cf. “Example 4” on page 1-409.

Environment Interactions

The functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate the rounding of real and complex numbers:

```
ceil(3.5), floor(3.5), round(3.5), trunc(3.5)
```

$$4,\ 3,\ 4,\ 3$$

```
ceil(-7/2), floor(-7/2), round(-7/2), trunc(-7/2)
```

$$-3,\ -4,\ -3,\ -3$$

```
ceil(3 + 5/2*I), floor(4.3 + 7*I), round(I/2), trunc(I/2)
```

$$3+3\,i,\ 4+7\,i,\ i,\ 0$$

Also symbolic expressions representing numbers can be rounded:

```
x := PI*I + 7*sin(exp(2)): ceil(x), floor(x), round(x), trunc(x)
```

$$7+4\,i,\ 6+3\,i,\ 6+3\,i,\ 6+3\,i$$

Rounding of expressions with symbolic identifiers produces unevaluated function calls:

```
delete x: ceil(x), floor(x - 1), round(x + 1), trunc(x^2 + 3)
```

$$\lceil x \rceil,\ \lfloor x-1 \rfloor,\ \mathrm{round}(x+1),\ \mathrm{trunc}(x^2+3)$$

Example 2

The call `round(x, n)` serves for rounding the `n`-th decimal digit of the floating-point representation of `x`:

```
round(123.456, 1), round(123.456, 2), round(123.456, 3),
round(123.456, 4), round(123.456, 5)
```

123.5, 123.46, 123.456, 123.456, 123.456

```
float(exp(5)*PI), round(exp(5)*PI, 3)
```

466.2536903, 466.254

The second argument may also be negative, leading to rounding of the digits to the left of the decimal point:

```
round(123.45, 1), round(123.45, 0), round(123.45, -1),
round(123.45, -2), round(123.45, -3)
```

123.4, 123.0, 120.0, 100.0, 0.0

Example 3

Care should be taken when rounding floating-point numbers of large absolute value:

```
x := 10^30/3.0
```

$3.333333333\ 10^{29}$

Note that only the first 10 decimal digits are “significant”. Further digits are subject to round-off effects caused by the internal binary representation. These “insignificant” digits are part of the integer produced by rounding:

```
floor(x), ceil(x)
```

333333333333333332997967970304, 333333333333333332997967970304

```
delete x:
```

Example 4

Exact numerical expressions are internally converted to floating point numbers before rounding. Consequently, the present setting of `DIGITS` can affect the result:

```
x := 10^30 - exp(30)^ln(10)
```

$$10000000000000000000000000000 - (e^{30})^{\ln(10)}$$

Note that the exact value of this number is 0. Floating point evaluation is subject to severe cancellations:

```
DIGITS := 10: float(x), floor(x), ceil(x)
```

$$-6.157265116\ 10^{13},\ 671088640,\ 671088640$$

The floating-point result is more accurate when a higher precision is used. The rounded values change accordingly:

```
DIGITS := 20: float(x), floor(x), ceil(x)
```

$$5504.0,\ 5504,\ 5504$$

```
DIGITS := 30: float(x), floor(x), ceil(x)
```

$$0.000000312924385070800781 25,\ 0,\ 1$$

```
delete x, DIGITS:
```

Example 5

On floating-point intervals, `ceil` and `floor` behave as expected:

```
ceil(3.5...6.7); floor(3.5...6.7)
```

$$4.0 \ldots 7.0$$

$$3.0 \ldots 6.0$$

These intervals, as easily seen, contain the results of `ceil(x)` and `floor(x)` for all $x \in 3.5 \ldots 6.7$, respectively.

Because there are finite numbers represented as `RD_INF` and `RD_NINF`, respectively, `ceil` and `floor` return very small or large representable numbers in certain cases:

```
ceil(RD_NINF...RD_NINF)
```

$$\text{RD\_NINF} \ldots -2.098578716\ 10^{323228496}$$

Parameters

x

An arithmetical expression or a floating-point interval

Return Values

Arithmetical expression.

Overloaded By

x

See Also

MuPAD Functions

`ceil` | `floor` | `frac` | `round`

Ci

Cosine integral function

Syntax

```
Ci(x)
```

Description

`Ci(x)` represents the cosine integral $\mathrm{EULER} + \ln(x) + \int_0^x \frac{\cos(t)-1}{t}\,dt$.

If `x` is a floating-point number, then `Ci(x)` returns numerical values. The special values $Ci(\infty) = 0$ and $Ci(-\infty) = i\pi$ are implemented. For all other arguments, `Ci` returns symbolic function calls.

The float attribute of `Ci` is a kernel function, i.e., floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
Ci(1), Ci(sqrt(2)), Ci(x + 1), Ci(infinity), Ci(-infinity)
```

$$\mathrm{Ci}(1),\ \mathrm{Ci}(\sqrt{2}),\ \mathrm{Ci}(x+1),\ 0,\ \pi\,i$$

```
Chi(1), Chi(sqrt(2)), Chi(x + 1), Chi(I*infinity), Chi(-I*infinity)
```

$$\mathrm{Chi}(1),\ \mathrm{Chi}(\sqrt{2}),\ \mathrm{Chi}(x+1),\ \frac{\pi\,i}{2},\ -\frac{\pi\,i}{2}$$

Floating point values are computed for floating-point arguments:

```
Ci(1.0), Ci(2.0 + 10.0*I)
```

$$0.3374039229,\ -242.5252694 - 1185.8387\,i$$

```
Chi(1.0), Chi(2.0 + 10.0*I)
```

$$0.837866941,\ -0.229320148 + 1.848954746\,i$$

Example 2

`Ci` and `Chi` are singular at the origin:

```
Ci(0)
```

```
Error: Singularity. [Ci]
```

```
Chi(0)
```

```
Error: Singularity. [Chi]
```

The negative real axis is a branch cut of `Ci` and `Chi`. A jump of height $2\,\pi\,i$ occurs when crossing this cut:

```
Ci(-1.0), Ci(-1.0 + 10^(-10)*I), Ci(-1.0 - 10^(-10)*I)
```

$$0.3374039229 + 3.141592654\,i,\ 0.3374039229 + 3.141592654\,i,\ 0.3374039229 - 3.141592654\,i$$

```
Chi(-1.0), Chi(-1.0 + 10^(-10)*I), Chi(-1.0 - 10^(-10)*I)
```

$$0.837866941 + 3.141592654\,i,\ 0.837866941 + 3.141592653\,i,\ 0.837866941 - 3.141592653\,i$$

Example 3

The functions `diff`, `float`, and `series` handle expressions involving `Ci` and `Chi`:

```
diff(Ci(x), x, x, x), float(ln(3 + Ci(sqrt(PI))))
```

$$\frac{2\cos(x)}{x^3} - \frac{\cos(x)}{x} + \frac{2\sin(x)}{x^2},\ 1.241299561$$

```
diff(Chi(x), x, x, x), float(ln(3 + Chi(sqrt(PI))))
```

$$\frac{\cosh(x)}{x} + \frac{2\cosh(x)}{x^3} - \frac{2\sinh(x)}{x^2},\ 1.618452185$$

```
series(Ci(x), x = 0)
```

$$\mathrm{EULER} + \ln(x) - \frac{x^2}{4} + \frac{x^4}{96} + O(x^6)$$

```
series(Chi(x), x = 0)
```

$$\mathrm{EULER} + \ln(x) + \frac{x^2}{4} + \frac{x^4}{96} + O(x^6)$$

```
series(Ci(x), x = infinity, 5)
```

$$\frac{\sin(x)}{x} - \frac{\cos(x)}{x^2} - \frac{2\sin(x)}{x^3} + \frac{6\cos(x)}{x^4} + \frac{24\sin(x)}{x^5} + O\left(\frac{1}{x^6}\right)$$

```
series(Chi(x), x = infinity, 3);
```

$$\frac{e^x}{2x} + \frac{e^x}{2x^2} + \frac{e^x}{x^3} + \frac{3e^x}{x^4} + O\left(\frac{e^x}{x^5}\right)$$

Parameters

x

An arithmetical expression

Return Values

Arithmetical expression.

Overloaded By

x

Algorithms

The functions `Ci(x)-ln(x)` and `Chi(x)-ln(x)` are entire functions. Thus, `Ci` and `Chi` have a logarithmic singularity at the origin and a branch cut along the negative real axis. The values on the negative real axis coincide with the limit “from above”:

$$\mathrm{Ci}(x) = \lim_{\varepsilon \to 0^+} \mathrm{Ci}(x + \varepsilon\, \mathrm{i}),\ \mathrm{Chi}(x) = \lim_{\varepsilon \to 0^+} \mathrm{Chi}(x + \varepsilon\, \mathrm{i})$$

for real $x < 0$.

`Ci` and `Chi` are related by $Ci(x) - ln(x) = Chi(\mathrm{i}\, x) - ln(\mathrm{i}\, x)$ for all x in the complex plane.

References

[1] Abramowitz, M. and I. Stegun, “Handbook of Mathematical Functions”, Dover Publications Inc., New York (1965).

See Also

MuPAD Functions

`Chi` | `cos` | `Ei` | `int` | `Shi` | `Si` | `Ssi`

Chi

Hyperbolic cosine integral function

Syntax

```
Chi(x)
```

Description

`Chi(x)` represents the hyperbolic cosine integral $\mathrm{EULER} + \ln(x) + \int_0^x \frac{\cosh(t)-1}{t}\,dt$.

If `x` is a floating-point number, then `Chi(x)` returns numerical values. The special values $Chi(\infty) = \infty$, $Chi(-\infty) = \infty + i\pi$, $\mathrm{Chi}(i\infty) = \frac{i\pi}{2}$, and $\mathrm{Chi}(-i\infty) = -\frac{i\pi}{2}$ are implemented. For all other arguments `Chi` returns symbolic function calls.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
Ci(1), Ci(sqrt(2)), Ci(x + 1), Ci(infinity), Ci(-infinity)
```

$$\mathrm{Ci}(1),\ \mathrm{Ci}(\sqrt{2}),\ \mathrm{Ci}(x+1),\ 0,\ \pi\, i$$

```
Chi(1), Chi(sqrt(2)), Chi(x + 1), Chi(I*infinity), Chi(-I*infinity)
```

$$\mathrm{Chi}(1),\ \mathrm{Chi}(\sqrt{2}),\ \mathrm{Chi}(x+1),\ \frac{\pi\,i}{2},\ -\frac{\pi\,i}{2}$$

Floating point values are computed for floating-point arguments:

```
Ci(1.0), Ci(2.0 + 10.0*I)
```

$$0.3374039229,\ -242.5252694 - 1185.8387\,i$$

```
Chi(1.0), Chi(2.0 + 10.0*I)
```

$$0.837866941,\ -0.229320148 + 1.848954746\,i$$

Example 2

`Ci` and `Chi` are singular at the origin:

```
Ci(0)
```

```
Error: Singularity. [Ci]
```

```
Chi(0)
```

```
Error: Singularity. [Chi]
```

The negative real axis is a branch cut of `Ci` and `Chi`. A jump of height $2\,\pi\,i$ occurs when crossing this cut:

```
Ci(-1.0), Ci(-1.0 + 10^(-10)*I), Ci(-1.0 - 10^(-10)*I)
```

$$0.3374039229 + 3.141592654\,i,\ 0.3374039229 + 3.141592654\,i,\ 0.3374039229 - 3.141592654\,i$$

```
Chi(-1.0), Chi(-1.0 + 10^(-10)*I), Chi(-1.0 - 10^(-10)*I)
```

$$0.837866941 + 3.141592654\,i,\ 0.837866941 + 3.141592653\,i,\ 0.837866941 - 3.141592653\,i$$

Example 3

The functions `diff`, `float`, and `series` handle expressions involving `Ci` and `Chi`:

```
diff(Ci(x), x, x, x), float(ln(3 + Ci(sqrt(PI))))
```

$$\frac{2\cos(x)}{x^3} - \frac{\cos(x)}{x} + \frac{2\sin(x)}{x^2},\ 1.241299561$$

```
diff(Chi(x), x, x, x), float(ln(3 + Chi(sqrt(PI))))
```

$$\frac{\cosh(x)}{x} + \frac{2\cosh(x)}{x^3} - \frac{2\sinh(x)}{x^2},\ 1.618452185$$

```
series(Ci(x), x = 0)
```

$$\mathrm{EULER} + \ln(x) - \frac{x^2}{4} + \frac{x^4}{96} + O(x^6)$$

```
series(Chi(x), x = 0)
```

$$\mathrm{EULER} + \ln(x) + \frac{x^2}{4} + \frac{x^4}{96} + O(x^6)$$

```
series(Ci(x), x = infinity, 5)
```

$$\frac{\sin(x)}{x} - \frac{\cos(x)}{x^2} - \frac{2\sin(x)}{x^3} + \frac{6\cos(x)}{x^4} + \frac{24\sin(x)}{x^5} + O\left(\frac{1}{x^6}\right)$$

```
series(Chi(x), x = infinity, 3);
```

$$\frac{e^x}{2x} + \frac{e^x}{2x^2} + \frac{e^x}{x^3} + \frac{3e^x}{x^4} + O\left(\frac{e^x}{x^5}\right)$$

Parameters

x

An arithmetical expression

Return Values

Arithmetical expression.

Overloaded By

x

Algorithms

The functions `Ci(x)-ln(x)` and `Chi(x)-ln(x)` are entire functions. Thus, `Ci` and `Chi` have a logarithmic singularity at the origin and a branch cut along the negative real axis. The values on the negative real axis coincide with the limit “from above”:

$$\mathrm{Ci}(x) = \lim_{\varepsilon \to 0^+} \mathrm{Ci}(x + \varepsilon\, \mathrm{i}),\ \mathrm{Chi}(x) = \lim_{\varepsilon \to 0^+} \mathrm{Chi}(x + \varepsilon\, \mathrm{i})$$

for real $x < 0$.

`Ci` and `Chi` are related by $Ci(x) - ln(x) = Chi(\mathrm{i}\, x) - ln(\mathrm{i}\, x)$ for all x in the complex plane.

References

[1] Abramowitz, M. and I. Stegun, “Handbook of Mathematical Functions”, Dover Publications Inc., New York (1965).

See Also

MuPAD Functions

`Ci` | `cos` | `Ei` | `int` | `Shi` | `Si` | `Ssi`

coeff

Coefficients of a polynomial

Syntax

```
coeff(p, <All>)

coeff(p, <x>, n, <All>)

coeff(p, <[x, …]>, [n, …], <All>)

coeff(f, <vars>, <All>)

coeff(f, <vars>, <x>, n, <All>)

coeff(f, <vars>, <[x, …]>, [n, …], <All>)
```

Description

`coeff(p)` returns a sequence of all nonzero coefficients of the polynomial `p`.

`coeff(p, x, n)` regards `p` as a univariate polynomial in `x` and returns the coefficient of the term `x^n`.

`coeff(p, [x,...], [n,...])` regards `p` as a multivariate polynomial in `x,...` and returns the coefficient of the term `x^n,...`.

If the first argument `f` is not element of a polynomial domain, then `coeff` converts the expression internally to a polynomial of type `DOM_POLY` via `poly(f)`. If a list of indeterminates is specified, the polynomial `poly(f, vars)` is considered.

Coefficients of polynomial expressions `f` are returned as arithmetical expressions.

There are various ways to call `coeff` with a polynomial `p` of type `DOM_POLY`:

- `coeff(p)` returns a sequence of all nonzero coefficients of `p`. They are ordered according to the lexicographical term ordering. The order in descending.

  The returned coefficients are elements of the coefficient ring of `p`.

- `coeff(p, x, n)` regards `p` as a univariate polynomial in the variable `x` and returns the coefficient of the term `x^n`.

  For univariate polynomials, the returned coefficients are elements of the coefficient ring of `p`.

  For multivariate polynomials, the coefficients are returned as polynomials of type `DOM_POLY` in the "remaining" variables.
- `coeff(p, n)` is equivalent to `coeff(p, x, n)`, where `x` is the "main variable" of `p`. This variable is the first element of the list of indeterminates `op(p, 2)`.
- `coeff(p, [x1,x2,...], [n1,n2,...])` regards `p` as a multivariate polynomial in the variables `x1,x2,...` and returns the coefficient of the term `x1^n1*x2^n2...`. Variable and exponent lists must have the same length.

  The returned coefficients are either elements of the coefficient ring of `p` or polynomials of type `DOM_POLY` in the "remaining" variables.
- `coeff(p, [n1,n2,...])` is equivalent to `coeff(p, [x1,x2,...], [n1,n2,...])`, where the variables `x1,x2,...` are the "main variables" of `p`, i.e., the leading elements of the list of indeterminates `op(p, 2)`.
- `coeff(p, All)` returns a sequence of coefficients of `p` including those equal to zero. The function returns the result in ascending lexicographical order. For univariate polynomial `p`, the call `coeff(p, All)` is equivalent to `coeff(p, i) $ i = 0 .. degree(p)`.

`coeff` returns 0 or a zero polynomial if the polynomial does not contain a term corresponding to the specified powers. In particular, this happens for a univariate polynomial if `n` is larger than the degree of the polynomial.

`coeff` returns `FAIL` if an expression cannot be regarded as a polynomial.

The result of `coeff` is not fully evaluated. Evaluation can be enforced by the function `eval`. See "Example 5" on page 1-423.

Examples

Example 1

`coeff(f)` returns a sequence of all non-zero coefficients:

```
f := 10*x^10 + 5*x^5 + 2*x^2: coeff(f)
```

$$10, 5, 2$$

`coeff(f, i)` returns a single coefficient:

```
coeff(f, i) $ i = 0..15
```

$$0, 0, 2, 0, 0, 5, 0, 0, 0, 0, 10, 0, 0, 0, 0, 0$$

```
delete f:
```

Example 2

We demonstrate how the indeterminates influence the result:

```
f := 3*x^3 + x^2*y^2 + 17*x + 23*y + 2
```

$$3\,x^3 + x^2\,y^2 + 17\,x + 23\,y + 2$$

```
coeff(f); coeff(f, [x, y]); coeff(f, [y, x])
```

$$3, 1, 17, 23, 2$$

$$3, 1, 17, 23, 2$$

$$1, 23, 3, 17, 2$$

```
delete f:
```

Example 3

The coefficients of `f` are selected with respect to the main variable `x` which is the first entry of the list of indeterminates:

```
f := 3*x^3 + x^2*y^2 + 2: coeff(f, [x, y], i) $ i = 0..3
```

$$2,\ 0,\ y^2,\ 3$$

The coefficients of `f` can be selected with respect to another main variable (in this case, `y`):

```
coeff(f, [y, x], i) $ i = 0..2
```

$$3\,x^3+2,\ 0,\ x^2$$

Alternatively:

```
coeff(f, y, i) $ i = 0..2
```

$$3\,x^3+2,\ 0,\ x^2$$

The coefficients of `f` can also be selected with respect to a multivariate term:

```
coeff(f, [x,y], [3,0]),
coeff(f, [x,y], [2,2]),
coeff(f, [x,y], [0,0])
```

$$3,\ 1,\ 2$$

```
delete f:
```

Example 4

In the same way, `coeff` can be applied to polynomials of type `DOM_POLY`:

```
p := poly(3*x^3 + x, [x], Dom::IntegerMod(7)):
coeff(p)
```

$$3 \bmod 7,\ 1 \bmod 7$$

```
coeff(p, i) $ i = 0..3
```

$$0 \bmod 7,\ 1 \bmod 7,\ 0 \bmod 7,\ 3 \bmod 7$$

For multivariate polynomials, the coefficients with respect to an indeterminate are polynomials in the other indeterminates:

```
p := poly(3*x^3 + x^2*y^2 + 2, [x, y]):
```

```
coeff(p, y, 0), coeff(p, y, 1), coeff(p, y, 2);
```

$$\mathrm{poly}(3x^3+2, [x]),\ \mathrm{poly}(0, [x]),\ \mathrm{poly}(x^2, [x])$$

```
coeff(p, x, 0), coeff(p, x, 1), coeff(p, x, 2)
```

$$\mathrm{poly}(2, [y]),\ \mathrm{poly}(0, [y]),\ \mathrm{poly}(y^2, [y])$$

Note that the indeterminates passed to `coeff` will be used, even if the polynomial provides different indeterminates :

```
coeff(p, z, 0), coeff(p, z, 1), coeff(p, z, 2)
```

$$\mathrm{poly}(3x^3+x^2y^2+2, [x, y]),\ \mathrm{poly}(0, [x, y]),\ \mathrm{poly}(0, [x, y])$$

```
delete p:
```

Example 5

The result of `coeff` is not fully evaluated:

```
p := poly(27*x^2 + a*x, [x]): a := 5:
coeff(p, x, 1), eval(coeff(p, x, 1))
```

$$a,\ 5$$

```
delete p, a:
```

Example 6

To return all coefficients of a polynomial, use the `All` option:

```
p := poly(a*x^3 + b*x^2 + c*x + d, [x, y]):
```

```
coefficients := coeff(p, All)
```

$$d, c, b, a$$

To revert the order of the resulting sequence, use the `revert` function. This function does not operate on sequences. To convert a sequence to a list, call `revert` for this list, and convert the result back to a sequence:

```
op(revert([coefficients]))
```

$$a, b, c, d$$

The `All` option also works for polynomial expressions:

```
p_expr := 2*x^5 + 5*x^2 + 10*x + 3:
coeff(p_expr, All)
```

$$3, 10, 5, 0, 0, 2$$

You can use the `coeff` function with the `All` option to compute scalar products of polynomials. For example, the following procedure computes a scalar product of two polynomials in an orthonormal basis. The `coeff` function extracts the coefficients of the polynomials and returns two lists of coefficients. The `zip` function multiplies the entries of these lists pairwise and returns a list. The `op` function accesses the entries of that list. Finally, the `_plus` function computes the sum of all products:

```
scalarProduct := proc(p, q)
  local lp, lq;
begin
  lp := [coeff(p, All)];
  lq := [coeff(conjugate(q), All)];
_plus(op(zip(lp, lq, _mult)));
end_proc:
```

The following polynomials are orthogonal:

```
scalarProduct(poly(x^2 + 2), poly(x^3 + 2*x^2 - 1))
```

$$0$$

Example 7

`coeff(p, All)` also works for multivariate polynomials and polynomial expressions:

```
p := poly(2*x^2*y + PI*x + y^2 - 2, [x, y]):
coeff(p, All)
```

$$-2,\ 0,\ 1,\ \pi,\ 0,\ 0,\ 0,\ 2,\ 0$$

For a multivariate polynomial or polynomial expression, the order in which `coeff` returns the coefficients is such that the coefficient of the exponent vector $[e_1, e_2, \ldots]$ appears at position $e_1 d_1 + e_2 d_2 + \ldots + 1$. For example, represent the coefficients returned for bivariate polynomial as a matrix:

```
A := matrix(degree(p, x) + 1, degree(p, y) + 1, [coeff(p, All)])
```

$$\begin{pmatrix} -2 & 0 & 1 \\ \pi & 0 & 0 \\ 0 & 2 & 0 \end{pmatrix}$$

Parameters

`p`

A polynomial of type `DOM_POLY`

`x`

An indeterminate

`n`

A power: a nonnegative integer

`f`

A polynomial expression

`vars`

A list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

Options

`All`

The `coeff` function with this option returns all the coefficients of a polynomial or a polynomial expression including those equal to zero. The function returns the result in ascending lexicographical order. See “Example 6” on page 1-423 and “Example 7” on page 1-425.

Return Values

One or more coefficients of the coefficient ring of the polynomial, or a polynomial, or `FAIL`.

Overloaded By

f, p

See Also

MuPAD Functions

`collect` | `content` | `degree` | `degreevec` | `ground` | `icontent` | `lcoeff` | `ldegree` | `lmonomial` | `lterm` | `monomials` | `nterms` | `nthcoeff` | `nthmonomial` | `nthterm` | `poly` | `poly2list` | `tcoeff`

coerce

Type conversion

Syntax

```
coerce(object, T)
```

Description

`coerce(object, T)` tries to convert `object` into an element of the domain `T`.

If this is not possible or not implemented, then `FAIL` is returned.

Domains usually implement the two methods `"convert"` and `"convert_to"` for conversion tasks.

`coerce` uses these methods in the following way: It first calls `T::convert(object)` to perform the conversion. If this call yields `FAIL`, then the result of the call `object::dom::convert_to(object, T)` is returned, which again may be the value `FAIL`.

To find out the possible conversions for the `object` or which conversions are provided by the domain `T`, please read the description of the method `"coerce"` or `"convert"`, respectively, that can be found on the help page of the domain `T`, and the description of the method `"convert_to"` on the help page of the domain of `object`.

Only few basic domains currently implement the methods `"convert"` and `"convert_to"`.

Use the function `expr` to convert an object into an element of a basic domain.

Note that often a conversion can also be achieved by a call of the constructor of the domain `T`. See “Example 3” on page 1-430.

Examples

Example 1

We start with the conversion of an array into a list of domain type DOM_LIST:

```
a := array(1..2, 1..3, [[1, 2, 3], [4, 5, 6]])
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

```
coerce(a, DOM_LIST)
```

[1, 2, 3, 4, 5, 6]

We convert the array into an hfarray of type DOM_HFARRAY:

```
coerce(a, DOM_HFARRAY)
```

$$\begin{pmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \end{pmatrix}$$

The conversion of an array into a polynomial is not implemented, and thus coerce returns FAIL:

```
coerce(a, DOM_POLY)
```

FAIL

One can convert a one- or two-dimensional array into a matrix, and vice versa. An example:

```
A := coerce(a, matrix); domtype(A)
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

Dom::Matrix()

The conversion of a matrix into a list is also possible. The result is then a list of inner lists, where the inner lists represent the rows of the matrix:

```
coerce(A, DOM_LIST)
```

$$[[1, 2, 3], [4, 5, 6]]$$

```
coerce([1, 2, 3, 2], DOM_SET)
```

$$\{1, 2, 3\}$$

Any MuPAD object can be converted into a string, such as the arithmetical expression `2*x + sin(x^2)`:

```
coerce(2*x + sin(x^2), DOM_STRING)
```

"2*x + sin(x^2)"

Example 2

The function `factor` computes a factorization of a polynomial expression and returns an object of the library domain `Factored`:

```
f := factor(x^2 + 2*x + 1);
domtype(f)
```

$$(x+1)^2$$

Factored

This domain implements the conversion routine `"convert_to"`, which we can call directly to convert the factorization into a list (see `factor` for details):

```
Factored::convert_to(f, DOM_LIST)
```

$$[1, x+1, 2]$$

However, it is more convenient to use `coerce`, which internally calls the slot routine `Factored::convert_to`:

```
coerce(f, DOM_LIST)
```

$$[1, x+1, 2]$$

Example 3

Note that often a conversion can also be achieved by a call of the constructor of a domain `T`. For example, the following call converts an array into a matrix of the domain type `Dom::Matrix(Dom::Rational)`:

```
a := array(1..2, 1..2, [[1, 2], [3, 4]]):
MatQ := Dom::Matrix(Dom::Rational):

MatQ(a)
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

The call `MatQ(a)` implies the call of the method `"new"` of the domain `MatQ`, which in fact calls the method `"convert"` of the domain `MatQ` to convert the array into a matrix.

Here, the same can be achieved with the use of `coerce`:

```
A := coerce(a, MatQ);
domtype(A)
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

$$\text{Dom::Matrix(Dom::Rational)}$$

Note that the constructor of a domain `T` is supposed to *create* objects, not to convert objects of other domains into the domain type `T`. The constructor often allows more than one argument which allows to implement various user-friendly ways to create the objects (e.g., see the several possibilities for creating matrices offered by `matrix`).

Parameters

`object`

Any object

`T`

Any domain

Return Values

Object of the domain `T`, or the value `FAIL`.

Overloaded By

T

See Also

MuPAD Functions
`domtype` | `expr` | `testtype` | `type`

collect

Collect terms with the same powers

Syntax

```
collect(p, x, <f>)
```

```
collect(p, [x1, x2, …], <f>)
```

Description

`collect(p, x)` groups terms with the same powers of `x` in an expression `p`.

`collect(p, [x1, x2, ...])` groups terms with the same powers of x_1, x_2, … in a multivariate expression `p`.

If you pass a function name `f` as a third argument to `collect`, the procedure collects the powers of x (x_1, x_2, … for multivariate expression). Then it applies the function `f` to the coefficients.

`collect(p, x)` presents `p` as a sum $\sum_{i=0}^{n} a_i x^i$. The coefficients a_i are not polynomials in x. These coefficients can contain some terms with x, for example, $sin(x)$ or e^x.

`collect` returns a modified copy of a polynomial. The function does not change the polynomial itself. See “Example 1” on page 1-433.

If `p` is a rational expression in `x`, `collect` handles the numerator and denominator separately.

If `p` is a multivariate expression, *collect*(p, [x_1, x_2, …]) returns an expression in the following form:

$$\sum_{i_1\, i_2 \ldots} \left(a_{i_1, i_2, \ldots} {x_1}^{i_1} {x_2}^{i_2} \ldots \right).$$

The coefficients $a_{i_1, i_2, \ldots}$ are not polynomials in x. These coefficients can contain some terms with x_1, x_2, …, for example, $\sin(x_1)\, e^{x_2}$.

If `p` is a rational expression in x_1, x_2, …, the `collect` command handles the numerator and denominator separately.

For polynomial expressions, `collect` internally calls two functions: `poly` and then `expr`. The function `poly` converts an expression `p` into a polynomial in the given unknowns. This function returns a polynomial with the terms collected by the same powers. Then `expr` converts this polynomial into a polynomial expression. See `poly` for more information and examples. When applied to a rational expression, `collect` handles the numerator and denominator separately.

You can use arbitrary expressions as indeterminates. See “Example 2” on page 1-434.

You can specify a function name instead of a variable. In this case, `collect` treats all calls of the function with different arguments as different variables. See “Example 4” on page 1-435.

`collect` does not recursively collect the operands of nonpolynomial subexpressions of `p`. See “Example 2” on page 1-434.

If `p` is not a polynomial expression, `collect` can return the unchanged expression `p`. See “Example 5” on page 1-436.

Examples

Example 1

You can define a polynomial expression `p` and collect terms with the same powers of `x` and `y`:

```
p := x*y + z*x*y + y*x^2 - z*y*x^2 + x + z*x;
collect(p, [x, y])
```

$$x + x\,y + x\,z + x^2\,y - x^2\,y\,z + x\,y\,z$$

$$(1 - z)\,x^2\,y + (z + 1)\,x\,y + (z + 1)\,x$$

`collect` does not modify the original expression:

```
p
```

$$x + x\,y + x\,z + x^2\,y - x^2\,y\,z + x\,y\,z$$

You can collect terms with same powers of `x`:

```
collect(p, [x])
```

$$(y - y\,z)\,x^2 + (y + z + y\,z + 1)\,x$$

If an expression contains only one indeterminate, you can omit the square brackets in the second argument of the function call:

```
collect(p, x)
```

$$(y - y\,z)\,x^2 + (y + z + y\,z + 1)\,x$$

To factor coefficients in a resulting expression, pass `factor` as a third argument to `collect`:

```
collect(p, x, factor)
```

$$-(y\,(z - 1))\,x^2 + ((z + 1)\,(y + 1))\,x$$

Example 2

`collect` does not modify nonpolynomial subexpressions even if they contain a given indeterminate. In particular, `collect` does not recursively handle the operands of a nonpolynomial subexpression:

```
collect(sin((x + 1)^2)*(x + 1) + 5*sin((x + 1)^2) + x, x)
```

$$(\sin((x + 1)^2) + 1)\,x + 6\,\sin((x + 1)^2)$$

`collect` accepts nonpolynomial subexpressions as indeterminates:

```
collect(sin((x + 1)^2)*(x + 1) + 5*sin((x + 1)^2) + x,
        sin((x + 1)^2))
```

$$(x+6)\,\sin((x+1)^2)+x$$

Example 3

`collect` normalizes a rational expression, and then handles the numerator and denominator separately:

```
collect(z/(x+y) + 3*z/(x+z), z)
```

$$\frac{z^2+(4\,x+3\,y)\,z}{(x+y)\,z+x^2+y\,x}$$

Example 4

If you specify the name of a function as an indeterminate, `collect` handles functions calls with different arguments as different indeterminates:

```
collect(a*f(1) + c*f(1) + f(2) + d*f(2), f)
```

$$(a+c)\,f(1)+(d+1)\,f(2)$$

```
collect(a*sin(x) + b*sin(x) + c*sin(y) + d*sin(y), sin)
```

$$(a+b)\,\sin(x)+(c+d)\,\sin(y)$$

```
p:= diff(besselJ(0, x), x $ 4);
collect(p, besselJ);
collect(p, besselJ, expand);
```

$$\frac{2\,\mathrm{J}_1(x)}{x^3}-\frac{\mathrm{J}_1(x)}{x}+\frac{\frac{\mathrm{J}_1(x)}{x^2}-\mathrm{J}_1(x)+\frac{\frac{\mathrm{J}_1(x)}{x}-\mathrm{J}_0(x)}{x}}{x}+\mathrm{J}_0(x)+\frac{2\left(\frac{\mathrm{J}_1(x)}{x}-\mathrm{J}_0(x)\right)}{x^2}$$

$$\left(1-\frac{3}{x^2}\right)\mathrm{J}_0(x)+\left(\frac{\frac{2}{x^2}-1}{x}-\frac{1}{x}+\frac{4}{x^3}\right)\mathrm{J}_1(x)$$

$$\left(1-\frac{3}{x^2}\right)\mathrm{J}_0(x)+\left(\frac{6}{x^3}-\frac{2}{x}\right)\mathrm{J}_1(x)$$

Example 5

If `p` is not a polynomial expression, `collect` can return the unchanged expression `p`:

```
p :=  y^2*sin(x) + y*sin(x) + y^2*cos(x) + y*cos(x);
collect(p, x)
```

$$y^2\cos(x)+y^2\sin(x)+y\cos(x)+y\sin(x)$$

$$y^2\cos(x)+y^2\sin(x)+y\cos(x)+y\sin(x)$$

The expression `p` is a polynomial expression in `y`. You can group the terms with the same powers in this variable:

```
collect(p, y)
```

$$(\cos(x)+\sin(x))\,y^2+(\cos(x)+\sin(x))\,y$$

Parameters

`p`

An arithmetical expression.

`x, x1, x2, …`

The indeterminates: typically, identifiers or indexed identifiers.

`f`

A function.

Return Values

arithmetical expression.

Overloaded By

p

See Also

MuPAD Functions

`coeff` | `combine` | `expand` | `factor` | `indets` | `normal` | `poly` | `rectform` | `rewrite` | `simplify`

More About

- “Manipulate Expressions”
- “Choose Simplification Functions”

combine

Combine terms of same algebraic structure

Syntax

```
combine(f, <IgnoreAnalyticConstraints>)
```

```
combine(f, target, <IgnoreAnalyticConstraints>)
```

```
combine(f, [target1, target2, …], <IgnoreAnalyticConstraints>)
```

Description

`combine(f)` rewrites products of powers in the expression `f` as a single power.

`combine(f, target)` combines several calls to the target function(s) in the expression `f` to a single call.

`combine(f)` applies these rewriting rules to products of powers occurring as subexpressions in an arithmetical expression `f`:

- $x^a x^b = x^{a+b}$,
- $x^b y^b = (xy)^b$,
- $(x^a)^b = x^{ab}$.

The last two rules are only valid under certain additional restrictions, such as when b is an integer. Except for the third rule, this behavior of `combine` is the inverse functionality of `expand`. See “Example 1” on page 1-441.

Note: In certain cases, the MuPAD internal simplifier automatically applies these rules in the reverse direction, and `combine` sometimes has no effect. See “Example 2” on page 1-442.

`combine(f, target)` applies rewriting rules applicable to the target function(s) to an arithmetical expression `f`. Some of the rules are only valid under certain additional restrictions. For most of the rules, `combine` implements the inverse functionality of `expand`. This list shows the rewriting rules for the targets.

- `target = arctan`:

  $$\arctan(x) + \arctan(y) = \arctan\left(\frac{x+y}{1-xy}\right)$$

  for `x` and `y`, such that $|xy| < 1$.

- `target = exp` (see “Example 4” on page 1-442):

  - $e^a e^b = e^{a+b}$,
  - $(e^a)^b = e^{ab}$,

    where valid, reacting to properties.

- `target = int` (see “Example 5” on page 1-443):

  - $a\int f(x)dx = \int af(x)dx.$
  - $\int f(x)dx + \int g(x)dx = \int f(x) + g(x)dx.$
  - $\int_a^b f(x)dx + \int_a^b g(x)dx = \int_a^b f(x) + g(x)dx.$
  - $\int_a^b f(x)dx + \int_a^b g(y)dy = \int_a^b f(y) + g(y)dy.$
  - $\int_a^b yf(x)dx + \int_a^b xg(y)dy = \int_a^b yf(c) + xf(c)dc.$

- `target = gamma` (see “Example 6” on page 1-443):

  - $a\,\Gamma(a) = \Gamma(a+1)$,

- $\frac{\Gamma(a+1)}{\Gamma(a)} = a$,
- $\Gamma(1-a)\,\Gamma(a) = \frac{\pi}{\sin(\pi\,a)}$,
- $\Gamma(-a)\,\Gamma(a) = -\frac{\pi}{a\,\sin(\pi\,a)}$,
- $\Gamma(a+n)\,\Gamma(a) = \Gamma(a)^2\left(\prod_{i=0}^{n-1}(a+i)\right)$,

  for positive integers `n`.

- `target = ln` (see “Example 7” on page 1-444):
  - $\ln(a) + \ln(b) = \ln(ab)$,
  - $b\,\ln(a) = \ln(a^b)$,

    if b is less than `N`. By default, `N = 1000`. You can change the number `N` using the `Pref::autoExpansionLimit` command. See “Example 8” on page 1-444.

  The rules do not hold for arbitrary complex values of `a`, `b`. Specify appropriate properties for `a` or `b` to enable these rewriting rules. These rules are only applied to *natural* logarithms.
- `target = sincos` (see “Example 3” on page 1-442):
  - $\sin(x)\,\sin(y) = \frac{\cos(x-y)}{2} - \frac{\cos(x+y)}{2}$,

    where similar rules apply to *sin*(*x*) *cos*(*y*) and *cos*(*x*) *cos*(*y*):
  - $A\,\cos(x) + B\,\sin(x) = A\,\sqrt{1+\frac{B^2}{A^2}}\,\cos\left(x+\arctan\left(-\frac{B}{A}\right)\right)$.
- `target = sinhcosh`:

- $$\sinh(x)\,\sinh(y) = \frac{\cosh(x+y)}{2} - \frac{\cosh(x-y)}{2},$$

  where similar rules apply to $sinh(x)\,cosh(y)$ and $cosh(x)\,cosh(y)$.
- These rules apply recursively to powers of *sinh* and *cosh* with positive integral exponents.

`combine` works recursively on the subexpressions of `f`.

If the second argument is a list of targets, then `combine` is applied to `f` subsequently for each of the targets in the list. See "Example 10" on page 1-446.

If `f` is an array, a list, or a set, `combine` is applied to all entries of `f`. See "Example 11" on page 1-446. If `f` is a polynomial or a series expansion, of type `Series::Puiseux` or `Series::gseries`, `combine` is applied to each coefficient. See "Example 12" on page 1-446.

Environment Interactions

`combine` reacts to properties of identifiers appearing in the input.

Examples

Example 1

Combine powers of the same base using `combine`.

```
combine(sin(x) + x*y*x^(exp(1)))
```

$$\sin(x) + x^{e+1}\,y$$

`combine` also combines powers with the same exponent in certain cases:

```
combine(sqrt(2)*sqrt(3))
```

$$\sqrt{6}$$

Example 2

In most cases, however, `combine` does not combine powers with the same exponent:

```
combine(y^5*x^5)
```

$$x^5\,y^5$$

Example 3

Rewrite products of sines and cosines as a sum of sines and cosines by setting the second argument to `sincos`:

```
combine(sin(a)*cos(b) + sin(b)^2, sincos)
```

$$\frac{\sin(a+b)}{2} - \frac{\cos(2\,b)}{2} + \frac{\sin(a-b)}{2} + \frac{1}{2}$$

Rewrite sums of sines and cosines by setting the second argument to `sincos`:

```
combine(cos(a) + sin(a), sincos)
```

$$\sqrt{2}\,\cos\left(a - \frac{\pi}{4}\right)$$

Powers of sines or cosines with negative integer exponents are not rewritten:

```
combine(sin(b)^(-2), sincos)
```

$$\frac{1}{\sin(b)^2}$$

Example 4

Combine terms with the exponential function by specifying the second argument as `exp`.

```
combine(exp(3)*exp(2), exp)
```

$$e^5$$

```
combine(exp(a)^2, exp)
```

$$e^{2\,a}$$

Example 5

Rewrite integrals by setting the second argument to `int`.

```
combine(int(f(x),x)+int(g(x),x),int)
```

$$\int (f(x)+g(x))\,dx$$

`combine` combines a constant term with the integral.

```
combine(a*int(f(x),x),int)
```

$$\int a\,f(x)\,dx$$

`combine` combines integrals with the same limits.

```
combine(int(f(x),x=a..b)+int(g(y),y=a..b),int)
```

$$\int_a^b (f(x)+g(x))\,dx$$

Example 6

Combine calls to `gamma` by specifying the target as `gamma`. The `combine` function simplifies quotients of gammas to rational expressions.

```
combine(gamma(n+3)*gamma(n+4/3) / gamma(n+1) / gamma(n+10/3), gamma)
```

$$\frac{n^2+3\,n+2}{n^2+\frac{11\,n}{3}+\frac{28}{9}}$$

Example 7

This example shows the application of rules for the logarithm, and their dependence on properties of the identifiers appearing in the input. In the complex plane, the logarithm of a product does not always equal the sum of the logarithms of its factors. For real positive numbers, however, this rule can apply.

Try to combine terms with calls to `ln` by specifying the target as `ln`.

```
combine(ln(a) + ln(b), ln)
```

$$\ln(a)+\ln(b)$$

`combine` does not combine the terms. Set the appropriate assumptions to combine the terms.

```
assume(a > 0): assume(b > 0):
combine(ln(a) + ln(b), ln)
```

$$\ln(a\,b)$$

```
unassume(a): unassume(b):
```

Example 8

If a and b are integer or rational numbers and b is less than `1000`, `combine` returns logarithms.

```
combine(3*ln(2), ln)
```

$$\ln(8)$$

If b is greater than or equal to `1000`, `combine` returns results as $b\,\ln(a)$:

```
combine(1234*ln(5), ln)
```

$$1234 \ln(5)$$

You can change the limit on the number b by using the `Pref::autoExpansionLimit` function. For example, when you use the default value `N = 1000`, `combine` returns the following result for this logarithm:

```
combine(12*ln(12), ln)
```

$$\ln(8916100448256)$$

If you set the value of `Pref::autoExpansionLimit` to `10`, `combine` returns this logarithm in its original form:

```
Pref::autoExpansionLimit(10):
combine(12*ln(12), ln)
```

$$12 \ln(12)$$

For further computations, restore the default value of `Pref::autoExpansionLimit`:

```
Pref::autoExpansionLimit(NIL):
```

Example 9

The `IgnoreAnalyticConstraints` option applies a set of purely algebraic simplifications including the equality of sum of logarithms and a logarithm of a product. Using the `IgnoreAnalyticConstraints` option, you get a simpler result, but one that might be incorrect for some of the values of `a`.

Combine logarithms using the `IgnoreAnalyticConstraints` option.

```
combine(ln(a^5) - ln(a^4), ln, IgnoreAnalyticConstraints)
```

$$\ln(a)$$

Without using this option, you get a mathematically correct, but long result:

```
combine(ln(a^5) - ln(a^4), ln)
```

$$\ln(a^5) - \ln(a^4)$$

Example 10

The second argument also can be a list of targets. Then the rewriting rules for each of the targets in the list are applied.

Rewrite `ln` and `sincos` terms in the expression.

```
combine(ln(2) + ln(3) + sin(a)*cos(a), [ln, sincos])
```

$$\frac{\sin(2\,a)}{2} + \ln(6)$$

Example 11

`combine` maps to sets:

```
combine({sqrt(2)*sqrt(5), sqrt(2)*sqrt(11)})
```

$$\{\sqrt{10}, \sqrt{22}\}$$

Example 12

`combine` maps to the coefficients of polynomials:

```
combine(poly(sin(x)*cos(x)*y, [y]), sincos)
```

$$\mathrm{poly}\left(\frac{\sin(2\,x)}{2}\,y, [y]\right)$$

However, it does not touch the polynomial's indeterminates:

```
combine(poly(sin(x)*cos(x)), sincos)
```

$$\mathrm{poly}(\cos(x)\,\sin(x), [\cos(x), \sin(x)])$$

Parameters

`f`

An arithmetical expression, an array, a list, a polynomial, or a set

`target`

One of the identifiers: `arctan`, `exp`, `int`, `gamma`, `ln`, `sincos`, or `sinhcosh`

Options

`IgnoreAnalyticConstraints`

Apply purely algebraic simplifications to an expression. For more information see the options for the `Simplify` command.

Return Values

Object of the same type as the input object `f`.

Overloaded By

f

Algorithms

Advanced users can extend the functionality of `combine` by implementing additional rewriting rules for other target functions. To extend functionality, define a new slot `target` of `combine`. To define a new slot, you need to first unprotect the identifier `combine` using unprotect. Afterwards, the command `combine(f, target)` leads to the call `combine::target(f)` of the corresponding slot routine.

By default, `combine` handles a subexpression `g(x1,x2,...)` of `f` by calling itself recursively for the operands `x1`, `x2`, etc. Users can change this behavior for their own

mathematical function given by a function environment `g` by implementing a `combine` slot of `g`. To handle the subexpression `g(x1,x2,...)`, `combine` then calls the slot routine `g::combine` with the argument sequence `x1,x2,...` of `g`.

See Also

MuPAD Functions

`denom` | `expand` | `factor` | `normal` | `numer` | `radsimp` | `rectform` | `rewrite` | `simplify`

Introduced in R2007b

complexInfinity

Complex infinity

Syntax

```
complexInfinity
```

Description

`complexInfinity` represents the only non-complex point of the one-point compactification of the complex numbers.

Mathematically, `complexInfinity` is the north pole of the Riemann sphere, with the unit circle as equator and the point 0 at the south pole.

With respect to arithmetic, `complexInfinity` behaves like "`1/0`". In particular, non-zero complex numbers may be multiplied or divided by `complexInfinity` or `1/ complexInfinity`. Adding `complexInfinity` to a finite number yields again `complexInfinity`.

With respect to arithmetical operations, `complexInfinity` is incompatible with the real `infinity`.

Examples

Example 1

`complexInfinity` can be used in arithmetical operations with complex numbers. The result in multiplications or divisions is either `complexInfinity`, `0`, or `undefined`:

```
3*complexInfinity, I*complexInfinity, 0*complexInfinity;
3/complexInfinity, I/complexInfinity, 0/complexInfinity;
complexInfinity/3, complexInfinity/I;
complexInfinity*complexInfinity, complexInfinity/complexInfinity;
```

```
complexInfinity, complexInfinity, undefined
```

```
0, 0, 0
```

```
complexInfinity, complexInfinity
```

```
complexInfinity, undefined
```

The result in additions is `undefined` if one of the operands is infinite, and `complexInfinity` otherwise:

```
complexInfinity + complexInfinity, infinity + complexInfinity;
3 + complexInfinity, I + complexInfinity, PI + complexInfinity
```

```
undefined, undefined
```

```
complexInfinity, complexInfinity, complexInfinity
```

Symbolic expressions in arithmetical operations involving `complexInfinity` are implicitly assumed to be different from both `0` and `complexInfinity`:

```
delete x:
x*complexInfinity, x/complexInfinity, complexInfinity/x,
x + complexInfinity
```

```
complexInfinity, 0, complexInfinity, complexInfinity
```

Algorithms

`complexInfinity` is the only element of the domain `stdlib::CInfinity`.

See Also

MuPAD Functions
`infinity`

conjugate

Complex conjugation

Syntax

```
conjugate(z)
```

```
conjugate(L)
```

Description

`conjugate(z)` computes the conjugate $\Re(z)$ - i $\Im(z)$ of a complex number $z = \Re(z) + i\,\Im(z)$.

For numbers of type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT`, or `DOM_COMPLEX`, the conjugate is computed directly and very efficiently.

`conjugate` can handle symbolic expressions. Properties of identifiers are taken into account (see `assume`). An identifier `z` without any property is assumed to be complex, and the symbolic call `conjugate(z)` is returned. See “Example 2” on page 1-452.

`conjugate` knows how to handle special mathematical functions, such as:

```
_mult   _plus   _power  abs     cos     cosh    cot
coth    csc     csch    erf     erfc    exp     gamma
igamma  sec     sech    sin     sinh    tan     tanh
```

See “Example 1” on page 1-452.

If `conjugate` does not know how to handle a special mathematical function, then a symbolic `conjugate` call is returned. See “Example 3” on page 1-452.

This function is automatically mapped to all entries of container objects such as arrays, lists, matrices, polynomials, sets, and tables.

Environment Interactions

`conjugate` is sensitive to properties of identifiers set via `assume`.

Examples

Example 1

`conjugate` knows how to handle sums, products, the exponential function and the sine function:

```
conjugate((1 + I)*exp(2 - 3*I))
```

$$e^{2+3\,i}\,(1-i)$$

```
delete z: conjugate(z + 2*sin(3 - 5*I))
```

$$2\,\sin(3+5\,i)+\overline{z}$$

Example 2

`conjugate` reacts to properties of identifiers:

```
delete x, y: assume(x, Type::Real):
conjugate(x), conjugate(y)
```

$$x,\ \overline{y}$$

Example 3

If the input contains a function that the system does not know, then a symbolic `conjugate` call is returned:

```
delete f, z: conjugate(f(z) + I)
```

$$\overline{f(z)}-i$$

Now suppose that `f` is some user-defined mathematical function, and that $f(\overline{z}) = \overline{f(z)}$ holds for all complex numbers z. To extend the functionality of `conjugate` to `f`, we embed it into a function environment and suitably define its `"conjugate"` slot:

```
f := funcenv(f):
f::conjugate := u -> f(conjugate(u)):
```

Now, whenever `conjugate` is called with an argument of the form `f(u)`, it calls `f::conjugate(u)`, which in turn returns `f(conjugate(u))`:

```
conjugate(f(z) + I), conjugate(f(I))
```

$$f(\overline{z}) - i,\ f(-i)$$

Parameters

z

An arithmetical expression

L

A container object: an array, an hfarray, a list, a matrix, a polynomial, a set, or a table.

Return Values

arithmetical expression or a container object containing such expressions

Overloaded By

z

Algorithms

If a subexpression of the form `f(u,..)` occurs in `z` and `f` is a function environment, then `conjugate` attempts to call the slot `"conjugate"` of `f` to determine the conjugate of `f(u,..)`. In this way, you can extend the functionality of `conjugate` to your own special mathematical functions.

The slot `"conjugate"` is called with the arguments `u,..` of `f`.

If `f` has no slot `"conjugate"`, then the subexpression `f(u,..)` is replaced by the symbolic call `conjugate(f(u...))` in the returned expression.

See “Example 3” on page 1-452.

Similarly, if an element `d` of a library domain`T` occurs as a subexpression of `z`, then `conjugate` attempts to call the slot `"conjugate"` of that domain with `d` as argument to compute the conjugate of `d`.

If `T` does not have a slot `"conjugate"`, then `d` is replaced by the symbolic call `conjugate(d)` in the returned expression.

See Also

MuPAD Functions

`abs` | `assume` | `Im` | `Re` | `rectform` | `sign`

contains

Test if an entry exists in a container

Syntax

```
contains(s, object)
```

```
contains(l, object, <i>)
```

```
contains(t, object)
```

Description

`contains(s, object)` tests if `object` is an element of the set `s`.

`contains(l, object)` returns the index of `object` in the list `l`.

`contains(t, object)` tests if the array, table, or domain `t` has an entry corresponding to the index `object`.

`contains` is a fast membership test for the MuPAD basic container data types. For lists and sets, `contains` searches the elements for the given object. However, for arrays, tables, and domains, `contains` searches the indices.

`contains` works syntactically, i.e., mathematically equivalent objects are considered to be equal only if they are syntactically identical. `contains` does *not* represent elementhood in the mathematical sense. See “Example 2” on page 1-456.

`contains` does not descend recursively into subexpressions; use `has` to achieve this. See “Example 3” on page 1-457.

`contains(s, object)` returns `TRUE` if `object` is an element of the set `s`. Otherwise, it returns `FALSE`.

`contains(l, object)` returns the position of `object` in the list `l` as a positive integer if `object` is an entry of `l`. Otherwise, the return value is `0`. If more than one entry of `l` is equal to `object`, then the index of the first occurrence is returned.

By passing a third argument `i` to `contains`, you can specify a position in the list where the search is to start. Then, entries with index less than `i` are not taken into account. If `i` is out of range, then the return value is `0`.

See “Example 4” on page 1-457 and “Example 5” on page 1-457.

`contains(t, object)` returns `TRUE` if the array, table, or domain `t` has an entry corresponding to the index `object`. Otherwise, it returns `FALSE`. Cf. “Example 6” on page 1-458.

Examples

Example 1

`contains` may be used to test if a set contains a given element:

```
contains({a, b, c}, a), contains({a, b, c}, 2)
```

```
TRUE, FALSE
```

Example 2

`contains` works syntactically, i.e., mathematically equivalent objects are considered to be equal only if they are syntactically identical. In this example `contains` returns `FALSE` since `y*(x + 1)` and `y*x + y` are different representations of the same mathematical expression:

```
contains({y*(x + 1)}, y*x + y)
```

```
FALSE
```

Elementhood in the mathematical sense is represented by the operator `in`:

```
simplify(y*x + y in {y*(x+1)}, condition)
```

```
TRUE
```

Example 3

`contains` does not descend recursively into the operands of its first argument. In the following example, `c` is not an element of the set, and therefore `FALSE` is returned:

```
contains({a, b, c + d}, c)
```

```
    FALSE
```

If you want to test whether a given expression is contained *somewhere inside* a complex expression, please use `has`:

```
has({a, b, c + d}, c)
```

```
    TRUE
```

Example 4

`contains` applied to a `list` returns the position of the specified object in the list:

```
contains([a, b, c], b)
```

```
    2
```

If the list does not contain the object, `0` is returned:

```
contains([a, b, c], d)
```

```
    0
```

Example 5

`contains` returns the position of the first occurrence of the given object in the list if it occurs more than once:

```
l := [a, b, a, b]: contains(l, b)
```

```
    2
```

A starting position for the search may be given as optional third argument:

```
contains(l, b, 1), contains(l, b, 2),
contains(l, b, 3), contains(l, b, 4)
```

```
2, 2, 4, 4
```

If the third argument is out of range, then the return value is `0`:

```
contains(l, b, -1), contains(l, b, 0), contains(l, b, 5)
```

```
0, 0, 0
```

Example 6

For tables, `contains` returns `TRUE` if the second argument is a valid index in the table. The entries stored in the table are not considered:

```
t := table(13 = value): contains(t, 13), contains(t, value)
```

```
TRUE, FALSE
```

Similarly, `contains` tests if an `array` has a value for a given index. The array `a` has a value corresponding to the index `(1, 1)`, but none for the index `(1, 2)`:

```
a := array(1..3, 1..2, (1, 1) = x, (2, 1) = PI):
contains(a, (1, 1)), contains(a, (1, 2))
```

```
TRUE, FALSE
```

`contains` is not intended for testing if an array contains a given value:

```
contains(a, PI)
```

```
Error: Index dimension mismatch. [array]
```

Even if the dimensions match, the index must not be out of range:

```
contains(a, (4, 4))
```

```
Error: The argument is invalid. [array]
```

Example 7

`contains` may be used to test, whether a domain has the specified `slot`:

```
T := newDomain("T"):  T::index := value:
contains(T, index), contains(T, value)
```

```
    FALSE, FALSE
```

There is no entry corresponding to the slot `index` in `T`. Please keep in mind that the syntax `T::index` is equivalent to `slot(T, "index" )`:

```
contains(T, "index")
```

```
    TRUE
```

Example 8

Users can overload`contains` for their own domains. For illustration, we create a new domain `T` and supply it with an `"contains"` slot, which tests is the set of entries of an element contains the given value `idx`:

```
T := newDomain("T"):
T::contains := (e, idx) -> contains({extop(e)}, idx):
```

If we now call `contains` with an object of domain type `T`, the slot routine `T::contains` is invoked:

```
e := new(T, 1, 2): contains(e, 2), contains(e, 3)
```

```
    TRUE, FALSE
```

Parameters

s

A set

`l`

A list

`t`

An array of type `DOM_ARRAY`, a table, or a domain

`object`

An arbitrary MuPAD object

`i`

An integer

Return Values

For sets, arrays, tables, or domains, `contains` returns one of the Boolean values `TRUE` or `FALSE`. For lists, the return value is a nonnegative integer.

Overloaded By

l, s, t

See Also

MuPAD Functions
`_index` | `has` | `in` | `op` | `slot`

content

Content of a polynomial

Syntax

```
content(p)
```

```
content(f, <vars>)
```

Description

`content(p)` computes the content of the polynomial `p` or polynomial expression, i.e., the greatest common divisor of its coefficients.

If `p` is the zero polynomial, then `content` returns 0.

If `p` is a non-zero polynomial with coefficient ring `IntMod(n)` and `n` is a prime number, then `content` returns 1. If `n` is not a prime number, an error message is issued.

If `p` is a polynomial with a library domain `R` as coefficient ring, the `gcd` of its coefficients is computed using the slot `gcd` of `R`. If no such slot exists, then `content` returns `FAIL`.

If `p` is a polynomial with coefficient ring `Expr`, then `content` does the following.

If all coefficients of `p` are either integers or rational numbers, `content(p)` is equivalent to `gcd(coeff(p))`, and the return value is a positive integer or rational number. See “Example 1” on page 1-462.

If at least one coefficient is a floating point number or a complex number and all other coefficients are numbers, then `content` returns 1. See “Example 2” on page 1-463.

If at least one coefficient is not a number and all coefficients of `p` can be converted into polynomials via `poly`, then `content(p)` is equivalent to `gcd(coeff(p))`. See “Example 3” on page 1-463.

Otherwise, `content` returns 1.

A polynomial expression `f` is converted into a polynomial with coefficient ring `Expr` via `p :=poly(f, vars)`, and then `content` is applied to `p`. See “Example 1” on page 1-462.

Use `icontent` for polynomials that are known to have integer or rational coefficients, since it is much faster than `content`.

Dividing the coefficients of `p` by its content gives its primitive part. This one can also be obtained directly using `polylib::primpart`.

Examples

Example 1

If `p` is a polynomial with integer or rational coefficients, the result is the same as for `icontent`:

```
content(poly(6*x^3*y + 3*x*y + 9*y, [x, y]))
```

```
    3
```

The following call, where the first argument is a polynomial expression and not a polynomial, is equivalent to the one above:

```
content(6*x^3*y + 3*x*y + 9*y, [x, y])
```

```
    3
```

If no list of indeterminates is specified, then `poly` converts the expression into a polynomial with respect to all occurring indeterminates, and we obtain yet another equivalent call:

```
content(6*x^3*y + 3*x*y + 9*y)
```

```
    3
```

Above, we considered the polynomial as a bivariate polynomial with integer coefficients. We can also consider the same expression as a univariate polynomial in `x`, whose

coefficients contain a parameter `y`. Then the coefficients and their gcd—the content—are polynomial expressions in `y`:

```
content(poly(6*x^3*y + 3*x*y + 9*y, [x]))
```

$$3\,y$$

Here is another example where the coefficients and the content are again polynomial expressions:

```
content(poly(4*x*y + 6*x^3 + 6*x*y^2 + 9*x^3*y, [x]))
```

$$3\,y+2$$

The following call is equivalent to the previous one:

```
content(4*x*y + 6*x^3 + 6*x*y^2 + 9*x^3*y, [x])
```

$$3\,y+2$$

Example 2

If a polynomial or polynomial expression has numeric coefficients and at least one floating-point number is among them, its content is 1:

```
content(2.0*x+2.0)
```

$$1$$

Example 3

If not all of the coefficients are numbers, the gcd of the coefficients is returned:

```
content(poly(x^2*y+x, [y]))
```

$$x$$

Parameters

p

A polynomial of type DOM_POLY

f

A polynomial expression

vars

A list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

Return Values

an object of the same type as the coefficients of the polynomial or the value FAIL.

Overloaded By

p

See Also

MuPAD Functions

coeff | factor | gcd | icontent | ifactor | igcd | ilcm | lcm | poly | polylib::primpart

context

Evaluate an object in the enclosing context

Syntax

```
context(object)
```

Description

Within a procedure, `context(object)` evaluates `object` in the context of the calling procedure.

Most MuPAD procedures evaluate their arguments before executing the body of the procedure. However, if the procedure is declared with option `hold`, then the arguments are passed to the procedure unevaluated. `context` serves to evaluate such arguments a posteriori from within the procedure.

Like most MuPAD procedures, `context` first evaluates its argument `object` as usual in the context of the current procedure. Then the result is evaluated again in the dynamical context that was valid before the current procedure was called. The enclosing context is either the interactive level or the procedure that called the current procedure.

`"func_call"`-methods of domains never evaluate their arguments, whether the option `hold` is used or not. See “Example 2” on page 1-467.

`context` is sensitive to the value of the environment variable `LEVEL`, which determines the maximal depth of the recursive process that replaces an identifier by its value during evaluation. The evaluation of the argument takes place with the value of `LEVEL` that is valid in the current procedure, which is `1` by default. The second evaluation uses the value of `LEVEL` that is valid in the enclosing context, which is usually `1` if the enclosing context is also a procedure, while it is `100` by default if the enclosing context is the interactive level. See “Example 3” on page 1-467.

Note: The function `context` must not be called at interactive level, and `context` calls must not be nested. Thus it is not possible to evaluate an object in higher levels of the dynamical call stack. See “Example 4” on page 1-468.

Environment Interactions

`context` is sensitive to the value of the environment variable `LEVEL`, which determines the maximal substitution depth for identifiers.

Examples

Example 1

We define a procedure `f` with option `hold`. If this procedure is called with an identifier as argument, such as `a` below, the `identifier` itself is the actual argument inside of `f`. `context` may be used to get the value of `a` in the outer context:

```
a := 2:
f := proc(i)
       option hold;
     begin
       print(i, context(i), i^2 + 2, context(i^2 + 2));
     end_proc:
f(a):
```

$$a,\ 2,\ a^2+2,\ 6$$

If a procedure with option `hold` is called from another procedure you will see strange effects if the procedure with option `hold` does not evaluate its formal parameters with `context`. Here, the value of the formal parameter `j` in `g` is the `variable` `i` which is defined in the context of procedure `f` and not its value `4`. When you want to access the value of this variable you have to use `context`, otherwise you see the output `DOM_VAR(0,2)` which is the variable `i` of `f` which has lost its scope:

```
f := proc()
       local i;
     begin
       i := 4:
       g(i);
     end_proc:
g := proc(j)
       option hold;
     begin
```

```
      print(j, eval(j), context(j));
      print(j + 1)
    end_proc:
f()
```

$$\mathrm{DOM\_VAR}(0, 2), \mathrm{DOM\_VAR}(0, 2), 4$$

$$\mathrm{DOM\_VAR}(0, 2) + 1$$

Example 2

The `"func_call"` method of a domain is implicitly declared with `option hold`. We define a `"func_call"` method for the domain `DOM_STRING` of MuPAD strings. The slot routine converts its remaining arguments into strings and appends them to the first argument, which coincides with the string that is the 0th operand of the function call:

```
unprotect(DOM_STRING):
DOM_STRING::func_call :=
  string -> _concat(string, map(args(2..args(0)), expr2text)):
a := 1:  "abc"(1, a, x)
```

"abc1ax"

You see that the identifier `a` was added to the string, and not its value `1`. Use `context` to access the value that `a` has before the `"func_call"` method is invoked:

```
DOM_STRING::func_call :=
  string -> _concat(string, map(context(args(2..args(0))),
                                expr2text)):
"abc"(1, a, x);
delete DOM_STRING::func_call:  protect(DOM_STRING, Error):
```

"abc11x"

Example 3

This example shows the influence of the environment variable `LEVEL` on the evaluation of `context` and the differences to the functions `eval` and `level`. `p` is a function with

option `hold`. `x` is a formal parameter of this procedure. When evaluating their arguments `context`, `eval` and `level` all replace `x` first by its value `a`. Then `eval` evaluates `a` in the current context with `LEVEL = 1` and yields the value `b`. `context` evaluates `a` in the enclosing context (which is the interactive level) with `LEVEL = 100` and yields `c`. `level` always returns the result of the first evaluation step, which is `a`.

When the `LEVEL` of the interactive level is `1`, `context` returns `b` like `eval` since the second evaluation is performed with `LEVEL = 1` like in `eval`.

The local variable `b` of `p` does not influence the evaluation in `context`, `eval` and `level` since it is only a locally declared variable of type `DOM_VAR` which has nothing to do with the identifier `b`, which is the value of `a`:

```
delete a, b, c:  a := b:  b := c:
p := proc(x)
       option hold;
       local b;
     begin
       b := 2;
       eval(x), context(x), level(x), level(x,2);
     end:
p(a);
LEVEL := 1: p(a);
delete LEVEL:
```

$$b, c, a, a$$

$$b, b, a, a$$

Example 4

The function `context` must not be called at interactive level:

```
context(x)
```

```
Error: The function call is not allowed on the interactive level. [context]
```

```
 Error: Function call not
allowed on interactive level. [context]
```

Parameters

`object`

Any MuPAD object

Return Values

Evaluated object.

See Also

MuPAD Domains
`DOM_PROC`

MuPAD Functions
`eval` | `freeze` | `hold` | `LEVEL` | `level` | `MAXLEVEL` | `proc`

contfrac

Domain of continued fractions

Syntax

```
contfrac(r, <n>)

contfrac(f, x, <m>)

contfrac(f, x = x0, <m>)
```

Description

`contfrac(r)` creates a continued fraction approximation of the real number `r`.

`contfrac(f, x = x0)` creates a continued fraction approximation of the expression `f` as a function of `x` around `x = x0`.

The continued fraction expansion `contfrac(r n )` of a real number or numerical expression `r` is an expansion of the form

$$a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ldots + a_{k-1} + \cfrac{1}{a_k + \ldots}}}}$$

where `a1` is the integer `floor(r)` and `a2`, `a3`, ... are positive integers.

The continued fraction is computed by `numlib::contfrac(r < n >)`; the expansion returned by `contfrac` is of domain type `numlib::contfrac`.

See the documentation of `numlib::contfrac` for further details.

A continued fraction expansion `contfrac(f, x = x0)` of a symbolic expression `f` in the indeterminate `x` is an expansion of the form

$$a_1 + \cfrac{(x-x_0)^{e_1}}{a_2 + \cfrac{(x-x_0)^{e_2}}{a_3 + \cfrac{(x-x_0)^{e_3}}{\ldots + a_{k-1} + \cfrac{(x-x_0)^{e_{k-1}}}{a_k + O\left((x-x_0)^{e_k}\right)}}}}$$

where

- $a_1, \ldots, a_k$ are arithmetical expressions not containing powers of $x - x_0$. The coefficients $a_2, \ldots, a_k$ are non-zero.
- e_1 is a rational number and $e_2, \ldots, e_k$ are positive rational numbers. If $a_1 \neq 0$, then e_1 is positive as well.

If $x_0 = \pm\infty$ or `x0 = complexInfinity`, the terms $(x - x_0)^{e_i}$ have to be replaced by $\frac{1}{x^{e_i}}$.

For symbolic expressions `f`, `contfrac(f, x = x0)` returns an expansion of domain type `contfrac`.

One may also call `contfrac(f)` without specifying an identifier `x`. In this case, `contfrac` extracts the indeterminates in `f` automatically via `indets`. `FAIL` is returned if more than one indeterminate is found.

If `m` is not specified, the default value `m = ORDER` is used.

`contfrac` uses the function `Series::Puiseux::contfrac` to compute the continued fraction in the symbolic case. If `f` is a rational function with respect to the expansion variable `x`, and the `truncation order' `m is not specified, then `contfrac` returns an exact continued fraction expansion of `f`. Cf. “Example 3” on page 1-473.

Environment Interactions

When called with an irrational numerical value `r`, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision. For symbolic expressions `f`, the function is sensitive to the environment variable `ORDER` which determines the number of terms in truncated series expansions.

Examples

Example 1

We compute some continued fraction expansions of real numbers:

```
contfrac(27/31), contfrac(PI, 5)
```

$$\frac{1}{1+\frac{1}{6+\frac{1}{1+\frac{1}{3+\ldots}}}},\ 3+\frac{1}{7+\frac{1}{15+\frac{1}{1+\ldots}}}$$

They can also be computed by direct calls to `numlib::contfrac`:

```
numlib::contfrac(27/31), numlib::contfrac(PI, 5)
```

$$\frac{1}{1+\frac{1}{6+\frac{1}{1+\frac{1}{3+\ldots}}}},\ 3+\frac{1}{7+\frac{1}{15+\frac{1}{1+\ldots}}}$$

Example 2

We compute symbolic continued fractions of functions:

```
contfrac(exp(x), x = 0), contfrac(exp(-3*x^2), x = 0)
```

$$1+\frac{x}{1+\frac{x}{-2+\frac{x}{-3+\frac{x}{2+\frac{x}{5+O(x)}}}}},\ 1+\frac{x^2}{-\frac{1}{3}+\frac{x^2}{-2+\frac{x^2}{1+O\left(x^2\right)}}}$$

If no expansion variable is specified, the symbolic expression to be expanded must be univariate:

```
contfrac(exp(x*y))
```

```
Error: The first argument must be a univariate expression. [contfrac::function]
```

Symbolic parameters are accepted if the expansion variable is specified:

```
contfrac(exp(x*y), x)
```

$$1+\cfrac{x}{y^{-1}+\cfrac{x}{-2+\cfrac{x}{-3\,y^{-1}+\cfrac{x}{2+\cfrac{x}{5\,y^{-1}+O(x)}}}}}$$

In the next call, we specify the expansion point `x = 1` and request a specific `number of terms' by the third argument:

```
contfrac(exp(x*y), x = 1, 3);
```

$$e^{y}+\cfrac{x-1}{y^{-1}\,e^{-y}+\cfrac{x-1}{-2\,e^{y}+O(x-1)}}$$

Example 3

For rational functions, exact representations are returned when no specific `number of terms' is requested. The method `"rational"` returns the rational expression equivalent to the continued fraction:

```
cf := contfrac((x - y)/(x^3 + y^3), x, 2):
cf, contfrac::rational(cf);
```

$$-y^{-2}+\cfrac{x}{y^{3}+O(x)},\ \frac{x-y}{y^{3}}$$

```
cf := contfrac((x - y)/(x^3 + y^3), x):
cf, contfrac::rational(cf);
```

$$-y^{-2}+\cfrac{x}{y^3+\cfrac{x^2}{-y^{-1}+\cfrac{x}{-y^2+\cfrac{x}{\frac{y^{-1}}{2}+\frac{x}{2\,y^2}}}}}, \quad \frac{x-y}{x^3+y^3}$$

Example 4

The coefficients and expansion terms of a continued fraction can be accessed by the functions `nthcoeff` and `nthterm`:

```
cf := contfrac(sin(1/x), x = infinity, 4)
```

$$\cfrac{x^{-1}}{1+\cfrac{x^{-2}}{6+O\left(x^{-2}\right)}}$$

```
nthcoeff(cf, 1), nthcoeff(cf, 2), nthcoeff(cf, 3), nthcoeff(cf, 4);
```

$$0,\ 1,\ 6,\ \text{FAIL}$$

```
nthterm(cf, 1), nthterm(cf, 2), nthterm(cf, 3)
```

$$\frac{1}{x},\ \frac{1}{x^2},\ \text{FAIL}$$

```
delete cf:
```

Example 5

We can compute a series expansion of a continued fraction via `series`:

```
cf := contfrac(sin(x)/(x - PI) - 1, x = PI)
```

$$-2+\cfrac{(\pi-x)^2}{6+\cfrac{(\pi-x)^2}{\frac{10}{3}+O\left((\pi-x)^2\right)}}$$

If no further arguments are given in `series`, the default expansion variable is `op`(cf, 3); the default expansion point is `op`(cf, 4):

```
op(cf, 3), op(cf, 4)
```

$$x,\ \pi$$

```
series(cf)
```

$$-2+\frac{(\pi-x)^2}{6}-\frac{(\pi-x)^4}{120}+O((\pi-x)^6)$$

Both the series variable as well as the expansion point may be passed explicitly to `series`.

```
series(cf, x = PI)
```

$$-2+\frac{(\pi-x)^2}{6}-\frac{(\pi-x)^4}{120}+O((\pi-x)^6)$$

However, the values must coincide with the values used to compute the continued fraction: In the following call, the default expansion point `x = 0` is used by `series`. This clashes with the expansion point `x = PI` of the continued fraction:

```
series(cf, x)
```

```
Error: The expansion point 'PI' of the continued fraction does not coincide with the re
```

```
delete cf:
```

Parameters

r

A real number or a numerical expression that can be converted to a real floating-point number

n

The number of significant decimal digits: a positive integer. The default value is `n` = `DIGITS`.

f

An arithmetical expression interpreted as a function of *x*

x

An identifier

x0

The expansion point: an arithmetical expression, $\pm\infty$ or `complexInfinity`. The default value is 0.

m

The `number of terms': a positive integer. The default value is `m` = `ORDER`.

Return Values

Call `contfrac(r n )` with a numerical value `r` returns an object of type `numlib::contfrac`. The call `contfrac(f, x = x0 m )` with a symbolic expression `f` returns an object of type `contfrac`. `FAIL` is returned if no series expansion of `f` around `x0` could be computed.

Methods

Mathematical Methods

`series` — Serie of a continued fraction

```
series(cf, <m>)
```

```
series(cf, <x>, <m>)
```

```
series(cf, <x = x0>, <m>)
```

If `x` is not specified, the default series variable is `op(cf, 3)`. If `x0` is not specified, the default expansion point is `op(cf, 4)`. If no `number of terms' `m` is specified, `m` = `ORDER` is used.

This method overloads the function `series`.

Access Methods

`op` — Operand of the continued fraction

```
op(cf, <n>)
```

See Also

MuPAD Functions

`numlib::contfrac` | `series`

copyClosure

Copies the lexical closure of a procedure

Syntax

```
copyClosure(f)
```

Description

`copyClosure(f)` copies the lexical closure of a procedure or procedure environment `f`.

Usually, when a procedure is copied, for example by assigning it to an identifier, the lexical closure of the procedure is not copied. Via the copied procedure one can change the lexical closure of the original procedure. Thus, the lexical closure of a procedure shows the so-called *reference effect*.

`copyClosure` may be used to copy the lexical closure of a procedure. Changes in the closure of the copy no longer affect the original procedure's closure.

Closures are implemented by procedure environments (kernel type `DOM_PROC_ENV`) in MuPAD. `copyClosure` works by copying all lexically enclosing procedure environments of a procedure.

`copyClosure` may also be used to copy a procedure environment and all its lexically enclosing environments only.

Examples

Example 1

Procedure closures show the reference effect: The procedure `f` generated by `gen` changes its closure via the variable `i`. A “normal” copy `g` of `f` changes the variable in the same closure, as is seen by repeatedly calling `f` versus `g`.

```
gen:= proc()
```

```
    option escape;
    local i;
begin
    i := 0;
    proc() begin i := i+1 end
end:

f := gen():
g := f:
f(), g(), f(), g()
```

$$1, 2, 3, 4$$

If one now generates `f` again by calling `gen`, but copies `g` by calling `copyClosure`, then `g` has its own closure and now longer changes the variable `i` in the closure of `f`.

```
f := gen():
g := copyClosure(f):
f(), g(), f(), g()
```

$$1, 1, 2, 2$$

Parameters

`f`

A procedure or procedure environment to be copied

Return Values

Copied procedure or procedure environment

See Also

MuPAD Functions

`_assign`

curl

Curl of a vector field

Syntax

```
curl(v, x)
curl(v, x, ogCoord, <c>)
```

Description

`curl(v, x)` computes the curl of the three-dimensional vector field $\vec{v}$ with respect to the three-dimensional vector $\vec{x}$ in Cartesian coordinates. This is the vector field

$$\mathrm{curl}(\vec{v}) = \begin{pmatrix} \frac{\partial}{\partial x_2} v_3 - \frac{\partial}{\partial x_3} v_2 \\ \frac{\partial}{\partial x_3} v_1 - \frac{\partial}{\partial x_1} v_3 \\ \frac{\partial}{\partial x_1} v_2 - \frac{\partial}{\partial x_2} v_1 \end{pmatrix}.$$

`curl(v, x, ogCoord)` computes the curl of `v` with respect to `x` in the orthogonally curvilinear coordinate system specified by `ogCoord`.

`ogCoord` can be the name of a three-dimensional orthogonal coordinate system predefined in the table `linalg::ogCoordTab`. See “Example 2” on page 1-481.

Alternatively, `ogCoord` can be a list of vector of algebraic expressions representing the scale factors of the coordinate system. See example “Example 3” on page 1-481. For details, see the description of the `Scales` option on the `linalg::ogCoordTab` page.

If `v` is a vector then the component ring of `v` must be a field (a domain of category `Cat::Field`) for which differentiation with respect to `x` is defined.

`curl` returns a vector of the domain `Dom::Matrix()`.

Examples

Example 1

Compute the curl of the vector field $\vec{v}(x, y, z) = (x\ y, 2\ y, z)$ in Cartesian coordinates:

```
delete x, y, z:
curl([x*y, 2*y, z], [x, y, z])
```

$$\begin{pmatrix} 0 \\ 0 \\ -x \end{pmatrix}$$

Example 2

Compute the curl of the vector field $\vec{v}(r, \phi, z) = (r, \cos(\phi), z)$, $(0 \le \phi < 2\,\pi)$ in cylindrical coordinates:

```
delete r, phi, z: V := matrix([r, cos(phi), z]):

curl(V, [r, phi, z], Cylindrical)
```

$$\begin{pmatrix} 0 \\ 0 \\ \frac{\cos(\mathrm{phi})}{r} \end{pmatrix}$$

The following relations between Cartesian and cylindrical coordinates hold:

$$x = r\cos(\phi),\ y = r\sin(\phi),\ z = z$$

.

Other predefined orthogonal coordinate systems can be found in the table `linalg::ogCoordTab`.

Example 3

Compute the curl of a vector field in spherical coordinates r, θ,ϕ given by

$$\vec{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} r \sin(\theta) \cos(\phi) \\ r \sin(\theta) \sin(\phi) \\ r \cos(\theta) \end{pmatrix}$$

with $0 \le \theta \le \pi$, $0 \le \phi < 2\,\pi$. The vectors

$$\vec{e}_r = \frac{\frac{\partial}{\partial r}\vec{x}}{\left|\frac{\partial}{\partial r}\vec{x}\right|} = \begin{pmatrix} \sin(\theta) \cos(\phi) \\ \sin(\theta) \sin(\phi) \\ \cos(\theta) \end{pmatrix}, \quad \vec{e}_\theta = \frac{\frac{\partial}{\partial \theta}\vec{x}}{\left|\frac{\partial}{\partial \theta}\vec{x}\right|} = \begin{pmatrix} \cos(\theta) \cos(\phi) \\ \cos(\theta) \sin(\phi) \\ -\sin(\theta) \end{pmatrix},$$

$$\vec{e}_\phi = \frac{\frac{\partial}{\partial \phi}\vec{x}}{\left|\frac{\partial}{\partial \phi}\vec{x}\right|} = \begin{pmatrix} -\sin(\phi) \\ \cos(\phi) \\ 0 \end{pmatrix}$$

form an orthogonal system of unit vectors corresponding to the spherical coordinates. The scaling factors of the coordinate transformation (see `linalg::ogCoordTab`) are $\left|\frac{\partial}{\partial r}\vec{x}\right| = 1$, $\left|\frac{\partial}{\partial \theta}\vec{x}\right| = r$, $\left|\frac{\partial}{\partial \phi}\vec{x}\right| = r \sin(\theta)$, which we use in the following example to compute the curl of the vector field $\vec{v}(r, \theta, \phi) = r^2\,\vec{e}_\phi$:

```
delete r, Theta, phi:
curl([0, 0, r^2], [r, Theta, phi], [1, r, r*sin(Theta)])
```

$$\begin{pmatrix} \frac{r \cos(\mathrm{Theta})}{\sin(\mathrm{Theta})} \\ -3\,r \\ 0 \end{pmatrix}$$

These are the coefficients of the curl of $\vec{v}$ in the bases given by the vectors $\vec{e}_r$, $\vec{e}_\theta$, $\vec{e}_\phi$, that is, the curl of $\vec{v}$ is given by the vector field $\frac{r \cos(\theta)}{\sin(\theta)}\,\vec{e}_r - 3\,r\,\vec{e}_\theta$.

The spherical coordinates are already defined in `linalg::ogCoordTab`. The last result can also be achieved with the input `curl([0, 0, r^2], [r, Theta, phi], Spherical[RightHanded])`.

```
curl([0, 0, r^2], [r, Theta, phi], Spherical[RightHanded])
```

$$\begin{pmatrix} \frac{r\cos(\mathrm{Theta})}{\sin(\mathrm{Theta})} \\ -3\,r \\ 0 \end{pmatrix}$$

Parameters

v

A list of three arithmetical expressions, or a three-dimensional vector (a 3×1 or 1 ×3 matrix of a domain of category `Cat::Matrix`)

x

A list of three (indexed) identifiers

ogCoord

The name of a three-dimensional orthogonal coordinate system predefined in the table `linalg::ogCoordTab`, or a list of algebraic expressions representing the scale factors of an orthogonal coordinate system.

c

The parameter of the coordinate systems EllipticCylindrical and Torus, respectively: an arithmetical expression. The default value is `c` = 1.

Return Values

Column vector.

See Also

MuPAD Functions

`divergence` | `gradient` | `laplacian` | `linalg::ogCoordTab` | `potential` | `vectorPotential`

', D

Differential operator for functions

Syntax

```
f '
```

```
D(f)
```

```
D([n1, n2, …], f)
```

Description

`D(f)` or, alternatively, `f'` computes the derivative of the univariate function `f`.

`D([n1, n2, ...], f)` computes the partial derivative $\frac{\partial}{\partial x_{n_2}} \frac{\partial}{\partial x_{n_1}} \cdots f$ of the multivariate function $f(x_1, x_2, \ldots)$.

MuPAD has two functions for differentiation: `diff` and `D`. `D` represents the differential operator that may be applied to functions; `diff` is used to differentiate arithmetical expressions. Mathematically, `D(f)(x)` coincides with `diff(f(x), x)`; `D([1, 2], f)(x, y)` coincides with `diff(f(x, y), x, y)`. Symbolic calls of `D` and `diff` can be converted to one another via `rewrite`. Cf. “Example 8” on page 1-490.

`D(f)` returns the derivative f' of the univariate function`f`. `f'` is shorthand for `D(f)`.

If `f` is a multivariate function and $D_n f$ denotes the partial derivative of `f` with respect to its n-th argument, then `D([n1, n2, ...], f)` computes the partial derivative $D_{n_1} D_{n_2} s f$. Cf. “Example 5” on page 1-488. In particular, `D([ ], f)` returns `f` itself.

Note: It is assumed that partial derivatives commute. Internally, `D([n1, n2, ...], f)` is converted to `D([m1, m2, ...], f)`, where `[m1, m2, ...] = sort([n1, n2, ...])`.

`f` may be any object which can represent a function. In particular, `f` may be a functional expression built from simple functions by means of arithmetic operators (`+`, `-`, `*`, `/`, `^`, `@`, `@@`). Any identifier different from CATALAN, EULER, and PI is regarded as an "unknown" function; the same holds for elements of kernel domains not explicitly mentioned on this page. Cf. "Example 1" on page 1-487. Any number and each of the three constant identifiers above is regarded as a constant function. Cf. "Example 2" on page 1-487.

If `f` is a list, a set, a table, or an array, then `D` is applied to each entry of `f`. Cf. "Example 3" on page 1-488.

A polynomial `f` of type `DOM_POLY` is regarded as polynomial function, the indeterminates being the arguments of the function. Cf. "Example 6" on page 1-489.

If `f` is a function environment, a procedure, then `D` can compute the derivative in some cases; see the "Background" section below. If this is not possible, a symbolic `D` call is returned.

Higher partial derivatives `D([n1], D([n2], f))` are simplified to `D([n1, n2], f)`. Cf. "Example 7" on page 1-490.

The derivative of a univariate function *f* —denoted by `D(f)`— is syntactically distinguished from the partial derivative `D([1], f)` with respect to the first variable, even if *f* represents a univariate function.

The usual rules of differentiation are implemented:

- `D(f + g) = D(f) + D(g)`,
- `D(f * g) = f * D(g) + g * D(f)`,
- `D(1/f) = -D(f) / f^2`,
- `D(f @ g) = D(f) @ g * D(g)`.

Note that the composition of functions is written as `f@g` and *not* as `f(g)`.

In order to express the *n*-th derivative of a univariate function for symbolic *n*, you can use the "repeated composition operator" `@@`. Cf. "Example 9" on page 1-491.

Environment Interactions

`D` uses `option remember`.

Examples

Example 1

`D(f)` computes the derivative of the function `f`:

```
D(sin), D(x -> x^2), D(id)
```

$$\cos,\ (x \to 2\,x),\ 1$$

`D` also works for more complex functional expressions:

```
D(sin @ exp + 2*(x -> x*ln(x)) + id^2)
```

$$(\cos \circ \exp)\ \exp + 2\ (x \to \ln(x) + 1) + 2\ \mathrm{id}$$

If `f` is an identifier without a value, a symbolic `D` call is returned:

```
delete f: D(f + sin)
```

$$f' + \cos$$

The same holds for objects of kernel type that cannot be regarded as functions:

```
D(NIL)
```

$$\mathrm{NIL}'$$

`f'` is shorthand for `D(f)`:

```
(f + sin)', (x -> x^2)', id'
```

$$f' + \cos,\ (x \to 2\,x),\ 1$$

Example 2

Constants are regarded as constant functions:

```
PI', 3', (1/2)'
```

0, 0, 0

Example 3

The usual rules of differentiation are implemented. Note that lists and sets may also be taken as input; in this case, `D` is applied to each element of the list or set:

```
delete f, g: D([f+g, f*g]); D({1/f, f@g})
```

$$[f' + g',\ f'\,g + f\,g']$$

$$\left\{g'\,(f' \circ g),\ -\frac{f'}{f^2}\right\}$$

Example 4

The derivatives of most special functions of the library can be computed. Again, `id` denotes the identity function:

```
D(tan); D(sin*cos); D(1/sin); D(sin@cos); D(2*sin + ln)
```

$$\tan^2 + 1$$

$$\cos^2 - \sin^2$$

$$-\frac{\cos}{\sin^2}$$

$$-(\cos \circ \cos)\,\sin$$

$$\frac{1}{\mathrm{id}} + 2\cos$$

Example 5

`D` can also compute derivatives of procedures:

```
f := x -> x^2:
g := proc(x) begin tan(ln(x)) end:
D(f), D(g)
```

$$(x \to 2\,x),\ \frac{(\tan \circ \ln)^2 + 1}{\mathrm{id}}$$

We differentiate a function of two arguments by passing a list of indices as first argument to `D`. In the example below, we first differentiate with respect to the second argument and then differentiate the result with respect to the first argument:

```
D([1, 2], (x, y) -> sin(x*y))
```

$$(x, y) \to \cos(x\,y) - x\,y\,\sin(x\,y)$$

The order of the partial derivatives is not relevant:

```
D([2, 1], (x, y) -> sin(x*y))
```

$$(x, y) \to \cos(x\,y) - x\,y\,\sin(x\,y)$$

```
delete f, g:
```

Example 6

A polynomial is regarded as a polynomial function:

```
D(poly(x^2 + 3*x + 2, [x]))
```

$$\mathrm{poly}(2\,x + 3, [x])$$

We differentiate the following bivariate polynomial `f` twice with respect to its second variable `y` and once with respect to its first variable `x`:

```
f := poly(x^3*y^3, [x, y]):
D([1, 2, 2], f) = diff(f, y, y, x)
```

$$\mathrm{poly}(18\,x^2\,y, [x, y]) = \mathrm{poly}(18\,x^2\,y, [x, y])$$

```
delete f:
```

Example 7

Nested calls to D are flattened:

```
D([1], D([2], f))
```

$$D_{1,2}(f)$$

However, this does not hold for calls with only one argument, since `D(f)` and `D([1], f)` are not considered to be the same:

```
D(D(f))
```

$$f''$$

Example 8

D may only be applied to functions whereas `diff` makes only sense for expressions:

```
D(sin), diff(sin(x), x)
```

$$\cos, \cos(x)$$

Applying D to expressions and `diff` to functions makes no sense:

```
D(sin(x)), diff(sin, x)
```

$$(\sin(x))', 0$$

`rewrite` allows to rewrite expressions with D into `diff`-expression:

```
rewrite(D(f)(y), diff), rewrite(D(D(f))(y), diff)
```

$$\frac{\partial}{\partial y} f(y), \frac{\partial^2}{\partial y^2} f(y)$$

The reverse conversion is possible as well:

```
map(%, rewrite, D)
```

$$f'(y),\ f''(y)$$

Example 9

Sometimes you may need the n-th derivative of a function, where n is unknown. This can be achieved using the repeated composition operator. For example, let us write a function that computes the k-th Taylor polynomial of a function f at a point x_0 and uses x as variable for that polynomial:

```
kthtaylorpoly:=
(f, k, x, x0) -> _plus(((D@@n)(f)(x0) * (x - x0)^n / n!) $ n = 0..k):
kthtaylorpoly(sin, 7, x, 0)
```

$$-\frac{x^7}{5040}+\frac{x^5}{120}-\frac{x^3}{6}+x$$

```
delete kthtaylorpoly:
```

Example 10

Advanced users can extend `D` to their own special mathematical functions (see "Background" section below). To this end, embed your mathematical function f, say, into a function environment`f` and implement the behavior of `D` for this function as the `"D"` slot of the function environment. The slot must handle two cases: it may be either called with only one argument which equals `f`, or with two arguments where the second one equals `f`. In the latter case, the first argument is a list of arbitrary many indices; that is, the slot must be able to handle higher partial derivatives also.

Suppose, for example, that we are given a function $f(t, x, y)$, and that we do not know anything about f except that it is differentiable infinitely often and satisfies the partial differential equation $\frac{\partial}{\partial t} f = \frac{\partial^2}{\partial x^2} f + \frac{\partial^2}{\partial y^2} f$. To make MuPAD eliminate derivatives with respect to `t`, we can do the following:

```
f := funcenv(f):
```

```
f::D :=
proc(indexlist, ff)
  local
    n        : DOM_INT,   // Number of t-derivatives.
    list_2_3 : DOM_LIST;  // List of indices of 2's and 3's.
                          // These remain unchanged.
begin
  if args(0) <> 2 then
    error("Wrong number of arguments")
  end_if;
  n        :=  nops(select(indexlist, _equal, 1));
  list_2_3 :=  select(indexlist, _unequal, 1);
  // rewrite (d/dt)^n = (d^2/dx^2 + d^2/dy^2)^n
  _plus(binomial(n, k) *
        hold(D)(sort([2 $ 2*(n-k), 3 $ 2*k].list_2_3), ff)
        $ k = 0..n)
end_proc:
```

Now, partial derivatives with respect to the first argument t are rewritten by derivatives with respect to the second and third argument:

```
D([1], f^2)(t, x, y)
```

$$2\left(\mathrm{D}_{2,2}(f)(t,x,y)+\mathrm{D}_{3,3}(f)(t,x,y)\right)f(t,x,y)$$

```
D([1, 2, 1], f)
```

$$\mathrm{D}_{2,2,2,2,2}(f)+2\,\mathrm{D}_{2,2,2,3,3}(f)+\mathrm{D}_{2,3,3,3,3}(f)$$

```
delete f:
```

Parameters

`f`

A function or a functional expression, an array, a list, a polynomial, a set, or a table

`n1, n2, …`

Indices: positive integers

Return Values

function or a functional expression. If `f` is an array or a list etc., a corresponding object containing the derivatives of the entries is returned.

Overloaded By

f

Algorithms

If `f` is a domain or a function environment with a slot`"D"`, this slot is called to compute the derivative. The slot procedure has the same calling syntax as `D`. In particular — and in contrast to the slot`"diff"`— the slot must be able to compute higher partial derivatives because the list of indices may have length greater than one. Cf. “Example 10” on page 1-491.

If `f` is a procedure, a function environment without a `"D"` slot, then `f` is called with auxiliary identifiers as arguments. The result of the call is then differentiated using the function `diff`. If the result of `diff` yields an expression which can be regarded as function in the auxiliary identifers, then this function is returned, otherwise an unevaluated call of `D` is returned.

Let us take the function environment`sin` as an example. It has no `"D"` slot, thus the procedure `op(sin, 1)`, which is responsible for evaluating the sine function, is used to compute `D(sin)`, as follows. This procedure is applied to an auxiliary identifier, say `x`, and differentiated with respect to this identifier via `diff`. The result is `diff(sin(x), x) = cos(x)`. Via `fp::expr_unapply` and `fp::unapply`, the function `cos` is computed as the derivative of `sin`.

See Also

MuPAD Functions
`diff` | `int` | `poly`

dawson

Dawson's integral

Syntax

```
dawson(x)
```

Description

`dawson(x)` represents Dawson's integral, which is defined as $e^{-x^2} \int_0^x e^{t^2} \, dt$.

`dawson(x)` returns special values for $x = 0$ and $x = \pm\infty$. For all other symbolic values of x, unevaluated function calls are returned. Cf. “Example 1” on page 1-494.

Environment Interactions

When called with floating-point arguments, this function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

For the following arguments, special values are returned:

```
dawson(0), dawson(infinity), dawson(-infinity)
```

```
0, 0, 0
```

For other symbolic arguments, a symbolic call is returned:

```
dawson(1), dawson(5+I)
```

$$\text{dawson}(1),\ \text{dawson}(5+i)$$

Floating point values are returned for floating-point arguments:

```
dawson(0.0), dawson(1.0), dawson(-3.4 + 0.2*I)
```

$$0.0,\ 0.5380795069,\ -0.1538060524 - 0.0100961726\,i$$

Example 2

The functions `diff`, `float`, `limit`, and `series` handle expressions involving the dawson function:

```
diff(dawson(x^2), x), float(dawson(7))
```

$$-2\,x\,(2\,x^2\,\text{dawson}(x^2) - 1),\ 0.07218097466$$

```
limit(x*dawson(x), x = infinity)
```

$$\frac{1}{2}$$

```
series(dawson(x), x = infinity, 4)
```

$$\frac{1}{2\,x} + \frac{1}{4\,x^3} + O\left(\frac{1}{x^5}\right)$$

Parameters

x

An arithmetical expression

Return Values

Arithmetical expression.

Overloaded By

x

See Also

MuPAD Functions
`erf` | `erfc`

debug

Execute a procedure in single-step mode

Syntax

```
debug()
```

```
debug(statement)
```

Description

`debug(statement)` starts the MuPAD debugger, allowing to execute `statement` step by step.

`debug` called with an argument switches the state of the MuPAD kernel to *debug mode* and, if `statement` contains procedure calls that can be debugged, enters the interactive MuPAD debugger for controlled single-step execution of `statement`.

If `debug` is called without arguments, the current state is returned *without* changing the state. If the debugger is on, the return value is `TRUE`, otherwise `FALSE`.

In a MuPAD version with a graphical user interface, a separate debugger window pops up. In the UNIX® terminal version, the text interface of the command line debugger is activated.

The debugger features single stepping, inspection of variables and stack frames, breakpoints, etc. Read the online help of the debugger window for a description.

Debugging is possible only for procedures written in the MuPAD language that do not have the option `noDebug`. In particular, debugging of kernel functions is not possible.

After calling `Pref::ignoreNoDebug(TRUE)`, the procedure option `noDebug` is ignored.

You can also debug a sequence of statements separated by semicolons if the sequence is enclosed in parentheses.

`debug(statement)` returns the same result as `statement`, if the execution is not aborted within the debugger by the user.

Examples

Example 1

`debug()` is called to check whether the kernel is in debug mode:

```
debug()
```

```
                              FALSE
```

To switch on the debugger mode, `debug(1)` is called:

```
debug(1)
```

```
Activating debugger...   For those library functions which are already
loaded, the format of the source code  displayed by the debugger
may differ from that of the original source code file. To avoid this,
restart the kernel in debug mode. Execution completed.
                     1
```

```
debug()
```

```
    TRUE
```

Example 2

We start the debugger for stepwise execution of the statement `int(cos(x),x)`, which integrates the cosine function:

```
debug(int(cos(x), x)):
```

Parameters

`statement`

Any MuPAD object; typically a function call

Return Values

Return value of `statement` or `TRUE` or `FALSE`.

Algorithms

In debug mode, the MuPAD parser is re-configured. When a procedure is read from a file, the parser inserts additional *debug nodes* containing file identifications and line numbers into procedures. These debug nodes allow the debugger to associate the currently executed piece of MuPAD code with the corresponding source text file.

If the debug mode is activated and MuPAD encounters a procedure without debug nodes, it will write the procedure to a temporary file and add debug nodes on the fly. This allows interactively entered procedures to be debugged in the same way as procedures read from files. The temporary debug file is deleted at the end of the session.

Since this also applies to procedures that were read before debug mode was switched on, it is recommended to start the kernel in debug mode (see below) when bigger applications are to be debugged.

If the MuPAD kernel was not started in debug mode, this mode is turned on at the first execution of `debug`. It remains activated until the end of the session.

It is possible to start the kernel in debug mode. In the MuPAD Notebook app, this can be configured by choosing “Configure …” in the “View” menu (“Preferences…” on the Apple Macintosh) and then clicking on “Kernel”. Enter “-g” in the text field “Arguments:”.

See Also

MuPAD Functions

`Pref::ignoreNoDebug` | `prog::check` | `prog::profile` | `prog::trace`

dedekindEta

The Dedekind eta function

Syntax

```
dedekindEta(z)
```

Description

`dedekindEta(z)` represents the Dedekind eta function $e^{\frac{\pi i z}{12}} \left(\prod_{n=1}^{\infty}\left(1-e^{2\pi i z n}\right)\right)$.

The Dedekind eta function is defined for all complex numbers `z` with positive imaginary part.

Floating-point results are computed for floating-point arguments. For all other arguments, the function returns symbolically.

Environment Interactions

When called with a floating-point argument, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

The Dedekind eta function takes on small values near the real axis:

```
dedekindEta(1 + 0.001*I)
```

$$6.122295553\ 10^{-113} + 1.640464149\ 10^{-113}\ i$$

Example 2

A symbolic call is returned if the argument is not a floating-point number:

```
dedekindEta(I), dedekindEta(x)
```

$$\text{dedekindEta}(i),\ \text{dedekindEta}(x)$$

Parameters

z

An arithmetical expression

Return Values

Arithmetical expression

See Also

MuPAD Functions
`theta`

degree

Degree of a polynomial

Syntax

```
degree(p)
degree(p, x)
degree(f, <vars>)
degree(f, <vars>, x)
```

Description

`degree(p)` returns the total degree of the polynomial `p`.

`degree(p, x)` returns the degree of `p` with respect to the variable `x`.

If the first argument `f` is not element of a polynomial domain, then `degree` converts the expression internally to a polynomial of type `DOM_POLY` via `poly(f)`. If a list of indeterminates is specified, the polynomial `poly(f, vars)` is considered.

`degree(f, vars, x)` returns 0 if `x` is not an element of the list `vars`.

The degree of the zero polynomial is defined as 0.

Examples

Example 1

The total degree of the terms in the following polynomial expression is computed:

```
degree(x^3 + x^2*y^2 + 2)
```

```
4
```

Example 2

`degree` may be applied to polynomials of type `DOM_POLY`:

```
degree(poly(x^2*z + x*z^3 + 1, [x, z]))
```

```
4
```

Example 3

The next expression is regarded as a bi-variate polynomial in `x` and `z`. The degree with respect to `z` is computed:

```
degree(x^2*z + x*z^3 + 1, [x, z], z)
```

```
3
```

Example 4

The degree of the zero polynomial is defined as 0:

```
degree(0, [x, y])
```

```
0
```

Parameters

`p`

A polynomial of type `DOM_POLY`

`f`

A polynomial expression

`vars`

A list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

x

An indeterminate

Return Values

Nonnegative number. `FAIL` is returned if the input cannot be converted to a polynomial.

Overloaded By

f, p

See Also

MuPAD Functions

`coeff` | `degreevec` | `ground` | `lcoeff` | `ldegree` | `lmonomial` | `lterm` | `monomials` | `nterms` | `nthcoeff` | `nthmonomial` | `nthterm` | `poly` | `poly2list` | `tcoeff`

degreevec

Exponents of the leading term of a polynomial

Syntax

```
degreevec(p, <order>)
```

```
degreevec(f, <vars>, <order>)
```

Description

`degreevec(p)` returns a list with the exponents of the leading term of the polynomial `p`.

For a polynomial in the variables x_1, x_2, …, x_n with the leading term $x_1^{e_1} x_2^{e_2} \ldots x_n^{e_n}$, the exponent vector $[e_1, e_2, \ldots, e_n]$ is returned.

`degreevec` returns a list of zeroes for the zero polynomial.

If the first argument `f` is not element of a polynomial domain, then `degreevec` converts the expression internally to a polynomial of type `DOM_POLY` via `poly(f)`. If a list of indeterminates is specified, the polynomial `poly(f, vars)` is considered. `FAIL` is returned if `f` cannot be converted to a polynomial.

Examples

Example 1

The leading term of the following polynomial expression (with respect to the main variable `x`) is x^4:

```
degreevec(x^4 + x^2*y^3 + 2, [x, y])
```

```
[4, 0]
```

With the main variable y, the leading term is $x^2 y^3$:

```
degreevec(x^4 + x^2*y^3 + 2, [y, x])
```

```
[3, 2]
```

For polynomials of type DOM_POLY, the indeterminates are an integral part of the data type:

```
degreevec(poly(x^4 + x^2*y^3 + 2, [x, y])),
degreevec(poly(x^4 + x^2*y^3 + 2, [y, x]))
```

```
[4, 0], [3, 2]
```

Example 2

For a univariate polynomial, the standard term orderings regard the same term as "leading":

```
degreevec(poly(x^2*z + x*z^3 + 1, [x]), LexOrder),
degreevec(poly(x^2*z + x*z^3 + 1, [x]), DegreeOrder),
degreevec(poly(x^2*z + x*z^3 + 1, [x]), DegInvLexOrder)
```

```
[2], [2], [2]
```

In the multivariate case, different polynomial orderings may yield different leading exponent vectors:

```
degreevec(poly(x^2*z + x*z^3 + 1, [x, z])),
degreevec(poly(x^2*z + x*z^3 + 1, [x, z]), DegreeOrder)
```

```
[2, 1], [1, 3]
```

```
degreevec(x^3 + x*y^2*z - 5*y^4, [x, y, z], LexOrder),
degreevec(x^3 + x*y^2*z - 5*y^4, [x, y, z], DegreeOrder),
degreevec(x^3 + x*y^2*z - 5*y^4, [x, y, z], DegInvLexOrder)
```

```
[3, 0, 0], [1, 2, 1], [0, 4, 0]
```

Example 3

The exponent vector of the zero polynomial is a list of zeroes:

```
degreevec(0, [x, y, z])
```

```
[0, 0, 0]
```

Parameters

`p`

A polynomial of type `DOM_POLY`

`f`

A polynomial expression

`vars`

A list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

`order`

The term ordering: either `LexOrder`, or `DegreeOrder`, or `DegInvLexOrder`, or a user-defined term ordering of type `Dom::MonomOrdering`. The default is the lexicographical ordering `LexOrder`.

Return Values

List of nonnegative integers. `FAIL` is returned if the input cannot be converted to a polynomial.

Overloaded By

f, p

See Also

MuPAD Functions

`coeff` | `degree` | `ground` | `lcoeff` | `ldegree` | `lmonomial` | `lterm` | `monomials` | `nterms` | `nthcoeff` | `nthmonomial` | `nthterm` | `poly` | `poly2list` | `tcoeff`

delete, _delete

Delete the value of an identifier

Syntax

```
delete x1, x2, …
```

```
_delete(x1, x2, …)
```

Description

The statement `delete x` deletes the value of the identifier `x`.

For many computations, symbolic variables are needed. E.g., solving an equation for an unknown `x` requires an identifier `x` that does not have a value. If `x` has a value, the statement `delete x` deletes the value and `x` can be used as a symbolic variable.

The statement `delete x1, x2, ...` is equivalent to the function call `_delete(x1, x2, ...)`. The values of all specified identifiers are deleted.

The statement `delete x[j]` deletes the entry `j` of a list, an array, an hfarray, or a table named `x`. Deletion of elements or entries reduces the size of lists and tables, respectively.

If `x` is an identifier carrying properties set via `assume`, then `delete x` detaches all properties from `x`, i.e., `delete x` has the same effect as `unassume(x)`. Cf. “Example 3” on page 1-511.

Examples

Example 1

The identifiers `x`, `y` are assigned values. After deletion, the identifiers have no values any longer:

```
x := 42: y := 7: delete x: x, y
```

$x, 7$

```
delete y: x, y
```

x, y

More than one identifier can be deleted by one call:

```
a := b := c := 42: a, b, c
```

$42, 42, 42$

```
delete a, b, c: a, b, c
```

a, b, c

Example 2

`delete` can also be used to delete specific elements of lists, arrays, hfarrays, and tables:

```
L := [7, 13, 42]
```

$[7, 13, 42]$

```
delete L[2]: L
```

$[7, 42]$

```
A := array(1..3, [7, 13, 42])
```

$(7\ 13\ 42)$

```
delete A[2]: A, A[2]
```

$(7\ \text{NIL}\ 42), A_2$

```
T := table(1 = 7, 2 = 13, 3 = 42)
```

1	7
2	13
3	42

```
delete T[2]: T
```

1	7
3	42

Note that `delete` does not evaluate the objects that are to be deleted. In the following, an element of the list `U` is deleted. The original value of `U` (the list `L`) is not changed:

```
U := L: delete U[1]: U, L
```

$$[42], [7, 42]$$

Finally, all assigned values are deleted:

```
delete U, L, A, T: U, L, A, T
```

$$U, L, A, T$$

Example 3

`delete` can also be used to delete properties of identifiers set via `assume`. With the assumption "`x > 1`", the expression `ln(x)` has the property "`ln(x) > 0`", i.e., its sign is 1:

```
assume(x > 1): sign(ln(x))
```

$$1$$

Without a property of `x`, the function `sign` cannot determine the sign of `ln(x)`:

```
delete x: sign(ln(x))
```

$$\mathrm{sign}(\ln(x))$$

Parameters

x_1, x_2, …

identifiers or indexed identifiers

Return Values

Void object of type `DOM_NULL`.

See Also

MuPAD Functions

`:=` | `_assign` | `assign` | `assignElements` | `evalassign`

denom

Denominator of a rational expression

Syntax

```
denom(f)
```

Description

`denom(f)` returns the denominator of the expression `f`.

`denom` regards the input as a rational expression: non-rational subexpressions such as `sin(x)`, `x^(1/2)` etc. are internally replaced by "temporary variables". The denominator of this rationalized expression is computed, the temporary variables are finally replaced by the original subexpressions.

Note: Numerator and denominator are not necessarily cancelled: the denominator returned by `denom` may have a non-trivial `gcd` with the numerator returned by `numer`. Pre-process the expression by `normal` to enforce cancellation of common factors. Cf. "Example 2" on page 1-514.

Examples

Example 1

We compute the denominators of some expressions:

```
denom(-3/4)
```

```
     4
```

```
denom(x + 1/(2/3*x -2/x))
```

$$2x^2 - 6$$

```
denom((cos(x)^2 -1)/(cos(x) -1))
```

$$\cos(x) - 1$$

Example 2

`denom` performs no cancellations if the rational expression is of the form "numerator/denominator":

```
r := (x^2 - 1)/(x^3 - x^2 + x - 1): denom(r)
```

$$x^3 - x^2 + x - 1$$

This denominator has a common factor with the numerator of `r`; `normal` enforces cancellation of common factors:

```
denom(normal(r))
```

$$x^2 + 1$$

However, automatic normalization occurs if the input expression is a sum:

```
denom(r + x/(x + 1) + 1/(x + 1) - 1)
```

$$x^2 + 1$$

```
delete r:
```

Parameters

`f`

An arithmetical expression

Return Values

Arithmetical expression.

Overloaded By

f

See Also

MuPAD Functions
`factor` | `gcd` | `normal` | `numer`

densematrix

Create a matrix or a vector

Syntax

```
densematrix(Array)
densematrix(List)
densematrix(ListOfRows)
densematrix(Matrix)
densematrix(m, n)
densematrix(m, n, Array)
densematrix(m, n, List)
densematrix(m, n, ListOfRows)
densematrix(m, n, f)
densematrix(m, n, List, Diagonal)
densematrix(m, n, g, Diagonal)
densematrix(m, n, List, Banded)
densematrix(1, n, Array)
densematrix(1, n, List)
densematrix(m, 1, Array)
densematrix(m, 1, List)
```

Description

`densematrix(m, n, [[a11, a12, ...], [a21, a22, ...], ...])` returns the $m \times n$ matrix

$$\begin{pmatrix} a_{11} & a_{12} & \cdots \\ a_{21} & a_{22} & \cdots \\ \vdots & \vdots & \vdots \end{pmatrix}.$$

`densematrix(n, 1, [a1, a2, ...])` returns the $n \times 1$ column vector

$$\begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \end{pmatrix}.$$

`densematrix(1, n, [a1, a2, ...])` returns the $1 \times n$ row vector

$$\begin{pmatrix} a_{11} & a_{21} & \cdots \end{pmatrix}.$$

`densematrix` creates matrices and vectors. A vector with n entries is either an $n \times 1$ matrix (a column vector) or a $1 \times n$ matrix (a row vector).

Matrix and vector components must be arithmetical expressions. For specific component domains, refer to the help page of `Dom::DenseMatrix`.

Arithmetical operations with matrices can be performed by using the standard arithmetical operators of MuPAD.

E.g., if `A` and `B` are two matrices defined by `densematrix`, then `A + B` computes the sum and `A * B` computes the product of the two matrices, provided that the dimensions are correct.

Similarly, `A^(-1)` or `1/A` computes the inverse of a square matrix `A` if it exists. Otherwise, `FAIL` is returned.

See “Example 1” on page 1-520.

Many system functions accept matrices as input, such as `map`, `subs`, `has`, `zip`, `conjugate` to compute the complex conjugate of a matrix, `norm` to compute matrix norms, or even `exp` to compute the exponential of a matrix. See “Example 4” on page 1-523.

Most of the functions in the MuPAD linear algebra package linalg work with matrices. For example, the command `linalg::gaussJordan(A)` performs Gauss-Jordan elimination on `A` to transform `A` to its reduced row echelon form. See “Example 2” on page 1-521.

See the help page of linalg for a list of available functions of this package.

`dendematrix` is an abbreviation for the domain `Dom::DenseMatrix()`. You find more information about this data type for matrices on the corresponding help page.

Matrix components can be extracted by the usual index operator `[ ]`, which also works for lists, arrays, hfarrays, and tables. The call `A[i, j]` extracts the matrix component in the *i*th row and the *j*th column.

Assignments to matrix components are performed similarly. The call `A[i, j] := c` replaces the matrix component in the *i*th row and the *j*th column of *A* by `c`.

If one of the indices is not in its valid range, then an error message is issued.

The index operator also extracts submatrices. The call `A[r1..r2, c1..c2]` creates the submatrix of *A* comprising the rows with the indices r_1, $r_1 + 1$, …, r_2 and the columns with the indices c_1, $c_1 + 1$, …, c_2 of *A*.

See “Example 3” on page 1-522 and “Example 5” on page 1-525.

`dendematrix(Array)` or `dendematrix(Matrix)` create a new matrix with the same dimension and the components of `Array` or `Matrix`, respectively. The array must not contain any uninitialized entries. If `Array` is one-dimensional, then the result is a column vector. Cf. “Example 7” on page 1-527.

`dendematrix(List)` creates an $m \times 1$ column vector with components taken from the nonempty list, where m is the number of entries of `List`. See “Example 5” on page 1-525.

`dendematrix(ListOfRows)` creates an $m \times n$ matrix with components taken from the nested list`ListOfRows`, where m is the number of inner lists of `ListOfRows`, and n is the maximal number of elements of an inner list. Each inner list corresponds to a row of the matrix. Both m and n must be non-zero.

If an inner list has less than n entries, then the remaining components in the corresponding row of the matrix are set to zero. See “Example 6” on page 1-526.

It might be a good idea first to create a two-dimensional array from that list before calling `densematrix`. This is due to the fact that creating a matrix from an array is the fastest way one can achieve. However, in this case the sublists must have the same number of elements.

The call `densematrix(m, n)` returns the $m \times n$ zero matrix.

The call `densematrix(m, n, Array)` creates an $m \times n$ matrix with components taken from `Array`, which must be an array or an hfarray. `Array` must have $m\,n$ operands. The first m operands define the first row, the next m operands define the second row, etc. The formatting of the array is irrelevant. E.g., any array with 6 elements can be used to create matrices of dimension 1×6, or 2×3, or 3×2, or 6×1.

`densematrix(m, n, List)` creates an $m \times n$ matrix with components taken row after row from the non-empty list. The list must contain $m\,n$ entries. Cf. “Example 6” on page 1-526.

`densematrix(m, n, ListOfRows)` creates an $m \times n$ matrix with components taken from the list `ListOfRows`.

If $m \geq 2$ and $n \geq 2$, then `ListOfRows` must consist of at most `m` inner lists, each having at most `n` entries. The inner lists correspond to the rows of the returned matrix.

If an inner list has less than `n` entries, then the remaining components of the corresponding row of the matrix are set to zero. If there are less than `m` inner lists, then the remaining lower rows of the matrix are filled with zeroes. See “Example 6” on page 1-526.

`densematrix(m, n, f)` returns the matrix whose (i, j)th component is `f(i,j)`. The row index i runs from 1 to m and the column index j from 1 to n. See “Example 8” on page 1-529.

`densematrix(m, 1, Array)` returns the $m \times 1$ column vector with components taken from `Array`. The `array` or `hfarray` `Array` must have `m` entries.

`densematrix(m, 1, List)` returns the $m \times 1$ column vector with components taken from `List`. The list `List` must have at most `m` entries. If there are fewer entries, then the remaining vector components are set to zero. See “Example 5” on page 1-525.

`densematrix(1, n, Array)` returns the $1 \times n$ row vector with components taken from `Array`. The `array` or `hfarray` `Array` must have `n` entries.

`densematrix(1, n, List)` returns the $1 \times n$ row vector with components taken from `List`. The list `List` must have at most `n` entries. If there are fewer entries, then the remaining vector components are set to zero. See “Example 5” on page 1-525.

Note: The components of a matrix are no longer evaluated after the creation of the matrix, i.e., if they contain free identifiers they will not be replaced by their values.

Examples

Example 1

We create the 2×2 matrix

$$\begin{pmatrix} 1 & 5 \\ 2 & 3 \end{pmatrix}$$

by passing a list of two rows to `densematrix`, where each row is a list of two elements, as follows:

```
A := densematrix([[1, 5], [2, 3]])
```

$$\begin{pmatrix} 1 & 5 \\ 2 & 3 \end{pmatrix}$$

In the same way, we generate the following 2 ×3 matrix:

```
B := densematrix([[-1, 5/2, 3], [1/3, 0, 2/5]])
```

$$\begin{pmatrix} -1 & \frac{5}{2} & 3 \\ \frac{1}{3} & 0 & \frac{2}{5} \end{pmatrix}$$

We can do matrix arithmetic using the standard arithmetical operators of MuPAD. For example, the matrix product $A\,B$, the 4th power of A, and the scalar multiplication of A by $\frac{1}{3}$ are given by:

```
A * B, A^4, 1/3 * A
```

$$\begin{pmatrix} \frac{2}{3} & \frac{5}{2} & 5 \\ -1 & 5 & \frac{36}{5} \end{pmatrix}, \begin{pmatrix} 281 & 600 \\ 240 & 521 \end{pmatrix}, \begin{pmatrix} \frac{1}{3} & \frac{5}{3} \\ \frac{2}{3} & 1 \end{pmatrix}$$

Since the dimensions of the matrices *A* and *B* differ, the sum of *A* and *B* is not defined and MuPAD returns an error message:

```
A + B
```

```
Error: The dimensions do not match. [(Dom::DenseMatrix(Dom::ExpressionField()))::_plus]
```

To compute the inverse of *A*, enter:

```
1/A
```

$$\begin{pmatrix} -\frac{3}{7} & \frac{5}{7} \\ \frac{2}{7} & -\frac{1}{7} \end{pmatrix}$$

If a matrix is not invertible, then the result of this operation is `FAIL`:

```
C := densematrix([[2, 0], [0, 0]])
```

$$\begin{pmatrix} 2 & 0 \\ 0 & 0 \end{pmatrix}$$

```
C^(-1)
```

FAIL

Example 2

In addition to standard matrix arithmetic, the library linalg offers a lot of functions handling matrices. For example, the function `linalg::rank` determines the rank of a matrix:

```
A := densematrix([[1, 5], [2, 3]])
```

$$\begin{pmatrix} 1 & 5 \\ 2 & 3 \end{pmatrix}$$

```
linalg::rank(A)
```

$$2$$

The function `linalg::eigenvectors` computes the eigenvalues and the eigenvectors of A:

```
linalg::eigenvectors(A)
```

$$\left[\left[2-\sqrt{11},\ 1,\ \left[\begin{pmatrix} -\frac{\sqrt{11}}{2}-\frac{1}{2} \\ 1 \end{pmatrix}\right]\right],\ \left[\sqrt{11}+2,\ 1,\ \left[\begin{pmatrix} \frac{\sqrt{11}}{2}-\frac{1}{2} \\ 1 \end{pmatrix}\right]\right]\right]$$

To determine the dimension of a matrix use the function `linalg::matdim`:

```
linalg::matdim(A)
```

$$[2, 2]$$

The result is a list of two positive integers, the row and column number of the matrix.

Use `info(linalg)` to obtain a list of available functions, or enter `?linalg` for details about this library.

Example 3

Matrix entries can be accessed with the index operator `[ ]`:

```
A := densematrix([[1, 2, 3, 4], [2, 0, 4, 1], [-1, 0, 5, 2]])
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 0 & 4 & 1 \\ -1 & 0 & 5 & 2 \end{pmatrix}$$

```
A[2, 1] * A[1, 2] - A[3, 1] * A[1, 3]
```

$$7$$

You can redefine a matrix entry by assigning a value to it:

```
A[1, 2] := a^2: A
```

$$\begin{pmatrix} 1 & a^2 & 3 & 4 \\ 2 & 0 & 4 & 1 \\ -1 & 0 & 5 & 2 \end{pmatrix}$$

The index operator can also be used to extract submatrices. The following call creates a copy of the submatrix of *A* comprising the second and the third row and the first three columns of *A*:

```
A[2..3, 1..3]
```

$$\begin{pmatrix} 2 & 0 & 4 \\ -1 & 0 & 5 \end{pmatrix}$$

The index operator does *not* allow to replace a submatrix of a given matrix by another matrix. Use `linalg::substitute` to achieve this.

Example 4

Some system functions can be applied to matrices. For example, if you have a matrix with symbolic entries and want to have all entries in expanded form, simply apply the function `expand`:

```
delete a, b:
A := densematrix([
  [(a - b)^2, a^2 + b^2],
  [a^2 + b^2, (a - b)*(a + b)]
])
```

$$\begin{pmatrix} (a-b)^2 & a^2+b^2 \\ a^2+b^2 & (a+b)\,(a-b) \end{pmatrix}$$

```
expand(A)
```

$$\begin{pmatrix} a^2 - 2ab + b^2 & a^2 + b^2 \\ a^2 + b^2 & a^2 - b^2 \end{pmatrix}$$

You can differentiate all matrix components with respect to some indeterminate:

```
diff(A, a)
```

$$\begin{pmatrix} 2a - 2b & 2a \\ 2a & 2a \end{pmatrix}$$

The following command evaluates all matrix components at a given point:

```
subs(A, a = 1, b = -1)
```

$$\begin{pmatrix} 4 & 2 \\ 2 & 0 \end{pmatrix}$$

Note that the function `subs` does not evaluate the result of the substitution. For example, we define the following matrix:

```
A := densematrix([[sin(x), x], [x, cos(x)]])
```

$$\begin{pmatrix} \sin(x) & x \\ x & \cos(x) \end{pmatrix}$$

Then we substitute $x = 0$ in each matrix component:

```
B := subs(A, x = 0)
```

$$\begin{pmatrix} \sin(0) & 0 \\ 0 & \cos(0) \end{pmatrix}$$

You see that the matrix components are not evaluated completely: for example, if you enter `sin(0)` directly, it evaluates to zero.

The function `eval` can be used to evaluate the result of the function `subs`. However, `eval` does not operate on matrices directly, and you must use the function `map` to apply the function `eval` to each matrix component:

```
map(B, eval)
```

$$\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

The function `zip` can be applied to matrices. The following call combines two matrices A and B by dividing each component of A by the corresponding component of B:

```
A := densematrix([[4, 2], [9, 3]]):
B := densematrix([[2, 1], [3, -1]]):
zip(A, B, `/`)
```

$$\begin{pmatrix} 2 & 2 \\ 3 & -3 \end{pmatrix}$$

Example 5

A vector is either an $m \times 1$ matrix (a column vector) or a $1 \times n$ matrix (a row vector). To create a vector with `densematrix`, pass the dimension of the vector and a list of vector components as argument to `densematrix`:

```
row_vector    := densematrix(1, 3, [1, 2, 3]);
column_vector := densematrix(3, 1, [1, 2, 3])
```

$$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

If the only argument of `densematrix` is a non-nested list or a one-dimensional array, then the result is a column vector:

```
densematrix([1, 2, 3])
```

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

For a row vector `r`, the calls `r[1, i]` and `r[i]` both return the *i*th vector component of `r`. Similarly, for a column vector `c`, the calls `c[i, 1]` and `c[i]` both return the *i*th vector component of `c`.

For example, to extract the second component of the vectors `row_vector` and `column_vector`, we enter:

```
row_vector[2], column_vector[2]
```

$$2, 2$$

Use the function `linalg::vecdim` to determine the number of components of a vector:

```
linalg::vecdim(row_vector), linalg::vecdim(column_vector)
```

$$3, 3$$

The number of components of a vector can also be determined directly by the call `nops(vector)`.

The dimension of a vector can be determined as described above in the case of matrices:

```
linalg::matdim(row_vector),
linalg::matdim(column_vector)
```

$$[1, 3], [3, 1]$$

See the linalg package for functions working with vectors, and the help page of `norm` for computing vector norms.

Example 6

In the following examples, we illustrate various calls of `densematrix` as described above. We start by passing a nested list to `densematrix`, where each inner list corresponds to a row of the matrix:

```
densematrix([[1, 2], [2]])
```

$$\begin{pmatrix} 1 & 2 \\ 2 & 0 \end{pmatrix}$$

The number of rows of the created matrix is the number of inner lists, namely $m = 2$. The number of columns is determined by the maximal number of entries of an inner list. In the example above, the first list is the longest one, and hence $n = 2$. The second list has only one element, and therefore the second entry in the second row of the returned matrix was set to zero.

In the following call, we use the same nested list, but in addition pass two dimension parameters to create a 4×4 matrix:

```
densematrix(4, 4, [[1, 2], [2]])
```

$$\begin{pmatrix} 1 & 2 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

In this case, the dimension of the matrix is given by the dimension parameters. As before, missing entries in an inner list correspond to zero, and in addition missing rows are treated as zero rows.

If the dimension $m \times n$ of the matrix is stated explicitly, the entries may also be specified by a plain list with $m\ n$ elements. The matrix is filled with these elements row by row:

```
densematrix(2, 3, [1, 2, 3, 4, 5, 6])
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

```
densematrix(3, 2, [1, 2, 3, 4, 5, 6])
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

Example 7

A one- or two-dimensional array of arithmetical expressions, such as:

```
a := array(1..3, 2..4, [[ 1,  1/3 ,  0 ],
                        [-2,  3/5 , 1/2],
```

```
[-3/2, 0 ,  -1 ]])
```

$$\begin{pmatrix} 1 & \frac{1}{3} & 0 \\ -2 & \frac{3}{5} & \frac{1}{2} \\ -\frac{3}{2} & 0 & -1 \end{pmatrix}$$

can be converted into a matrix as follows:

```
A := densematrix(a)
```

$$\begin{pmatrix} 1 & \frac{1}{3} & 0 \\ -2 & \frac{3}{5} & \frac{1}{2} \\ -\frac{3}{2} & 0 & -1 \end{pmatrix}$$

Arrays serve, for example, as an efficient structured data type for programming. However, arrays do not have any algebraic meaning, and no mathematical operations are defined for them. If you convert an array into a matrix, you can use the full functionality defined for matrices as described above. For example, let us compute the matrix $2A - A^2$ and the Frobenius norm of A:

```
2*A - A^2, norm(A, Frobenius)
```

$$\begin{pmatrix} \frac{5}{3} & \frac{2}{15} & -\frac{1}{6} \\ -\frac{1}{20} & \frac{113}{75} & \frac{6}{5} \\ -3 & \frac{1}{2} & -3 \end{pmatrix}, \frac{\sqrt{450}\,\sqrt{4037}}{450}$$

Note that an array may contain uninitialized entries:

```
b := array(1..4): b[1] := 2: b[4] := 0: b
```

$$(2 \text{ NIL NIL } 0)$$

`densematrix` cannot handle arrays that have uninitialized entries, and responds with an error message:

```
densematrix(b)
```

```
Error: Cannot define a matrix over 'Dom::ExpressionField()'. [(Dom::DenseMatrix(Dom::Ex
```

We initialize the remaining entries of the array `b` and convert it into a matrix, or more precisely, into a column vector:

```
b[2] := 0: b[3] := -1: densematrix(b)
```

$$\begin{pmatrix} 2 \\ 0 \\ -1 \\ 0 \end{pmatrix}$$

Example 8

We show how to create a matrix whose components are defined by a function of the row and the column index. The entry in the *i*th row and the *j*th column of a Hilbert matrix (see also `linalg::hilbert`) is $\frac{1}{(i+j-1)}$. Thus the following command creates a 2×2 Hilbert matrix:

```
densematrix(2, 2, (i, j) -> 1/(i + j - 1))
```

$$\begin{pmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{3} \end{pmatrix}$$

The following two calls produce different results. In the first call, `x` is regarded as an unknown function, while it is a constant in the second call:

```
delete x:
densematrix(2, 2, x), densematrix(2, 2, (i, j) -> x)
```

$$\begin{pmatrix} x(1,1) & x(1,2) \\ x(2,1) & x(2,2) \end{pmatrix}, \begin{pmatrix} x & x \\ x & x \end{pmatrix}$$

Example 9

Diagonal matrices can be created by passing the option `Diagonal` and a list of diagonal entries:

```
densematrix(3, 4, [1, 2, 3], Diagonal)
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \end{pmatrix}$$

Hence, you can generate the 3×3 identity matrix as follows:

```
densematrix(3, 3, [1 $ 3], Diagonal)
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Equivalently, you can use a function of one argument:

```
densematrix(3, 3, i -> 1, Diagonal)
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Since the integer `1` also represents a constant function, the following shorter call creates the same matrix:

```
densematrix(3, 3, 1, Diagonal)
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Example 10

Banded Toeplitz matrices (see above) can be created with the option `Banded`. The following command creates a matrix of bandwidth 3 with all main diagonal entries equal to 2 and all entries on the first sub- and superdiagonal equal to - 1:

```
densematrix(4, 4, [-1, 2, -1], Banded)
```

$$\begin{pmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{pmatrix}$$

Parameters

`Array`

A one- or two-dimensional array of type `DOM_ARRAY` or `DOM_HFARRAY`

`List`

A list of arithmetical expressions

`ListOfRows`

A nested list of rows, each row being a list of arithmetical expressions

`Matrix`

A matrix, i.e., an object of a data type of category `Cat::Matrix`

`m`

The number of rows: a positive integer

`n`

The number of columns: a positive integer

`f`

A function or a functional expression of two arguments

`g`

A function or a functional expression of one argument

Options

`Diagonal`

Create a diagonal matrix

With the option `Diagonal`, diagonal matrices can be created with diagonal elements taken from a list, or computed by a function or a functional expression.

`densematrix(m, n, List, Diagonal)` creates the $m \times n$ diagonal matrix whose diagonal elements are the entries of `List`. See "Example 9" on page 1-530.

`List` must have at most $min(m, n)$ entries. If it has fewer elements, then the remaining diagonal elements are set to zero.

`densematrix(m, n, g, Diagonal)` returns the matrix whose ith diagonal element is `g(i, i)`, where the index i runs from 1 to $min(m, n)$. See "Example 9" on page 1-530.

`Banded`

Create a banded Toeplitz matrix

A *banded matrix* has all entries zero outside the main diagonal and some of the adjacent sub- and superdiagonals.

`densematrix(m, n, List, Banded)` creates an $m \times n$ banded Toeplitz matrix with the elements of `List` as entries. The number of entries of `List` must be odd, say $2h + 1$, and must not exceed `n`. The bandwidth of the resulting matrix is at most h.

All elements of the main diagonal of the created matrix are initialized with the middle element of `List`. All elements of the ith subdiagonal are initialized with the $(h + 1 - i)$th element of `List`. All elements of the ith superdiagonal are initialized with the $(h + 1 + i)$th element of `List`. All entries on the remaining sub- and superdiagonals are set to zero.

See "Example 10" on page 1-530.

Return Values

Matrix of the domain type`Dom::DenseMatrix()`.

See Also

MuPAD Domains
`Dom::DenseMatrix` | `Dom::Matrix` | `DOM_ARRAY` | `DOM_HFARRAY`

MuPAD Functions
`array` | `hfarray` | `matrix`

det

Determinant of a matrix

Syntax

```
det(A, options)
```

Description

`det(A)` returns the determinant of the matrix *A*.

If the input matrix is an `array` of domain type `DOM_ARRAY`, then `numeric::det(A, Symbolic)` is called to compute the result.

The determinant of `hfarray`s of domain type `DOM_HFARRAY` is internally computed via `numeric::det(A)`.

If the argument does not evaluate to a matrix of one of the types mentioned above, a symbolic call `det(A)` is returned.

The `MinorExpansion` option is useful for small matrices (typically, matrices of dimension up to 10) containing many symbolic entries. By default, `det` tries to recognize matrices that can benefit from using `MinorExpansion`, and uses this option when computing their determinants. Nevertheless, `det` does not always recognize these matrices. Also, identifying that a matrix is small enough and contains many symbolic entries takes time. To improve performance, use the `MinorExpansion` option explicitly.

By default, `det` calls `normal` before returning results. This additional internal call ensures that the final result is normalized. This call can be computationally expensive. It also affects the result returned by `det` only if a matrix contains variables or exact expressions, such as `sqrt(5)` or `sin(PI/7)`.

To avoid this additional call, specify `Normal = FALSE`. In this case, `det` also can return normalized results, but does not guarantee such normalization. See “Example 3” on page 1-536 and “Example 4” on page 1-536.

Examples

Example 1

We compute the determinant of a matrix given by various data types:

```
A := array(1..2, 1..2, [[1, 2], [3, PI]]);
det(A)
```

$$\begin{pmatrix} 1 & 2 \\ 3 & \pi \end{pmatrix}$$

$$\pi - 6$$

```
B := hfarray(1..2, 1..2, [[1, 2], [3, PI]]);
det(B)
```

$$\begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 3.141592654 \end{pmatrix}$$

$$-2.858407346$$

```
C := matrix(2, 2, [[1, 2], [3, PI]]);
det(C)
```

$$\begin{pmatrix} 1 & 2 \\ 3 & \pi \end{pmatrix}$$

$$\pi - 6$$

```
delete A, B, C:
```

Example 2

If the input does not evaluate to a matrix, then symbolic calls are returned:

```
delete A, B:
det(A + 2*B)
```

$$\det(A + 2B)$$

Example 3

If you use the `Normal` option, `det` calls the `normal` function for final results. This call ensures that `det` returns results in normalized form:

```
det(matrix([[x, x^2], [x/(x + 2), 1/x]]))
```

$$\frac{-x^3 + x + 2}{x + 2}$$

If you specify `Normal = FALSE`, `det` does not call `normal` for the final result:

```
det(matrix([[x, x^2], [x/(x + 2), 1/x]]), Normal = FALSE)
```

$$1 - \frac{x^3}{x + 2}$$

Example 4

Using `Normal` can significantly decrease performance of `det`. For example, computing the determinant of this matrix takes a long time:

```
n := 5:
det5 := det(matrix([[(x[i*j]^(i + j) + x[i+j]^j)/(i + j) $
                                                  j = 1..n] $
                                                  i = 1..n])):
```

For better performance, specify `Normal = FALSE`:

```
n := 5:
det5 := det(matrix([[(x[i*j]^(i + j) + x[i+j]^j)/(i + j) $
                                                  j = 1..n] $
                                                  i = 1..n]),
                                            Normal = FALSE):
```

Parameters

A

Square matrix: either a two-dimensional `array`, a two-dimensional `hfarray`, or an object of the category `Cat::Matrix`

Options

`MinorExpansion`

Compute the determinant by a recursive minor expansion along the first column.

`Normal`

Option, specified as `Normal = b`

Return normalized results. The value `b` must be `TRUE` or `FALSE`. By default, `Normal = TRUE`, meaning that `det` guarantees normalization of the returned results. Normalizing results can be computationally expensive.

Return Values

Arithmetical expression.

Overloaded By

A

See Also

MuPAD Functions
`numeric::det`

More About

- "Compute Determinants and Traces of Square Matrices"

diff

Differentiate an expression or a polynomial

Syntax

```
diff(f)
```

```
diff(f, x)
```

```
diff(f, x1, x2, …)
```

Description

`diff(f, x)` computes the derivative $\frac{\partial}{\partial x} f$ of the function `f` with respect to the variable `x`.

`diff(f, x)` computes the partial derivative of the arithmetical expression (or polynomial) `f` with respect to the indeterminate `x`.

`diff(f)` computes the 0th derivative of `f`. Since the 0th derivative of `f` is `f` itself, `diff(f)` returns its evaluated argument.

`diff(f, x1, x2, ...)` is equivalent to `diff(...diff(diff(f, x1), x2)...)`. In both cases, MuPAD first differentiates `f` with respect to `x1`, then differentiates the result with respect to `x2`, and so on. The result is the partial derivative $\cdots \frac{\partial}{\partial x_1} \frac{\partial}{\partial x_2} f$. See "Example 2" on page 1-540.

If you use nested `diff` calls, the system internally converts them into a single `diff` call with multiple arguments. See "Example 3" on page 1-540.

When computing the second and higher derivatives, use the sequence operator as a shortcut. If `n` is a nonnegative integer, `diff(f, x $ n)` returns the *n*th derivative of `f` with respect to `x`. See "Example 4" on page 1-540.

The indeterminates `x, x1, x2, ...` must be identifiers of domain type `DOM_IDENT` or indexed identifiers of the form `x[n]` where `x` is an identifier and `n` is an integer. If any

indeterminate comes in any other form, MuPAD returns an unresolved `diff` call. See “Example 5” on page 1-541.

If `f` is an arithmetical expression, `diff` returns an arithmetical expression. If `f` is a polynomial, `diff` returns a polynomial. See “Example 6” on page 1-541.

If the system cannot compute the derivative, it returns an unresolved `diff` call. See “Example 7” on page 1-541.

MuPAD assumes that partial derivatives with respect to different indeterminates commute. The function calls `diff(f, x1, x2)` and `diff(f, x2, x1)` produce the same result `diff(f, y1, y2)`. Here `[y1, y2] = sort([x1, x2])`. See “Example 8” on page 1-542.

MuPAD provides two functions, `diff` and `D`, for computing derivatives. Use the differential operator `D` to compute the derivatives of functions. Use the `diff` function to compute the derivatives of arithmetical expressions. Mathematically, `D(f)(x)` coincides with `diff(f(x), x)` and `D([1, 2], f)(x, y)` coincides with `diff(f(x, y), x, y)`. You can convert symbolic calls of `D` to the calls of `diff` and vice versa by using `rewrite`. See “Example 10” on page 1-542.

You can extend the functionality of `diff` for your own special mathematical functions via overloading. This approach works by turning the corresponding function into a function environment and implementing the differentiation rule for the function as the `"diff"` `slot` of the function environment.

If a subexpression of the form `g(..)` occurs in `f`, and `g` is a function environment, then `diff(f, x)` attempts to call the `"diff"` slot of `g` to determine the derivative of `g(..)`.

The system calls the `"diff"` slot with the arguments `g(..), x`.

If `g` does not have a `"diff"` slot, then the system function `diff` returns the symbolic expression `diff(g(..), x)` for the derivative of the subexpression.

The system always calls the `"diff"` slot with exactly two arguments. If you call the `diff` function with more indeterminates (for example, if you compute a higher derivative), then MuPAD calls the `"diff"` slot several times. Each call computes the derivative with respect to one indeterminate. The system caches the results of the calls of `"diff"` slots in `diff` in order to prevent redundant function calls. See “Example 11” on page 1-543.

Similarly, if an element `d` of a library domain `T` occurs as a subexpression of `f`, then `diff(f, x)` calls the slot `T::diff(d, x)` to compute the derivative of `d`.

If the domain `T` does not have a `"diff"` slot, then `diff` considers this object as a constant and returns 0 for the corresponding subexpression.

Examples

Example 1

Compute the derivative of x^2 with respect to `x`:

```
diff(x^2, x)
```

$$2\,x$$

Example 2

You can differentiate with respect to multiple variables within a single `diff` call. For example, differentiate this expression with respect to `x`, and then with differentiate the result with respect to `y`:

```
diff(x^2*sin(y), x, y) = diff(diff(x^2*sin(y), x), y)
```

$$2\,x\,\cos(y) = 2\,x\,\cos(y)$$

Example 3

MuPAD internally converts nested `diff` calls into a single `diff` call with multiple arguments:

```
diff(diff(f(x, y), x), y)
```

$$\frac{\partial}{\partial y}\,\frac{\partial}{\partial x}\,f(x, y)$$

Example 4

Use the sequence operator `$` as a shortcut to compute the third derivative of this expression with respect to `x`:

```
diff(sin(x)*cos(x), x $ 3)
```

$$4\sin(x)^2 - 4\cos(x)^2$$

Example 5

You can differentiate with respect to an indexed identifier. For example, differentiate this expression with respect to `x[1]`:

```
diff(x[1]*y + x[1]*x[r], x[1])
```

$$y + x_r + \delta_{r-1,0}\, x_1$$

Example 6

You can differentiate `polynomials` with respect to the polynomial indeterminates or the parameters in the coefficients. For example, differentiate this polynomial with respect to the indeterminate `x`:

```
diff(poly(sin(a)*x^3 + 2*x, [x]), x)
```

$$\mathrm{poly}((3\sin(a))\, x^2 + 2, [x])$$

Now differentiate the same polynomial with respect to its symbolic parameter `a`:

```
diff(poly(sin(a)*x^3 + 2*x, [x]), a)
```

$$\mathrm{poly}(\cos(a)\, x^3, [x])$$

Example 7

MuPAD returns the derivative of an unknown function as an unresolved `diff` call:

```
diff(f(x) + x, x)
```

$$\frac{\partial}{\partial x} f(x) + 1$$

Example 8

MuPAD assumes that all partial derivatives with respect to different indeterminates commute. Therefore, the system can change the order of indeterminates:

```
diff(f(x, y), x, y) = diff(f(x, y), y, x);
```

$$\frac{\partial}{\partial y}\frac{\partial}{\partial x} f(x, y) = \frac{\partial}{\partial y}\frac{\partial}{\partial x} f(x, y)$$

Example 9

You can use `diff` to differentiate symbolic integrals. For example, compute the second derivative of this indefinite integral:

```
F1 := int(f(x), x):
diff(F1, x, x)
```

$$\frac{\partial}{\partial x} f(x)$$

Now compute the derivative of the definite integral:

```
F2 := int(f(t, x), t = x..x^2):
diff(F2, x)
```

$$\int_x^{x^2} \frac{\partial}{\partial x} f(t, x)\, \mathrm{d}t - f(x, x) + 2\,x\, f(x^2, x)$$

Example 10

Use the operator `D` to compute the derivatives of functions. Use the `diff` function to compute the derivatives of expressions:

```
D(sin), diff(sin(x), x)
```

$$\cos, \cos(x)$$

Applying `D` to expressions and `diff` to functions makes no sense:

```
D(sin(x)), diff(sin, x)
```

$$(\sin(x))', 0$$

Use the `rewrite` function to rewrite an expression replacing the operator `D` with the `diff` function:

```
rewrite(D(f)(x), diff), rewrite(D(D(f))(x), diff)
```

$$\frac{\partial}{\partial x} f(x), \frac{\partial^2}{\partial x^2} f(x)$$

Also, use `rewrite` to rewrite an expression replacing `diff` with `D`:

```
diff(f(x, x), x) = rewrite(diff(f(x, x), x), D)
```

$$\frac{\partial}{\partial x} f(x, x) = \mathrm{D}_1(f)(x, x) + \mathrm{D}_2(f)(x, x)$$

Example 11

You can extend `diff` to your own special functions. To do so, embed your function, `f`, into a function environment, `g`, and implement the behavior of `diff` for this function as the `"diff"` slot of the function environment.

If a subexpression of the form `g(..)` occurs in an expression `f`, then `diff(f, x)` calls `g::diff(g(..), x)` to determine the derivative of the subexpression `g(..)`.

This example demonstrates extending `diff` to the exponential function. Since the function environment `exp` already has a `"diff"` slot, call the new function environment `Exp` to avoid overwriting the existing system function `exp`.

Here, the `"diff"` slot implements the chain rule for the exponential function. The derivative is the product of the original function call and the derivative of the argument:

```
Exp := funcenv(Exp):
Exp::diff := proc(f, x)
begin
```

```
   // f = Exp(something), i.e., something = op(f, 1)
   f*diff(op(f, 1), x):
end_proc:
diff(Exp(x^2), x)
```

$$2\,x\,\mathrm{Exp}(x^2)$$

The report created by `prog::trace` shows one call to `Exp::diff` with two arguments. Instead of calling `Exp::diff` twice, the system reads the required result of the second call from an internal cache for intermediate results in `diff`:

```
prog::trace(Exp::diff):
diff(Exp(x^2), x, x)
```

```
enter Exp::diff(Exp(x^2), x)
computed 2*x*Exp(x^2)
```

$$2\,\mathrm{Exp}(x^2) + 4\,x^2\,\mathrm{Exp}(x^2)$$

```
prog::untrace(Exp::diff):  delete f, Exp:
```

Parameters

`f`

An arithmetical expression or a polynomial of type `DOM_POLY`

`x, x1, x2, …`

Indeterminates: identifiers or indexed identifiers

Return Values

arithmetical expression or a polynomial

Overloaded By

f

See Also

MuPAD Functions
`D` | `int` | `limit` | `poly` | `taylor`

DIGITS

Significant digits of floating-point numbers

Description

The environment variable `DIGITS` determines the number of significant decimal digits in floating-point numbers. The default value is `DIGITS = 10`.

Possible values: a positive integer larger than 1 and smaller than $2^{29} + 1$.

Floating point numbers are created by applying the function `float` to exact numbers or numerical expressions. Elementary objects are approximated by the resulting floats with a relative precision of `10^(-DIGITS)`, i.e., the first `DIGITS` decimal digits are correct. Cf. “Example 1” on page 1-547.

In arithmetical operations with floating-point numbers, only the first `DIGITS` decimal digits are taken into account. The numerical error propagates and may grow in the course of computations. Cf. “Example 2” on page 1-548.

If a real floating-point number is entered directly (e.g., by `x := 1.234`), a number with at least `DIGITS` internal decimal digits is created.

If a real float is entered with more than `DIGITS` digits, the internal representation stores the extra digits. However, they are not taken into account in arithmetical operations, unless `DIGITS` is increased accordingly. Cf. “Example 3” on page 1-548.

In particular, complex floating-point numbers are created by adding the real and imaginary part. This addition truncates extra decimal places in the real and imaginary part.

The value of `DIGITS` may be changed at any time during a computation. If `DIGITS` is decreased, only the leading digits of existing floating numbers are taken into account in the following arithmetical operations. If `DIGITS` is increased, existing floating-point numbers are internally padded with trailing binary zeroes. Cf. “Example 4” on page 1-549.

Depending on `DIGITS`, certain functions such as the trigonometric functions may give wrong results if floats as arguments are too inaccurate. Cf. “Example 5” on page 1-549.

Depending on DIGITS, only significant digits of floating-point numbers are displayed on the screen. The preferences Pref::floatFormat and Pref::trailingZeroes can be used to modify the screen output. Cf. “Example 4” on page 1-549.

At least one digit after the decimal point is displayed; if it is insignificant, it is replaced by zero. Cf. “Example 6” on page 1-550.

Internally, floating-point numbers are created and stored with some extra “guard digits.” These are also taken into account by the basic arithmetical operations.

For example, for DIGITS = 10, the function float converts exact numbers to floats with some more decimal digits. The number of guard digits depends on DIGITS.

At least 2 internal guard digits are available for any value of DIGITS.

See “Example 4” on page 1-549 and “Example 7” on page 1-550.

Environment variables such as DIGITS are global variables. Upon return from a procedure that changes DIGITS, the new value is valid outside the context of the procedure as well! Use save DIGITS to restrict the modified value of DIGITS to the procedure. See “Example 8” on page 1-551.

The default value of DIGITS is 10; DIGITS has this value after starting or resetting the system via reset. Also the command delete DIGITS; restores the default value.

See the helppage of float for further information.

Examples

Example 1

We convert some exact numbers and numerical expressions to floating point approximations:

```
DIGITS := 10:
float(PI), float(1/7), float(sqrt(2) + exp(3)), float(exp(-20))
```

```
3.141592654, 0.1428571429, 21.49975049, 0.000000002061153622
```

```
DIGITS := 20:
```

```
float(PI), float(1/7), float(sqrt(2) + exp(3)), float(exp(-20))
```

```
3.1415926535897932385, 0.14285714285714285714, 21.49975048556076279,
  0.000000002061153622438557828
```

```
delete DIGITS:
```

Example 2

We illustrate error propagation in numerical computations. The following rational number approximates `exp(2)` to 17 decimal digits:

```
r := 738905609893065023/100000000000000000:
```

The following `float` call converts `exp(2)` and `r` to floating point approximations. The approximation errors propagate and are amplified in the following numerical expression:

```
DIGITS := 6: float(10^20*(r - exp(2)))
```

```
16777200.0
```

None of the digits in this result is correct. A better result is obtained by increasing `DIGITS`:

```
DIGITS := 20: float(10^20*(r - exp(2)))
```

```
276.95725393295288086
```

```
delete r, DIGITS:
```

Example 3

In the following, only 10 of the entered 30 digits are regarded as significant. The extra digits are stored internally, anyway:

```
DIGITS := 10:
a := 1.23456789666666666666666666666;
b := 1.23456789444444444444444444444
```

```
1.234567897
```

```
1.234567894
```

We increase `DIGITS`. Because the internal representation of `a` and `b` is correct to 30 decimal place, the difference can be computed correctly to 20 decimal places:

```
DIGITS := 30: a - b
```

```
0.000000002222222222222222222222000000001
```

```
delete a, b, DIGITS:
```

Example 4

We compute a floating-point number with a precision of 10 digits. Internally, this number is stored with some guard digits. Increasing `DIGITS` to 30, the correct guard digits become visible. With the the call `Pref::trailingZeroes(TRUE)`, trailing zeroes of the decimal representation become visible:

```
DIGITS := 10: a := float(1/9)
```

```
0.1111111111
```

```
0.11111111111111111101473758466795516142155975103378295898437500000000000000000000000
```

```
Pref::trailingZeroes(TRUE): DIGITS := 100: a
```

```
0.11111111111111111101473758466795516142155975103378295898437500000000000000000000000
```

```
Pref::trailingZeroes(FALSE): delete a, DIGITS:
```

Example 5

For the float evaluation of the sine function, the argument is reduced to the standard interval [0, 2 π]. For this reduction, the argument must be known to some digits after the decimal point. For small `DIGITS`, the digits after the decimal point are pure round-off if the argument is a large floating-point number:

```
DIGITS := 10: sin(float(2*10^30))
```

$$-0.6054240282$$

Increasing `DIGITS` to 50, the argument of the the sine function has about 30 correct digits after the decimal point. The first 30 digits of the following result are reliable:

```
DIGITS := 50: sin(float(2*10^30))
```

$$0.17950046751493908795061771231643809505098047699633$$

```
delete DIGITS:
```

Example 6

At least one digit after the decimal point is always displayed. In the following example, the number `39.9` is displayed as `40.0` because "`40.`" would not be a valid MuPAD input:

```
DIGITS := 2: float(10*PI), 39.9, -30.2
```

$$31.0,\ 40.0,\ -30.0$$

```
delete DIGITS:
```

Example 7

We compute `float(10^40*8/9)` with various values of `DIGITS`. Rounding takes into account all guard digits, i.e., the resulting integer makes all guard digits visible:

```
for DIGITS in [7, 8, 9, 17, 18, 19, 26, 27, 28] do
    print("DIGITS" = DIGITS, round(float(10^40*8/9)))
end_for:
```

$$\text{"DIGITS"} = 7,\ 88888888888888880843201856109995486085 12$$

$$\text{"DIGITS"} = 8,\ 88888888888888892932460052256287233146 88$$

$$\text{"DIGITS"} = 9,\ 88888888888888888398988228701427827998 72$$

"DIGITS" = 17, 88888888888888888888888884187865840353 28

"DIGITS" = 18, 88888888888888888888888884099904910131 2

"DIGITS" = 19, 88888888888888888888888893775607234560

"DIGITS" = 26, 88888888888888888888888888888888655872

"DIGITS" = 27, 88888888888888888888888888888888918016

"DIGITS" = 28, 88888888888888888888888888888888885248

Example 8

The following procedure allows to compute numerical approximations with a specified precision without changing `DIGITS` as a global variable. Internally, `DIGITS` is set to the desired precision and the float approximation is computed. Because of `save DIGITS`, the value of `DIGITS` is not changed outside the procedure:

```
myfloat := proc(x, digits)
           save DIGITS;
           begin
             DIGITS := digits:
             float(x);
           end_proc:
```

The float approximation of the following value `x` suffers from numerical cancellation. The procedure `myloat` is used to approximate `x` with 30 digits. The result is displayed with only 7 digits because of the value `DIGITS` = 7 valid outside the procedure. However, all displayed digits are correct:

```
x := PI^7 - exp(8013109200945801/1000000000000000):
DIGITS := 7:
float(x), myfloat(x, 30)
```

$$5.09317\ 10^{-11},\ 6.614563\ 10^{-13}$$

```
delete myfloat, x, DIGITS:
```

Algorithms

If a floating-point number `x` has been created with high precision, and the computation is to continue at a lower precision, the easiest method to get rid of memory-consuming insignificant digits is `x := x + 0.0`.

See Also

MuPAD Functions

`float` | `Pref::floatFormat` | `Pref::outputDigits` | `Pref::trailingZeroes`

dilog

Dilogarithm function

Syntax

```
dilog(x)
```

Description

`dilog(x)` represents the dilogarithm function $\int_1^x \frac{\ln(t)}{1-t}\,dt$.

If `x` is a floating-point number, then `dilog(x)` returns the numerical value of the dilogarithm function. The special values:

`dilog(-1)` = $\frac{\pi^2}{4} - i\,\pi\,\ln(2)$,

`dilog(0)` = $\frac{\pi^2}{6}$,

`dilog(1/2)` = $\frac{\pi^2}{12} - \frac{\ln(2)^2}{2}$,

`dilog(1)` = 0,

`dilog(2)` = $-\frac{\pi^2}{12}$,

`dilog(I)` = $\frac{\pi^2}{16} - i\,\mathrm{CATALAN} - \frac{i\,\pi\,\ln(2)}{4}$,

`dilog(-I)`= $\frac{\pi^2}{16} + i\,\mathrm{CATALAN} + \frac{i\,\pi\,\ln(2)}{4}$,

`dilog(1+I)` = $-\frac{\pi^2}{48} - i\,\mathrm{CATALAN}$,

`dilog(1-I)` = $-\frac{\pi^2}{48} + i\,\mathrm{CATALAN}$,

`dilog(infinity) = -infinity`

are implemented. For all other arguments, `dilog` returns a symbolic function call.

Functional identities are used to rewrite the result for exact numerical arguments of `Type::Numeric` that have a negative real part or are of absolute value larger than 1. Cf. "Example 2" on page 1-554.

`dilog(x)` coincides with `polylog(2, 1-x)`.

Environment Interactions

When called with a floating-point argument, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
dilog(0), dilog(2/3), dilog(sqrt(2)), dilog(1 + I), dilog(x)
```

$$\frac{\pi^2}{6},\ \mathrm{Li}_2\left(\frac{2}{3}\right),\ \mathrm{Li}_2(\sqrt{2}),\ -\frac{\pi^2}{48}-\mathrm{CATALAN}\,i,\ \mathrm{Li}_2(x)$$

Floating point values are computed for floating-point arguments:

```
dilog(-1.2), dilog(3.4 - 5.6*I)
```

$$2.458586602-2.477011851\,i,\ -2.529187195+2.25273709\,i$$

Example 2

Arguments built from integers and rational numbers are rewritten, if they lie in the left half of the complex plane or are of absolute value larger than 1. The following arguments have a negative real part:

```
dilog(-400/3), dilog(-1/2 + I)
```

$$\mathrm{Li}_2\left(\frac{3}{403}\right) - \ln\left(\frac{403}{3}\right)\left(\ln\left(\frac{400}{3}\right) + \pi\,i\right) + \frac{\pi^2}{6} + \frac{\ln\left(\frac{403}{3}\right)^2}{2},\ \mathrm{Li}_2\left(\frac{6}{13} + \frac{4\,i}{13}\right) - \ln\left(-\frac{1}{2} + i\right)\sigma_1 + \frac{\pi^2}{6} + \frac{{\sigma_1}^2}{2}$$

where

$$\sigma_1 = \ln\left(\frac{3}{2} - i\right)$$

The following arguments have an absolute value larger than 1:

```
dilog(31/30), dilog(1 + 2/3*I)
```

$$-\mathrm{Li}_2\left(\frac{30}{31}\right) - \frac{\ln\left(\frac{31}{30}\right)^2}{2},\ -\mathrm{Li}_2\left(\frac{9}{13} - \frac{6\,i}{13}\right) - \frac{\ln\left(1 + \frac{2\,i}{3}\right)^2}{2}$$

Example 3

The negative real axis is a branch cut of `dilog`. A jump of height $2\,\pi\,i\,ln(1 - x)$ occurs when crossing this cut at the real point $x < 0$:

```
dilog(-1.2), dilog(-1.2 + I/10^100), dilog(-1.2 - I/10^100)
```

$$2.458586602 - 2.477011851\,i,\ 2.458586602 - 2.477011851\,i,\ 2.458586602 + 2.477011851\,i$$

Example 4

The functions `diff`, `float`, `limit`, and `series` handle expressions involving `dilog`:

```
diff(dilog(x), x, x, x), float(ln(3 + dilog(sqrt(PI))))
```

$$\frac{2}{x\,(x-1)^2} + \frac{1}{x^2\,(x-1)} - \frac{2\ln(x)}{(x-1)^3},\ 0.8503829845$$

```
limit(dilog(x^10 + 1)/x, x = infinity)
```

```
0
```

```
series(dilog(x + 1/x)/x, x = -infinity, 3)
```

$$-\frac{\frac{\pi^2}{6}+\frac{(\ln(-x)+\pi\,i)^2}{2}}{x}+\frac{\ln(-x)+1+\pi\,i}{x^2}-\frac{\frac{\ln(-x)}{2}-\frac{1}{4}+\frac{\pi\,i}{2}}{x^3}+O\left(\frac{1}{x^4}\right)$$

Parameters

x

An arithmetical expression

Return Values

Arithmetical expression.

Overloaded By

x

Algorithms

`dilog(x)` coincides with $\sum_{k=1}^{\infty}\frac{(1-x)^k}{k^2}$ for $|x|<1$.

`dilog` has a branch cut along the negative real axis. The value at a point x on the cut coincides with the limit "from above":

$$\mathrm{Li}_2(x)=\lim_{\varepsilon\to 0+}x+\varepsilon\,i=\left(\lim_{\varepsilon\to 0-}x+\varepsilon\,i\right)-2\,\pi\,i\ln(1-x)$$

.

References

L. Lewin (ed.), “Structural Properties of Polylogarithms”, Mathematical Surveys and Monographs Vol. 37, American Mathematical Society, Providence (1991).

See Also

MuPAD Functions

`ln` | `polylog`

dirac

The Dirac delta distribution

Syntax

```
dirac(x)
```

```
dirac(x, n)
```

Description

`dirac(x)` represents the Dirac delta distribution.

`dirac(x, n)` represents the n-th derivative of the delta distribution.

The calls `dirac(x, 0)` and `dirac(x)` are equivalent.

If the argument `x` represents a non-zero number, then 0 is returned. If `x` is a non-real number of domain type `DOM_COMPLEX`, then `undefined` is returned. For all other arguments, a symbolic function call is returned.

`dirac` does not have a predefined value at the origin. Use

```
unprotect(dirac): dirac(0) := myValue:
```

and

```
dirac(float(0)) := myFloatValue: protect(dirac):
```

to assign a value (e.g., `infinity`).

For univariate linear expressions, the simplification rule

$$\delta^{(n)}(a\,x-b)=\frac{\operatorname{sign}(a)}{a^{n+1}}\,\delta^{(n)}\left(x-\frac{b}{a}\right)$$

is implemented for real numerical values `a`.

The integration function `int` treats `dirac` as the usual delta distribution. Cf. “Example 3” on page 1-560.

Environment Interactions

`dirac` reacts to properties of identifiers.

Examples

Example 1

`dirac` returns 0 for arguments representing non-zero real numbers:

```
dirac(-3), dirac(3/2), dirac(2.1, 1),
dirac(3*PI), dirac(sqrt(3), 3)
```

$$0,\ 0,\ 0.0,\ 0,\ 0$$

Arguments of domain type `DOM_COMPLEX` yield `undefined`:

```
dirac(1 + I), dirac(2/3 + 7*I), dirac(0.1*I, 1), dirac(ln(-5))
```

$$\text{undefined},\ \text{undefined},\ \text{undefined},\ \text{undefined}$$

A symbolic call is returned for other arguments:

```
dirac(0), dirac(x), dirac(x + I, 2), dirac(x, n)
```

$$\delta(0),\ \delta(x),\ \delta''(x+\mathrm{i}),\ \delta^{(n)}(x)$$

```
dirac(2*x - 1, n)
```

$$\frac{1}{2^{n+1}}\,\delta^{(n)}\left(x-\frac{1}{2}\right)$$

A natural value for `dirac(0)` is `infinity`:

```
unprotect(dirac): dirac(0) := infinity: dirac(0)
```

$$\infty$$

```
delete dirac(0): protect(dirac): dirac(0)
```

$$\delta(0)$$

Example 2

`dirac` reacts to assumptions set by `assume`:

```
assume(x < 0): dirac(x)
```

$$0$$

```
assume(x, Type::Real): assume(x <> 0, _and): dirac(x)
```

$$0$$

```
unassume(x):
```

Example 3

The symbolic integration function `int` treats `dirac` as the delta distribution:

```
int(f(x)*dirac(x - y^2), x = -infinity..infinity)
```

$$f(y^2)$$

```
int(int(f(x, y)*dirac(x - y^2), x = -infinity..infinity),
    y = -1..1)
```

$$\int_{-1}^{1} f(y^2, y)\,\mathrm{d}\,y$$

The indefinite integral of `dirac` involves the `sign` function:

```
int(f(x)*dirac(x), x), int(f(x)*dirac(x, 1), x)
```

$$\frac{f(0)\,\mathrm{sign}(x)}{2},\ f(0)\,\delta(x) - \frac{f'(0)\,\mathrm{sign}(x)}{2}$$

`int` can handle the distribution only if the argument of `dirac` is linear in the integration variable:

```
int(f(x)*dirac(2*x - 3), x = -10..10),
int(f(x)*dirac(x^2), x = -10..10)
```

$$\frac{f\left(\frac{3}{2}\right)}{2},\ \int_{-10}^{10} \delta(x^2)\,f(x)\,\mathrm{d}x$$

Also note that `dirac` should not be used for numerical integration, since the numerical algorithm will typically fail to detect the delta peak:

```
numeric::int(dirac(x - 3), x = -10..10)
```

$$0.0$$

Parameters

x

An arithmetical expression

n

An arithmetical expression representing a nonnegative integer

Return Values

Arithmetical expression.

Overloaded By

x

See Also

MuPAD Functions
`heaviside`

discont

Discontinuities of a function

Syntax

```
discont(f, x)

discont(f, x, <Undefined>)

discont(f, x, <Real>)

discont(f, x = a .. b)

discont(f, x = a .. b, <Undefined>)

discont(f, x = a .. b, <Real>)
```

Description

`discont(f, x)` computes the set of all discontinuities of the function $f(x)$.

`discont(f, x = a..b)` computes the set of all discontinuities of $f(x)$ lying in the interval $[a, b]$.

`discont(f, x)` returns a set of numbers containing all discontinuities of f when f is regarded as a function of x on the set of all complex numbers that may be attained by `x` as values, as specified by the assumptions on `x`. Please note that a real number that is a discontinuity of a complex function need not be a discontinuity of the restriction of that function to the set of real numbers: consider, for example, a function that has its branch cut on the real axis, as in "Example 2" on page 1-564 below.

Discontinuities include points where the function is not defined as well as points where the function is defined but not continuous. If the option `Undefined` is used, only points where the function is not defined are returned.

If the option `Real` is used, it is assumed that `f` and all of its subexpressions represent real numbers.

If a range `a..b` is given, it is assumed that `x` can take on values only in the interval $[a, b]$.

The set returned by `discont` may contain numbers that are not discontinuities of *f*. See “Example 7” on page 1-566.

If `discont` is unable to compute the discontinuities, then a symbolic `discont` call is returned; see “Example 8” on page 1-566.

`discont` can be extended to user-defined mathematical functions via overloading. To this end, embed the mathematical function in a function environment and assign the set of real discontinuities to its `"realDiscont"` slot, the set of its complex discontinuities to its `"complexDiscont"` slot, and the set of points where the function is not defined to its `"undefined"` slot. See `solve` for an overview of the various types of sets. See also “Example 8” on page 1-566 below.

Environment Interactions

`discont` reacts to properties of free parameters both in `f` as well as in `a` and `b`. `discont` sometimes reacts to properties of `x`.

Examples

Example 1

The `gamma` function has poles at all integers less or equal to zero. Hence `x -> gamma(x/2)` has poles at all even integers less or equal to zero:

```
discont(gamma(x/2), x)
```

$$(-\infty, 0] \cap \{2k \mid k \in \mathbb{Z}\}$$

Example 2

The logarithm has a branch cut on the negative real axis; hence, it is not continuous there. However, its restriction to the real numbers is continuous at every point except zero:

```
discont(ln(x), x), discont(ln(x), x, Real)
```

$$(-\infty, 0], \{0\}$$

Example 3

The function `sign` is defined everywhere; it is not continous at zero:

```
discont(sign(x), x), discont(sign(x), x, Undefined)
```

$$\{0\}, \varnothing$$

Example 4

If a range is given, only the discontinuities in that range are returned.

```
discont(1/x/(x - 1), x = 0..1/2)
```

$$\{0\}$$

Example 5

A range may have arbitrary arithmetical expressions as boundaries. `discont` implicitly assumes that the right boundary is greater or equal to the left boundary:

```
discont(1/x, x = a..b)
```

$$\begin{cases} \{0\} & \text{if } a \le 0 \wedge 0 \le b \\ \varnothing & \text{if } 0 < a \vee b < 0 \end{cases}$$

Example 6

As can be seen from the previous example, `discont` reacts to properties of free parameters (because `piecewise` does). The result also depends on the properties of `x`: it may omit values that `x` cannot take on anyway because of its properties.

```
assume(x > 0):
discont(1/x, x)
```

$$\varnothing$$

```
delete x:
```

Example 7

Sometimes, `discont` returns a proper superset of the set of discontinuities:

```
discont(piecewise([x<>0, x*sin(1/x)], [x=0, 0]), x)
```

$$\{0\}$$

Example 8

A symbolic `discont` call is returned if the system does not know how to determine the discontinuities of a given function:

```
delete f: discont(f(x), x)
```

$$\text{discont}(f(x), x)$$

You can provide the necessary information by adding slots to `f`. For example, assume that `f` is not continuous at `1` but everywhere else; and that also its restriction to the real numbers remains discontinuous at `1`. After adding the corresponding slots, `discont` takes care to handle `f` correctly also if it appears in a more complicated expression:

```
f:= funcenv(x->procname(x)):
f::realDiscont:= {1}:
f::complexDiscont:= {1}:
discont(f(sin(x)), x=-4..34)
```

$$\left\{\frac{\pi}{2}, \frac{5\,\pi}{2}, \frac{9\,\pi}{2}, \frac{13\,\pi}{2}, \frac{17\,\pi}{2}, \frac{21\,\pi}{2}\right\}$$

Example 9

We define a function that implements the logarithm to base 2. For simplicity, we let it always return the unevaluated function call. The logarithm has a branch cut on the negative real axis; its restriction to the reals is continuous everywhere except at zero:

```
binlog := funcenv(x -> procname(x)):
binlog::realDiscont := {0}:
binlog::undefined := {0}:
binlog::complexDiscont := Dom::Interval(-infinity, [0]):
discont(binlog(x), x);
discont(binlog(x), x=-2..2, Real);
discont(binlog(x), x=-2..2, Undefined)
```

$$(-\infty, 0]$$

$$\{0\}$$

$$\{0\}$$

Parameters

`f`

An arithmetical expression representing a function in `x`

`x`

An identifier

`a, b`

Interval boundaries: arithmetical expressions

Options

`Undefined`

Return only those points where `f` is not defined (and not just discontinous).

`Real`

Assume that all subexpressions of `f` are real.

Return Values

Set—see the help page for `solve` for an overview of all types of sets—or a symbolic `discont` call.

Overloaded By

f

See Also

MuPAD Functions
`limit` | `solve`

div, _div

Integer part of a quotient

Syntax

```
m div n
```

```
_div(x, m)
```

Description

`x div m` represents the integer q satisfying $x = q\,m + r$ with $0 \le r < |m|$.

For positive `x` and `m`, `q = x div m` is the integer part of the quotient `x/m`, i.e., `q = trunc(x/m)`.

`x div m` is equivalent to the function call `_div(x, m)`.

An integer is returned if both `x` and `m` evaluate to integers. A symbolic expression of type `"_div"` is returned if either `x` or `m` does not evaluate to a number. An error is raised if `x` or `m` evaluates to a number that is not an integer.

`div` does not operate on polynomials. Use `divide`.

Examples

Example 1

With the default setting for `mod`, the identity `(x div m) * m + (x mod m) = x` holds for integer numbers `x` and `m`:

```
43 div 13 = trunc(43/13), 43 mod 13 = frac(43/13) * 13
```

$$3 = 3,\ 4 = 4$$

```
(43 div 13) * 13 + (43 mod 13) = 43
```

$$43 = 43$$

Example 2

Symbolic expressions of type `"_div"` are returned, if either `x` or `m` does not evaluate to a number:

```
43 div m, x div 13, x div m
```

$$43 \operatorname{div} m,\ x \operatorname{div} 13,\ x \operatorname{div} m$$

```
type(x div m)
```

$$\text{"\_div"}$$

If `x` or `m` are numbers, they must be integer numbers:

```
1/2 div 2
```

```
Error: The argument is invalid. [div]
```

```
x div 2.0
```

```
Error: The argument is invalid. [div]
```

Parameters

`x, m`

Integers or symbolic arithmetical expressions; `m` must not be zero.

Return Values

Integer or an arithmetical expression of type `"_div"`.

Overloaded By

m, x

See Also

MuPAD Functions

`/` | `divide` | `mod` | `mod` | `modp` | `mods`

divergence

Divergence of a vector field

Syntax

```
divergence(v, x)
```

```
divergence(v, x, ogCoord, <c>)
```

Description

`divergence(v, x)` computes the divergence of the vector field $\vec{v}$ with respect to $\vec{x}$ in Cartesian coordinates. This is the sum $\mathrm{div}(\vec{v}) = \sum_{i=1}^{n} \frac{\partial}{\partial x_i} v_i$.

`ogCoord` can be the name of a three-dimensional orthogonal coordinate system predefined in the table `linalg::ogCoordTab`. See “Example 2” on page 1-573.

Alternatively, `ogCoord` can be a list of vector of algebraic expressions representing the scale factors of the coordinate system. See example “Example 3” on page 1-573. For details, see the description of the `Scales` option on the `linalg::ogCoordTab` page.

If `v` is a vector then the component ring of `v` must be a field (a domain of category `Cat::Field`) for which differentiation with respect to `x` is defined.

Examples

Example 1

Compute the divergence of the vector field $\vec{v}(x, y, z) = (x^2, 2\,y, z)$ in Cartesian coordinates:

```
delete x, y, z:
```

```
v := matrix([x^2, 2*y, z])
```

$$\begin{pmatrix} x^2 \\ 2\,y \\ z \end{pmatrix}$$

```
divergence(v, [x, y, z])
```

$$2\,x+3$$

Example 2

Compute the divergence of the vector field $\vec{v}(r, \phi, z) = (r, \cos(\phi), z)$ $(0 \le \phi < 2\,\pi)$ in cylindrical coordinates:

```
delete r, phi, z:
divergence([r, sin(phi), z], [r, phi, z], Cylindrical)
```

$$\frac{3\,r+\cos(\mathrm{phi})}{r}$$

The following relations between Cartesian and cylindrical coordinates hold:

$$x = r\,\cos(\phi),\ y = r\,\sin(\phi),\ z = z$$

.

Other predefined orthogonal coordinate systems can be found in the table `linalg::ogCoordTab`.

Example 3

Compute the divergence of a vector field in spherical coordinates r, θ, ϕ given by

$$\vec{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} r\,\sin(\theta)\,\cos(\phi) \\ r\,\sin(\theta)\,\sin(\phi) \\ r\,\cos(\theta) \end{pmatrix}$$

with $0 \le \theta \le \pi$, $0 \le \phi < 2\,\pi$. The vectors

$$\vec{e}_r = \frac{\frac{\partial}{\partial r}\vec{x}}{\left|\frac{\partial}{\partial r}\vec{x}\right|} = \begin{pmatrix} \sin(\theta)\cos(\phi) \\ \sin(\theta)\sin(\phi) \\ \cos(\theta) \end{pmatrix}, \quad \vec{e}_\theta = \frac{\frac{\partial}{\partial \theta}\vec{x}}{\left|\frac{\partial}{\partial \theta}\vec{x}\right|} = \begin{pmatrix} \cos(\theta)\cos(\phi) \\ \cos(\theta)\sin(\phi) \\ -\sin(\theta) \end{pmatrix},$$

$$\vec{e}_\phi = \frac{\frac{\partial}{\partial \phi}\vec{x}}{\left|\frac{\partial}{\partial \phi}\vec{x}\right|} = \begin{pmatrix} -\sin(\phi) \\ \cos(\phi) \\ 0 \end{pmatrix}$$

form an orthogonal system of unit vectors corresponding to the spherical coordinates. The scaling factors of the coordinate transformation (see `linalg::ogCoordTab`) are $\left|\frac{\partial}{\partial r}\vec{x}\right| = 1$, $\left|\frac{\partial}{\partial \theta}\vec{x}\right| = r$, $\left|\frac{\partial}{\partial \phi}\vec{x}\right| = r\sin(\theta)$, which we use in the following example to compute the divergence of the vector field $\vec{v}(r, \theta, \phi) = r^2\,\vec{e}_r$:

```
delete r, Theta, phi:
divergence([r^2, 0, 0], [r, Theta, phi], [1, r, r*sin(Theta)])
```

$$4\,r$$

Note that the spherical coordinates are already defined in `linalg::ogCoordTab`. The last result can also be achieved with the input `divergence([r^2, 0, 0], [r, Theta, phi], Spherical[RightHanded])`:

```
divergence([r^2, 0, 0], [r, Theta, phi], Spherical[RightHanded])
```

$$4\,r$$

Parameters

v

A list of arithmetical expressions, or a vector (i.e., an $n \times 1$ or $1 \times n$ matrix of a domain of category `Cat::Matrix`)

x

A list of identifiers or indexed identifiers

ogCoord

The name of a 3 dimensional orthogonal coordinate system predefined in the table `linalg::ogCoordTab`, or a list of algebraic expressions representing the scale factors of an orthogonal coordinate system.

c

The parameter of the coordinate systems EllipticCylindrical and Torus, respectively: an arithmetical expression. The default value is `c` = 1.

Return Values

Arithmetical expression, or an element of the component ring of `v`.

See Also

MuPAD Functions

`curl` | `gradient` | `laplacian` | `linalg::ogCoordTab` | `potential` | `vectorPotential`

divide

Divide polynomials

Syntax

```
divide(p, q, <[x]>, <order>, options)
```

```
divide(p, q, <[x1, x2, …]>, <order>, options)
```

```
divide(p, q1, q2, …, <order>, options)
```

Description

`divide(p, q)` divides polynomials or polynomial expressions `p` and `q`. By default, the function returns the quotient s and the remainder r, such that $p = s\,q + r$. Here $degree(r) < degree(q)$.

`divide`(p, q_1, q_2, q_3, …, q_n) divides a polynomial or a polynomial expression `p` by polynomials or polynomial expressions q_1, q_2, q_3, …, q_n. The function returns the quotients s_1, s_2, s_3, …, s_n and the remainder `r`, such that $p = s_1\,q_1 + s_2\,q_2 + s_3\,q_3 + \ldots + s_4\,q_4 + r$. Here the leading coefficient of the remainder `r` cannot be divided by the leading coefficients of any of divisors q_1, q_2, q_3, …, q_n.

`divide(p, q)` divides the polynomial or polynomial expression `p` by the polynomial or polynomial expression `q`. Use the `Quo` option to return the quotient only. Use the `Rem` option to return the remainder only.

The `divide` function operates on polynomials or polynomial expressions.

Polynomials must be of the same type: their variables and coefficient rings must be identical.

When you call `divide` for polynomial expressions, MuPAD internally converts these expressions to polynomials. See the `poly` function. If you do not specify the list of indeterminates, `divide` treats all symbolic variables in the expressions as indeterminates. If the expressions cannot be converted to polynomials, the `divide` function returns `FAIL`. See “Example 1” on page 1-577.

If you call `divide` for polynomials, it returns polynomials. If you call `divide` for polynomial expressions, the function returns polynomial expressions. See “Example 2” on page 1-578.

If you divide polynomial expressions that contain more than one variable, you can specify particular variables to be treated as variables. The `divide` function treats all other variables as symbolic parameters. By default, `divide` assumes that all variables in polynomial expressions are variables, and none of them is a symbolic parameter. See “Example 3” on page 1-578.

`divide`(p, q_1, q_2, q_3, …, q_n) divides a polynomial or a polynomial expression `p` by polynomials or polynomial expressions q_1, q_2, q_3, …, q_n. The function returns quotients s_1, s_2, s_3, …, s_n and remainder `r`, such that $p = s_1 q_1 + s_2 q_2 + s_3 q_3 + \ldots + s_4 q_4 + r$. Here the leading coefficient of the remainder `r` cannot be divided by the leading coefficients of any of the divisors q_1, q_2, q_3, …, q_n. See “Example 6” on page 1-580.

When dividing a polynomial by one or more polynomials, you can select the term ordering. The ordering accepts the following values:

- `LexOrder` sets the lexicographical ordering.
- `DegreeOrder` sets the total degree ordering. When using this ordering, MuPAD sorts the terms of a polynomial according to the total degree of each term (the sum of the exponents of the variables).
- `DegInvLexOrder` sets the total degree inverse lexicographic ordering. When using this ordering, MuPAD sorts the terms of a polynomial according to the total degree of each term (the sum of the exponents of the variables). If the several terms have equal total degrees, MuPAD sorts them using the inverse lexicographic ordering.
- your custom term ordering of type `Dom::MonomOrdering`.

The coefficient ring of the polynomials must implement the `"_divide"` method. MuPAD uses this method internally to divide coefficients. If the coefficients cannot be divided, this method must return `FAIL`.

Examples

Example 1

For polynomial expressions, `divide` internally calls the `poly` function, which converts an expression to a polynomial. If you do not specify the indeterminate of an expression,

MuPAD assumes that all variables are indeterminates. For example, The `divide` function cannot divide the following polynomial expressions because it assumes that both `x` and `y` are indeterminates:

```
divide(x/y, x)
```

FAIL

If you specify that only `x` is an indeterminate, the result is:

```
divide(x/y, x, [x])
```

$$\frac{1}{y}, 0$$

Example 2

The `divide` divides polynomials or polynomial expressions. When you divide polynomials, the function returns polynomials:

```
divide(poly(x^3 + x + 1, [x]), poly(x^2 + x + 1, [x]))
```

$$\mathrm{poly}(x-1, [x]), \mathrm{poly}(x+2, [x])$$

When you divide polynomial expressions, MuPAD internally converts these expressions to polynomials, divides these polynomials, and then converts the result of division to polynomial expressions:

```
divide(x^3 + x + 1, x^2 + x + 1)
```

$$x-1, x+2$$

Example 3

When dividing multivariate polynomials, you can specify the list of variables. The `divide` function assumes all other variables are symbolic parameters. For example,

divide the following two polynomial expressions specifying that both `x` and `y` are variables:

```
divide(x^2 - 2*x - y, y*x - 1, [x, y])
```

$$0, x^2 - 2x - y$$

Divide the same polynomial expressions specifying that only `x` is a variable. MuPAD assumes that `y` is a symbolic parameter:

```
divide(x^2 - 2*x - y, y*x - 1, [x])
```

$$\frac{\frac{1}{y} - 2}{y} + \frac{x}{y}, \frac{\frac{1}{y} - 2}{y} - y$$

Now, divide these expressions specifying that only `y` is a variable. MuPAD assumes that `x` is a symbolic parameter:

```
divide(x^2 - 2*x - y, y*x - 1, [y])
```

$$-\frac{1}{x}, x^2 - \frac{1}{x} - 2x$$

By default, the `divide` function treats polynomial expressions with more than one variable as multivariate polynomial expressions. The function does not assume that any of the variables are symbolic parameters:

```
divide(x^2 - 2*x - y, y*x - 1)
```

$$0, x^2 - 2x - y$$

Example 4

By default, `divide` returns the quotient and the remainder of the division of polynomials:

```
divide(x^3 + x + 1, x^2 + x + 1)
```

$$x-1, x+2$$

To return the quotient only, use the `Quo` option:

```
divide(x^3 + x + 1, x^2 + x + 1, Quo)
```

$$x-1$$

To return the remainder only, use the `Rem` option:

```
divide(x^3 + x + 1, x^2 + x + 1, Rem)
```

$$x+2$$

Example 5

Suppose, you want to get the result of the division only when the exact division is possible. To return the quotient `s` of the exact division of polynomials or polynomial expressions, use the `Exact` option:

```
divide(x^4 + 12*x^3 + 28*x^2 + 204*x + 187, x + 11, Exact)
```

$$x^3+x^2+17x+17$$

When exact division without remainder is impossible, the `divide` function with the `Exact` option returns `FAIL`:

```
divide(x^4 + 12*x^3 + 28*x^2 + 204*x + 187, x + 12, Exact)
```

FAIL

Example 6

The `divide` function allows you to divide a polynomial (or polynomial expression) by multiple polynomials (or polynomial expressions):

```
divide(4*x^4 + 2*x^2 + 1, x^3 - x + 1, x - 1)
```

$$4\,x,\ 6\,x+2,\ 3$$

When dividing a polynomial by multiple polynomials, you can select the term ordering:

```
divide(x^2+y^3+1, x-y^2, y, LexOrder)
```

$$y^2+x,\ y^3+y^2,\ 1$$

```
divide(x^2+y^3+1, x-y^2, y, DegreeOrder)
```

$$-y,\ x,\ x^2+1$$

Parameters

`p, q`

Univariate or multivariate polynomials or polynomial expressions.

$p, q_1, q_2, \ldots$

Univariate or multivariate polynomials or polynomial expressions.

`x`

The indeterminate of the polynomial: typically, an identifier or an indexed identifier. `divide` treats the expressions as univariate polynomials in the indeterminate `x`.

$x_1, x_2, \ldots$

The indeterminates of the polynomial: typically, identifiers or indexed identifiers. `divide` treats multivariate expressions as multivariate polynomials in these indeterminates.

`order`

The term ordering when dividing one multivariate polynomial by one or more multivariate polynomials: `LexOrder`, `DegreeOrder`, `DegInvLexOrder`, or a custom term ordering of type `Dom::MonomOrdering`. The default is the lexicographical ordering `LexOrder`.

Options

`Exact`

Return the quotient `s` of the exact division of multivariate polynomials. If no exact division without remainder is possible, return `FAIL`.

`Quo, Rem`

Return the quotient `s` or the remainder `r`. By default, the `divide` function returns both the quotient and the remainder.

Return Values

Polynomial, a polynomial expression, a sequence of polynomials or polynomial expressions, or the value `FAIL`.

Overloaded By

p, q

See Also

MuPAD Functions

`/` | `content` | `degree` | `div` | `factor` | `gcd` | `gcdex` | `groebner::normalf` | `ground` | `lcoeff` | `mod` | `multcoeffs` | `pdivide` | `poly` | `powermod`

domtype

Data type of an object

Syntax

```
domtype(object)
```

Description

`domtype(object)` returns the domain type (the data type) of the object.

For most data types, the domain type as returned by `domtype` coincides with the type returned by the function `type`. Only for expressions of domain type `DOM_EXPR`, the function `type` yields a distinction according to the 0-th operand. Cf. “Example 2” on page 1-584.

In contrast to most other functions, `domtype` does not flatten arguments that are expression sequences.

Examples

Example 1

Real floating-point numbers are of domain type `DOM_FLOAT`:

```
domtype(12.345)
```

```
DOM_FLOAT
```

Complex numbers are of domain type `DOM_COMPLEX`. The operands may be integers (`DOM_INT`), rational numbers (`DOM_RAT`), or floating-point numbers (`DOM_FLOAT`). The operands can be accessed via `op`:

```
domtype(1 - 2*I), op(1 - 2*I);
domtype(1/2 - I), op(1/2 - I);
```

```
domtype(2.0 - 3.0*I), op(2.0 - 3.0*I)
```

DOM_COMPLEX, $1, -2$

DOM_COMPLEX, $\frac{1}{2}, -1$

DOM_COMPLEX, $2.0, -3.0$

Example 2

Expressions are objects of the domain type `DOM_EXPR`. The type of expressions can be queried further with the function `type`:

```
domtype(x + y), type(x + y);
domtype(x - 1.0*I), type(x - 1.0*I);
domtype(x*I), type(x*I);
domtype(x^y), type(x^y);
domtype(x[i]), type(x[i])
```

DOM_EXPR, "_plus"

DOM_EXPR, "_plus"

DOM_EXPR, "_mult"

DOM_EXPR, "_power"

DOM_EXPR, "_index"

Example 3

`domtype` evaluates its argument. In this example, the assignment is first evaluated and `domtype` is applied to the return value of the assignment. This is the right hand side of the assignment, i.e., `5`:

```
domtype((a := 5))
```

```
DOM_INT
```

```
delete a:
```

Example 4

Here the identifier `a` is first evaluated to the `expression sequence3, 4`. Its domain type is `DOM_EXPR`, its type is `"_exprseq"`:

```
a := 3, 4: domtype(a), type(a)
```

```
DOM_EXPR, "_exprseq"
```

```
delete a:
```

Example 5

`factor` creates objects of the domain type `Factored`:

```
domtype(factor(x^2 - x))
```

```
Factored
```

Example 6

`matrix` creates objects of the domain type `Dom::Matrix()`:

```
domtype(matrix([[1, 2], [3, 4]]))
```

```
Dom::Matrix()
```

Example 7

Domains are of the domain type `DOM_DOMAIN`:

```
domtype(DOM_INT), domtype(DOM_DOMAIN)
```

```
DOM_DOMAIN, DOM_DOMAIN
```

Example 8

`domtype` is overloadable, i.e., a domain can pretend to be of another domain type. The special `slot"dom"` always gives the actual domain:

```
d := newDomain("d"): d::domtype := x -> "domain type d":
e := new(d, 1): e::dom, type(e), domtype(e)
```

```
d, d, "domain type d"
```

```
delete d, e:
```

Parameters

`object`

Any MuPAD object

Return Values

Data type, i.e., an object of type `DOM_DOMAIN`.

Overloaded By

object

See Also

MuPAD Domains
`DOM_DOMAIN`

MuPAD Functions
`coerce | hastype | testtype | type`

doprint

Print large matrices

Syntax

```
doprint(object1, object2, …)
```

Description

`doprint` serves for displaying large matrices on the screen. In fact, `doprint(object)` displays any MuPAD object like `print(object)`. The only difference is that large matrices contained in the object are printed, too.

Matrices of type `matrix` or of the more general type `Dom::Matrix(R)` with some coefficient ring `R` are not willing to print themselves on the screen if they are large.

An $m \times n$ matrix `A` is printed like a formatted two-dimensional array only if $m\,n \le printMaxSize$, where the default value of `printMaxSize` is 500. (You can change the `printMaxSize` value to any other integer value `m` by calling `A::dom::setPrintMaxSize(m)`).

For larger matrices, a warning is issued and some symbolic dummy object without the matrix entries is printed.

This serves to avoid output problems when printing is invoked accidentally (the output for large formatted arrays is very expensive concerning time and memory).

If you do insist on printing large matrices on the screen, the function `doprint` can be used to create a sparse table like output of the matrix.

Note: With `doprint`, only non-zero entries of large matrices are printed!

`doprint` allows to print arbitrary MuPAD objects. It behaves like `print` for all objects apart from matrices contained in the object.

For small matrices, it switches off the formatted array like output and replaces it by a sparse table like output. For large matrices, it suppresses warnings such as `"This matrix is too large for display. ..."` and prints matrices using the sparse table like output.

See “Example 1” on page 1-588 and “Example 2” on page 1-589.

Environment Interactions

`doprint` is sensitive to the environment variables `DIGITS`, `PRETTYPRINT`, and `TEXTWIDTH`, and to the output preferences `Pref::floatFormat`, `Pref::keepOrder`, and `Pref::trailingZeroes`.

Examples

Example 1

Small matrices are printed like formatted arrays:

```
A := matrix(5, 5, [i $ i = 1..30], Diagonal)
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 5 \end{pmatrix}$$

Calling `doprint`, this matrix is printed in a different way:

```
doprint(A)
```

$$\text{Dom::Matrix}()(5, 5, [(1, 1) = 1, (2, 2) = 2, (3, 3) = 3, (4, 4) = 4, (5, 5) = 5])$$

We create a larger diagonal matrix of dimension 30 ×30:

```
A := matrix(30, 30, [i $ i = 1..30], Diagonal):
```

If we had not suppressed the output by the colon terminating the command above, the following warning would have been issued by the output system:

```
A
```

```
Warning: This matrix is too large for display. To see all nonzero entries of a matrix A
```

Dom::Matrix()(30, 30, ["..."])

```
 Warning: This matrix is too
large for display. If you want to see all nonzero entries of a matrix,
say A, then call 'A::dom::doprint(A)'. [(Dom::Matrix(Dom::ExpressionField()))::print]

 Warning: This matrix is too large for display.
If you want to see all nonzero entries of a matrix, say A, then call
'A::dom::doprint(A)'. [(Dom::Matrix(Dom::ExpressionField()))::print]
```

Dom::Matrix()(30, 30, ["..."])

Since the matrix is extremely sparse, it does make sense to print the matrix. Calling `doprint`, we obtain a print output of all non-zero elements:

```
doprint(A)
```

Dom::Matrix()(30, 30, [(1, 1) = 1, (2, 2) = 2, (3, 3) = 3, (4, 4) = 4, (5, 5) = 5, (6, 6) = 6,
(7, 7) = 7, (8, 8) = 8, (9, 9) = 9, (10, 10) = 10, (11, 11) = 11, (12, 12) = 12, (13, 13) = 13,
(14, 14) = 14, (15, 15) = 15, (16, 16) = 16, (17, 17) = 17, (18, 18) = 18, (19, 19) = 19,
(20, 20) = 20, (21, 21) = 21, (22, 22) = 22, (23, 23) = 23, (24, 24) = 24, (25, 25) = 25,
(26, 26) = 26, (27, 27) = 27, (28, 28) = 28, (29, 29) = 29, (30, 30) = 30])

```
delete A:
```

Example 2

We compute a numerical QR factorization of a zero matrix of dimension 30×30. Since the command is not terminated by a colon, the output system tries to print the list with the factors Q and R. Both matrices send a warning:

```
[Q, R] := numeric::factorQR(matrix(30, 30, [])):
```

```
Q, R
```

```
Warning: This matrix is too large for display. To see all nonzero entries of a matrix A

Warning: This matrix is too large for display. To see all nonzero entries of a matrix A
```

Dom::Matrix()(30, 30, ["..."]), Dom::Matrix()(30, 30, ["..."])

```
 Warning: This matrix is too
large for display. If you want to see all nonzero entries of a matrix,
say A, then call 'A::dom::doprint(A)'. [(Dom::Matrix(Dom::ExpressionField()))::print]

 Warning: This matrix is too large for display.
If you want to see all nonzero entries of a matrix, say A, then call
'A::dom::doprint(A)'. [(Dom::Matrix(Dom::ExpressionField()))::print]

 Warning: This matrix is too large for display.
If you want to see all nonzero entries of a matrix, say A, then call
'A::dom::doprint(A)'. [(Dom::Matrix(Dom::ExpressionField()))::print]

 Warning: This matrix is too large for display.
If you want to see all nonzero entries of a matrix, say A, then call
'A::dom::doprint(A)'. [(Dom::Matrix(Dom::ExpressionField()))::print]
```

Dom::Matrix()(30, 30, ["..."]), Dom::Matrix()(30, 30, ["..."])

We can enforce a sparse output via `doprint`. The matrix factor Q is the identity matrix, the matrix factor R is zero:

```
doprint([Q, R])
```

[Dom::Matrix()(30, 30, [(1, 1) = 1.0, (2, 2) = 1.0, (3, 3) = 1.0, (4, 4) = 1.0, (5, 5) = 1.0,
(6, 6) = 1.0, (7, 7) = 1.0, (8, 8) = 1.0, (9, 9) = 1.0, (10, 10) = 1.0, (11, 11) = 1.0,
(12, 12) = 1.0, (13, 13) = 1.0, (14, 14) = 1.0, (15, 15) = 1.0, (16, 16) = 1.0, (17, 17) = 1.0,
(18, 18) = 1.0, (19, 19) = 1.0, (20, 20) = 1.0, (21, 21) = 1.0, (22, 22) = 1.0, (23, 23) = 1.0,
(24, 24) = 1.0, (25, 25) = 1.0, (26, 26) = 1.0, (27, 27) = 1.0, (28, 28) = 1.0, (29, 29) = 1.0,
(30, 30) = 1.0]), Dom::Matrix()(30, 30, [])]

```
delete Q, R:
```

Parameters

`object1, object2, …`

Any MuPAD objects

Return Values

`doprint` returns the void object `null()` of type `DOM_NULL`.

Overloaded By

See Also

MuPAD Functions

`DIGITS` | `expose` | `expr2text` | `fprint` | `Pref::floatFormat` | `Pref::keepOrder` | `Pref::trailingZeroes` | `PRETTYPRINT` | `print` | `protocol` | `strprint` | `TEXTWIDTH` | `write`

Ei

Exponential integral function

Syntax

```
Ei(x)
```

```
Ei(n, x)
```

Description

`Ei(x)` represents the exponential integral $\int_{-\infty}^{x} \frac{e^t}{t}\,dt$.

`Ei(n, x)` represents the exponential integral $\int_{1}^{\infty} \frac{e^{-x\,t}}{t^n}\,dt$.

If `x` is a floating-point number, then `Ei(x)` returns the numerical value of the exponential integral. The special values $Ei(\infty) = \infty$ and $Ei(-\infty) = 0$ are implemented. For all other arguments, `Ei(x)` returns a symbolic function call.

If both `n` and `x` are numerical values and if at least one of them is a floating-point number, then `Ei(n, x)` returns a floating-point value.

The special values $Ei(n, \infty) = 0$ and $Ei(n, -\infty) = -\infty$ are implemented for arbitrary `n`.

If `n` is a non-positive integer not larger than `Pref::autoExpansionLimit()`, then `Ei(n, x)` returns an explicit expression of the form `exp(-x)*p(1/x)`, where `p` is a polynomial of degree `1 - n`. E.g.:

$$\mathrm{Ei}(0, x) = \frac{e^{-x}}{x},\ \mathrm{Ei}(-1, x) = e^{-x}\left(\frac{1}{x} + \frac{1}{x^2}\right),\ \mathrm{Ei}(-2, x) = e^{-x}\left(\frac{1}{x} + \frac{2}{x^2} + \frac{2}{x^3}\right).$$

Use `expand` if such representations are also desired for `|n|` larger than `Pref::autoExpansionLimit()`.

If `x` is a positive constant, `Ei(1, x)` returns `- Ei(-x)`. For a negative constant `x`, `Ei(1, x)` returns `- Ei(-x) - π i`.

For all other arguments `Ei(n, x)` returns a symbolic function call.

Environment Interactions

When called with a floating-point argument, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
Ei(1), Ei(sqrt(2)), Ei(x + 1), Ei(infinity), Ei(-infinity)
```

$$\mathrm{Ei}(1),\ \mathrm{Ei}(\sqrt{2}),\ \mathrm{Ei}(x+1),\ \infty,\ 0$$

```
Ei(sqrt(2), PI), Ei(2, x + 1), Ei(3, infinity), Ei(I, -infinity)
```

$$\mathrm{Ei}(\sqrt{2},\ \pi),\ \mathrm{Ei}(2, x+1),\ 0,\ -\infty$$

If the first argument is a non-positive integer, an explicit expresssion is returned:

```
Ei(-5, x)
```

$$e^{-x}\left(\frac{1}{x}+\frac{5}{x^2}+\frac{20}{x^3}+\frac{60}{x^4}+\frac{120}{x^5}+\frac{120}{x^6}\right)$$

Floating point values are computed for floating-point arguments:

```
Ei(-1000.0), Ei(1.0), Ei(12.3), Ei(2.0 + 10.0*I)
```

$$-5.07089306\ 10^{-438},\ 1.895117816,\ 19643.40095,\ -0.4549646329 + 3.710255582\ i$$

```
Ei(3, -1000.0), Ei(1 + I, 1.0), Ei(-2, 12.3), Ei(1.0 + I, 2 + 10*I)
```

$$-1.976005087\,10^{431},\ 0.1866485916 - 0.08748205256\,i,$$

$$0.0000004351250372,\ 0.003718889079 + 0.01125612573\,i$$

For a positive constant `x`, `Ei(1, x)` returns `- Ei(-x)`. For a negative constant `x`, `Ei(1, x)` returns `- Ei(-x) - π i`:

```
Ei(1, 3), Ei(1, -3)
```

$$-\mathrm{Ei}(-3),\ -\mathrm{Ei}(3) - \pi\,i$$

Example 2

The 1-argument function `Ei(x)` is singular at the origin:

```
Ei(0)
```

```
Error: Singularity. [Ei]
```

The negative real axis is a branch cut. A jump of height $2\,\pi\,i$ occurs when crossing this cut:

```
Ei(-1.0), Ei(-1.0 + 10^(-10)*I), Ei(-1.0 - 10^(-10)*I)
```

$$-0.2193839344,\ -0.2193839344 + 3.141592654\,i,\ -0.2193839344 - 3.141592654\,i$$

Example 3

System functions such as `diff`, `float`, `limit`, `expand`, and `series` handle expressions involving `Ei`:

```
diff(Ei(x), x, x, x), float(ln(3 + Ei(sqrt(PI))))
```

$$\frac{e^x}{x} - \frac{2\,e^x}{x^2} + \frac{2\,e^x}{x^3},\ 1.968209389$$

```
diff(Ei(3, x), x, x, x), float(ln(3 + Ei(I, sqrt(PI))))
```

$$-\frac{e^{-x}}{x},\ 1.126485483 - 0.01132113135\,i$$

```
limit(Ei(2*x^2/(1+x)), x = infinity)
```

$$\infty$$

```
expand(Ei(3, x))
```

$$\frac{e^{-x}}{2} - \frac{x\,e^{-x}}{2} + \frac{x^2\,Ei(1, x)}{2}$$

```
series(Ei(3, x), x = 0, 3)
```

$$\frac{1}{2} - x - x^2\left(\frac{EULER}{2} + \frac{\ln(x)}{2} - \frac{3}{4}\right) + O(x^3)$$

```
series(Ei(7/2, x), x = infinity, 3)
```

$$\frac{e^{-x}}{x} - \frac{7\,e^{-x}}{2\,x^2} + \frac{63\,e^{-x}}{4\,x^3} + O\left(\frac{e^{-x}}{x^4}\right)$$

Parameters

n, x

arithmetical expressions

Return Values

Arithmetical expression.

Overloaded By

n, x

Algorithms

If `n` is a non-positive integer, then `Ei(n, x)` is an analytic function of `x` throughout the complex plane apart from a pole at the origin. For all other values of `n`, the function `Ei(n, x)` has a branch cut along the negative real semi axis, where the values coincide with the limit "from above":

$$\mathrm{Ei}(x) = \lim_{\varepsilon \to 0^+} \mathrm{Ei}(x + \varepsilon\, \mathrm{i})$$

for real $x < 0$.

The 1-argument function `Ei(x)` is related to the 2-argument function by

$$\mathrm{Ei}(x) = -\mathrm{Ei}(1, -x) + \frac{\ln(x) - \ln\left(\frac{1}{x}\right)}{2} - \ln(-x)$$

.

It has a logarithmic singularity at the origin and a branch cut along the negative real axis. Unlike the 2-argument function `Ei(n, x)` the 1-argument function $Ei(x)$ is not continuous from either above or below along the branch cut.

The functions `Ei(n, x)` are related to the incomplete gamma function `igamma` by $\mathrm{Ei}(n, x) = x^{n-1}\, \Gamma(1 - n, x)$.

The functions `Ei(x)` and `Ei(n, x)` correspond to the exponential integral functions $Ei(x)$ and $E_n(x)$ considered in M. Abramowitz and I. Stegun, "Handbook of Mathematical Functions", Dover Publications Inc., New York (1965).

See Also

MuPAD Functions

`Chi` | `Ci` | `exp` | `igamma` | `int` | `Li` | `Shi` | `Si` | `Ssi`

ellipticK

Complete elliptic integral of the first kind

Syntax

```
ellipticK(m)
```

Description

`ellipticK(m)` represents the complete elliptic integral of the first kind $\mathrm{K}(m)$ which is defined as

$$\mathrm{K}(m) = \mathrm{F}\left(\frac{\pi}{2} \mid m\right) = \int_0^{\frac{\pi}{2}} \frac{1}{\sqrt{1 - m\,\sin(\theta)^2}}\,\mathrm{d}\,\theta$$

.

The complete elliptic integral of the first kind is defined for a complex argument m.

A floating-point value is computed if all arguments are numerical and at least one is a floating-point value. Unevaluated symbolic calls are returned for most exact arguments. For some special cases explicit symbolic representations are returned.

Environment Interactions

When called with floating-point arguments, this function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Most calls with exact arguments return themselves unevaluated:

```
ellipticK(1/2); ellipticF(PI/4, I);
```

$$K\left(\frac{1}{2}\right)$$

$$F\left(\frac{\pi}{4} \mid i\right)$$

Some special arguments return explicit symbolic representations:

```
ellipticF(PI/2, 1/2); ellipticF(1, 1);
```

$$K\left(\frac{1}{2}\right)$$

$$\ln\left(\tan\left(\frac{\pi}{4}+\frac{1}{2}\right)\right)$$

If one argument is a floating-point value and the others can be converted to a floating-point values, then a floating-point result will be returned:

```
ellipticPi(0.5, PI/3, 1);
```

$$1.625993807$$

Parameters

m

An arithmetical expression specifying the parameter.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`ellipticCE` | `ellipticCK` | `ellipticCPi` | `ellipticE` | `ellipticF` | `ellipticPi`

ellipticCK

Complementary complete elliptic integral of the first kind

Syntax

```
ellipticCK(m)
```

Description

`ellipticCK(m)` represents the complementary complete elliptic integral of the first kind $\mathrm{K}'(m) = \mathrm{K}(1-m)$.

The complementary complete elliptic integral of the first kind is defined for a complex argument m.

A floating-point value is computed if all arguments are numerical and at least one is a floating-point value. Unevaluated symbolic calls are returned for most exact arguments. For some special cases explicit symbolic representations are returned.

Environment Interactions

When called with floating-point arguments, this function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Most calls with exact arguments return themselves unevaluated:

```
ellipticK(1/2); ellipticF(PI/4, I);
```

$$\mathrm{K}\left(\frac{1}{2}\right)$$

$$F\left(\frac{\pi}{4} \mid i\right)$$

Some special arguments return explicit symbolic representations:

```
ellipticF(PI/2, 1/2); ellipticF(1, 1);
```

$$K\left(\frac{1}{2}\right)$$

$$\ln\left(\tan\left(\frac{\pi}{4}+\frac{1}{2}\right)\right)$$

If one argument is a floating-point value and the others can be converted to a floating-point values, then a floating-point result will be returned:

```
ellipticPi(0.5, PI/3, 1);
```

$$1.625993807$$

Parameters

`m`

An arithmetical expression specifying the parameter.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`ellipticCE` | `ellipticCPi` | `ellipticE` | `ellipticF` | `ellipticK` | `ellipticPi`

ellipticF

Incomplete elliptic integral of the first kind

Syntax

```
ellipticF
 (φ, m)
```

Description

`ellipticF( φ ,m)` represents the incomplete elliptic integral of the first kind $F(\varphi \mid m)$ which is defined as

$$F(\varphi \mid m) = \int_0^{\varphi} \frac{1}{\sqrt{1 - m \sin(\theta)^2}} \, d\theta$$

.

The incomplete elliptic integral of the first kind is defined for complex arguments ϕ and m.

A floating-point value is computed if all arguments are numerical and at least one is a floating-point value. Unevaluated symbolic calls are returned for most exact arguments. For some special cases explicit symbolic representations are returned.

Environment Interactions

When called with floating-point arguments, this function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Most calls with exact arguments return themselves unevaluated:

```
ellipticK(1/2); ellipticF(PI/4, I);
```

$$K\left(\frac{1}{2}\right)$$

$$F\left(\frac{\pi}{4} \mid i\right)$$

Some special arguments return explicit symbolic representations:

```
ellipticF(PI/2, 1/2); ellipticF(1, 1);
```

$$K\left(\frac{1}{2}\right)$$

$$\ln\left(\tan\left(\frac{\pi}{4}+\frac{1}{2}\right)\right)$$

If one argument is a floating-point value and the others can be converted to a floating-point values, then a floating-point result will be returned:

```
ellipticPi(0.5, PI/3, 1);
```

$$1.625993807$$

Parameters

`m`

An arithmetical expression specifying the parameter.

`φ`

An arithmetical expression specifying the amplitude. In case of `ellipticE` and `ellipticPi`, the default is $\frac{\pi}{2}$.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`ellipticCE` | `ellipticCK` | `ellipticCPi` | `ellipticE` | `ellipticK` | `ellipticPi`

ellipticE

Complete and incomplete elliptic integrals of the second kind

Syntax

```
ellipticE
 (<ϕ>, m)
```

Description

`ellipticE(m)` represents the complete elliptic integral of the second kind $E(m)$ which is defined as

$$E(m) = E\left(\frac{\pi}{2} \mid m\right) = \int_0^{\frac{\pi}{2}} \sqrt{1 - m \sin(\theta)^2}\, d\theta$$

.

`ellipticE( φ ,m)` represents the incomplete elliptic integral of the second kind $E(\varphi \mid m)$ which is defined as

$$E(\varphi \mid m) = \int_0^{\varphi} \sqrt{1 - m \sin(\theta)^2}\, d\theta$$

.

The elliptic integrals of the second kind are defined for complex arguments ϕ and m.

A floating-point value is computed if all arguments are numerical and at least one is a floating-point value. Unevaluated symbolic calls are returned for most exact arguments. For some special cases explicit symbolic representations are returned.

Environment Interactions

When called with floating-point arguments, this function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Most calls with exact arguments return themselves unevaluated:

```
ellipticK(1/2); ellipticF(PI/4, I);
```

$$K\left(\frac{1}{2}\right)$$

$$F\left(\frac{\pi}{4} \mid i\right)$$

Some special arguments return explicit symbolic representations:

```
ellipticF(PI/2, 1/2); ellipticF(1, 1);
```

$$K\left(\frac{1}{2}\right)$$

$$\ln\left(\tan\left(\frac{\pi}{4}+\frac{1}{2}\right)\right)$$

If one argument is a floating-point value and the others can be converted to a floating-point values, then a floating-point result will be returned:

```
ellipticPi(0.5, PI/3, 1);
```

$$1.625993807$$

Parameters

`m`

An arithmetical expression specifying the parameter.

φ

An arithmetical expression specifying the amplitude. In case of `ellipticE` and `ellipticPi`, the default is $\frac{\pi}{2}$.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`ellipticCE` | `ellipticCK` | `ellipticCPi` | `ellipticF` | `ellipticK` | `ellipticPi`

ellipticCE

Complementary complete elliptic integral of the second kind

Syntax

```
ellipticCE(m)
```

Description

`ellipticCE(m)` represents the complementary complete elliptic integral of the second kind $\mathrm{E}'(m) = \mathrm{E}(1-m)$.

The complementary complete elliptic integral of the second kind is defined for a complex argument m.

A floating-point value is computed if all arguments are numerical and at least one is a floating-point value. Unevaluated symbolic calls are returned for most exact arguments. For some special cases explicit symbolic representations are returned.

Environment Interactions

When called with floating-point arguments, this function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Most calls with exact arguments return themselves unevaluated:

```
ellipticK(1/2); ellipticF(PI/4, I);
```

$$\mathrm{K}\left(\frac{1}{2}\right)$$

$$F\left(\frac{\pi}{4} \mid i\right)$$

Some special arguments return explicit symbolic representations:

```
ellipticF(PI/2, 1/2); ellipticF(1, 1);
```

$$K\left(\frac{1}{2}\right)$$

$$\ln\left(\tan\left(\frac{\pi}{4}+\frac{1}{2}\right)\right)$$

If one argument is a floating-point value and the others can be converted to a floating-point values, then a floating-point result will be returned:

```
ellipticPi(0.5, PI/3, 1);
```

$$1.625993807$$

Parameters

`m`

An arithmetical expression specifying the parameter.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`ellipticCK` | `ellipticCPi` | `ellipticE` | `ellipticF` | `ellipticK` | `ellipticPi`

ellipticPi

Complete and incomplete elliptic integrals of the third kind

Syntax

```
ellipticPi(n, <φ>, m)
```

Description

`ellipticPi(n,m)` represents the complete elliptic integral of the third kind

$$\Pi(n \mid m) = \Pi\left(n; \frac{\pi}{2} \mid m\right) = \int_0^{\frac{\pi}{2}} \frac{1}{(1 - n \sin(\theta)^2)\sqrt{1 - m \sin(\theta)^2}}\, d\theta$$

`ellipticPi(n, φ ,m)` represents the incomplete elliptic integral of the third kind

$$\Pi(n; \varphi \mid m) = \int_0^{\varphi} \frac{1}{(1 - n \sin(\theta)^2)\sqrt{1 - m \sin(\theta)^2}}\, d\theta$$

The elliptic integrals of the third kind are defined for complex arguments m, ϕ, and n.

A floating-point value is computed if all arguments are numerical and at least one is a floating-point value. Unevaluated symbolic calls are returned for most exact arguments. For some special cases explicit symbolic representations are returned.

Environment Interactions

When called with floating-point arguments, this function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Most calls with exact arguments return themselves unevaluated:

```
ellipticK(1/2); ellipticF(PI/4, I);
```

$$K\left(\frac{1}{2}\right)$$

$$F\left(\frac{\pi}{4} \mid i\right)$$

Some special arguments return explicit symbolic representations:

```
ellipticF(PI/2, 1/2); ellipticF(1, 1);
```

$$K\left(\frac{1}{2}\right)$$

$$\ln\left(\tan\left(\frac{\pi}{4}+\frac{1}{2}\right)\right)$$

If one argument is a floating-point value and the others can be converted to a floating-point values, then a floating-point result will be returned:

```
ellipticPi(0.5, PI/3, 1);
```

$$1.625993807$$

Parameters

`m`

An arithmetical expression specifying the parameter.

φ

An arithmetical expression specifying the amplitude. In case of `ellipticE` and `ellipticPi`, the default is $\frac{\pi}{2}$.

n

An arithmetical expression specifying the characteristic.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`ellipticCE` | `ellipticCK` | `ellipticCPi` | `ellipticE` | `ellipticF` | `ellipticK`

ellipticCPi

Complementary complete elliptic integral of the third kind

Syntax

```
ellipticCPi(n, m)
```

Description

`ellipticCPi(n,m)` represents the complementary complete elliptic integral of the third kind $\Pi'(n \mid m) = \Pi(n \mid 1 - m)$.

The complementary complete elliptic integral of the third kind is defined for complex arguments m and n.

A floating-point value is computed if all arguments are numerical and at least one is a floating-point value. Unevaluated symbolic calls are returned for most exact arguments. For some special cases explicit symbolic representations are returned.

Environment Interactions

When called with floating-point arguments, this function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Most calls with exact arguments return themselves unevaluated:

```
ellipticK(1/2); ellipticF(PI/4, I);
```

$$\mathrm{K}\left(\frac{1}{2}\right)$$

$$F\left(\frac{\pi}{4} \mid i\right)$$

Some special arguments return explicit symbolic representations:

```
ellipticF(PI/2, 1/2); ellipticF(1, 1);
```

$$K\left(\frac{1}{2}\right)$$

$$\ln\left(\tan\left(\frac{\pi}{4}+\frac{1}{2}\right)\right)$$

If one argument is a floating-point value and the others can be converted to a floating-point values, then a floating-point result will be returned:

```
ellipticPi(0.5, PI/3, 1);
```

$$1.625993807$$

Parameters

`m`

An arithmetical expression specifying the parameter.

`n`

An arithmetical expression specifying the characteristic.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`ellipticCE` | `ellipticCK` | `ellipticE` | `ellipticF` | `ellipticK` | `ellipticPi`

ellipticNome

Elliptic nome

Syntax

```
ellipticNome
 (m)
```

Description

`ellipticNome(m)` represents the elliptic nome q which is defined as

$$\mathrm{q}(m) = \mathrm{e}^{-\frac{\pi \mathrm{K}'(m)}{\mathrm{K}(m)}}.$$

The elliptic nome $\mathrm{q}(m)$ is defined for complex arguments m.

$|\mathrm{q}(m)| \le 1$ holds for all $m \in \mathbb{C}$.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

For most exact arguments, `ellipticNome` will return unevaluated:

```
ellipticNome(1/3); ellipticNome(2); ellipticNome(I);
```

$$q\left(\frac{1}{3}\right)$$

$$q(2)$$

$$q(i)$$

For 0, $\frac{1}{2}$ and 1 an explicit result is returned:

```
ellipticNome(0); ellipticNome(1/2); ellipticNome(1);
```

$$0$$

$$e^{-\pi}$$

$$1$$

If the argument is a floating-point number, a floating-point result will be returned:

```
ellipticNome(0.5)
```

$$0.04321391826$$

Using `float`, floating-point evaluation can be enforced:

```
float(ellipticNome(3/4))
```

$$0.0857957337$$

Parameters

`m`

An arithmetical expression specifying the parameter.

Return Values

Arithmetical expression.

See Also

MuPAD Functions
`ellipticCK` | `ellipticK`

end

Close a block statement

Description

`end` is a keyword which, depending on the context, is parsed as one of the following keywords:

- `end_case`
- `end_for`
- `end_if`
- `end_proc`
- `end_repeat`
- `end_while`

Examples

Example 1

Each of the keywords `proc`, `case`, `if`, `for`, `repeat`, and `while` starts some block construct in the MuPAD language. Each block can be closed with `end` or with the corresponding special keyword `end_proc`, `end_case` etc.:

```
f :=
proc(a, b)
  local i;
begin
  for i from a to b do
    if isprime(i) then
       print(Unquoted, expr2text(i)." is a prime")
    end
  end
end:

f(20, 30):
```

```
23 is a prime


29 is a prime
```

The parser translates `end` to the appropriate keyword matching the type of the block:

```
expose(f)


proc(a, b)
  name f;
  local i;
begin
  for i from a to b do
    if isprime(i) then
      print(Unquoted, expr2text(i)." is a prime")
    end_if
  end_for
end_proc

delete f:
```

See Also

MuPAD Functions

`end_case` | `end_for` | `end_if` | `end_proc` | `end_repeat` | `end_while`

erf

Error function

Syntax

```
erf(x)
```

Description

`erf(x)` represents the error function $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$.

This function is defined for all complex arguments `x`.

For floating-point arguments the error functions `erf`, `erfc`, and `erfi` return floating-point values. The implemented exact values are:

$erf(0) = 0$, $erf(\infty) = 1$, $erf(-\infty) = -1$, $erf(i\infty) = i\infty$, $erf(-i\infty) = -i\infty$,

For all other arguments, the error function returns symbolic function calls.

For the function `erf` with floating-point arguments of large absolute value, internal numerical underflow or overflow can happen.

The error functions $erf(x) = 1 - erfc(x)$ and $erfi(x) = i\,(erfc(x\,i) - 1)$ return corresponding values for large arguments. See “Example 2” on page 1-623.

MuPAD can simplify expressions that contain error functions and their inverses. For real values `x`, the system applies the following simplification rules:

$inverf(erf(x)) = inverf(1 - erfc(x)) = inverfc(1 - erf(x)) = inverfc(erfc(x)) = x$,

$inverf(-erf(x)) = inverf(erfc(x) - 1) = inverfc(1 + erf(x)) = inverfc(2 - erfc(x)) = -x$

For any value `x`, the system applies the following simplification rules:

$inverf(-x) = -inverf(x)$,

$inverfc(2 - x) = - inverfc(x)$,

$erf(inverf(x)) = erfc(inverfc(x)) = x$.

$erf(inverfc(x)) = erfc(inverf(x)) = 1 - x$.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

You can call error functions with exact and symbolic arguments:

```
erf(0), erf(3/2), erf(sqrt(2)), erf(infinity)
```

$$0,\ \mathrm{erf}\left(\frac{3}{2}\right),\ \mathrm{erf}(\sqrt{2}),\ 1$$

```
erfc(0), erfc(x + 1), erfc(-infinity)
```

$$1,\ \mathrm{erfc}(x+1),\ 2$$

```
erfc(0, n), erfc(x + 1, -1), erfc(-infinity, 5)
```

$$\frac{1}{2^n\,\Gamma\left(\frac{n}{2}+1\right)},\ \frac{2\,e^{-(x+1)^2}}{\sqrt{\pi}},\ \infty$$

```
erfi(0), erfi(x + 1), erfi(-infinity)
```

$$0,\ \mathrm{erfi}(x+1),\ -\infty$$

```
inverf(-1), inverf(0), inverf(1), inverf(x + 1), inverf(1/5)
```

$$-\infty,\ 0,\ \infty,\ \text{inverf}(x+1),\ \text{inverf}\left(\frac{1}{5}\right)$$

```
inverfc(0), inverfc(1), inverfc(2), inverfc(15), inverfc(x/5)
```

$$\infty,\ 0,\ -\infty,\ \text{inverfc}(15),\ \text{inverfc}\left(\frac{x}{5}\right)$$

For floating-point arguments, the error functions return floating-point values:

```
erf(-7.2), erfc(2.0 + 3.5*I), erfc(3.0, 4), erfi(5.5 + 1.0*I)
```

$$-1.0,\ -420.8123327 - 343.6612334\ i,$$

$$0.000000009433438115,\ -93412361266.0 - 5.089108513\ 10^{11}\ i$$

For floating-point arguments `x` from the interval `[-1, 1]`, `inverf` returns floating-point values:

```
inverf(-0.5), inverf(0.85)
```

$$-0.4769362762,\ 1.017902465$$

For floating-point arguments outside of this interval, `inverf` returns symbolic function calls:

```
inverf(-5.3), inverf(10.0)
```

$$-\text{inverf}(5.3),\ \text{inverf}(10.0)$$

For floating-point arguments `x` from the interval `[0, 2]`, `inverfc` returns floating-point values:

```
inverfc(0.5), inverfc(1.25)
```

$$0.4769362762,\ -0.225312055$$

For floating-point arguments outside of this interval, `inverfc` returns symbolic function calls:

```
inverfc(-1.25), inverfc(2.5)
```

$$-\text{inverfc}(3.25),\ \text{inverfc}(2.5)$$

Example 2

For large complex arguments, the error functions can return NaN:

```
erf(38000.0 + 3801.0*I),
erfi(38000.0 + 3801.0*I),
erfc(38000.0 + 3801.0*I)
```

$$\text{NaN},\ \text{NaN},\ \text{NaN}$$

For large floating-point arguments with positive real parts, `erfc` can return values truncated to `0.0`:

```
erfc(27281.1), erfc(27281.2)
```

$$4.085187475\ 10^{-323227329},\ 0.0$$

Example 3

The functions `diff`, `float`, `limit`, `expand`, `rewrite`, and `series` handle expressions involving the error functions:

```
diff(erf(x), x, x, x)
```

$$\frac{8\,x^2\,e^{-x^2}}{\sqrt{\pi}} - \frac{4\,e^{-x^2}}{\sqrt{\pi}}$$

```
diff(erfc(x, 3), x, x)
```

$$\text{erfc}(x,\ 1)$$

```
diff(inverf(x), x)
```

$$\frac{\sqrt{\pi}\, e^{\mathrm{inverf}(x)^2}}{2}$$

```
float(ln(3 + erfi(sqrt(PI)*I)))
```

$$1.150079617 + 0.3180894436\,i$$

```
limit(x/(1 + x)*erf(x), x = infinity)
```

$$1$$

```
expand(erfc(x, 3))
```

$$\frac{e^{-x^2}}{6\sqrt{\pi}} - \frac{x^3\,\mathrm{erfc}(x)}{6} - \frac{x\,\mathrm{erfc}(x)}{4} + \frac{x^2\, e^{-x^2}}{6\sqrt{\pi}}$$

```
rewrite(inverfc(x), inverf)
```

$$-\mathrm{inverf}(x-1)$$

```
series(erf(x), x = infinity, 3)
```

$$1 - \frac{e^{-x^2}}{\sqrt{\pi}\,x} + \frac{e^{-x^2}}{2\sqrt{\pi}\,x^3} + O\left(\frac{e^{-x^2}}{x^4}\right)$$

```
series(erfc(x), x = infinity, 3)
```

$$\frac{e^{-x^2}}{\sqrt{\pi}\,x} - \frac{e^{-x^2}}{2\sqrt{\pi}\,x^3} + O\left(\frac{e^{-x^2}}{x^4}\right)$$

```
series(erfi(x), x = I*infinity, 3)
```

$$i + \frac{e^{x^2}}{\sqrt{\pi}\,x} + \frac{e^{x^2}}{2\sqrt{\pi}\,x^3} + O\left(\frac{e^{x^2}}{x^4}\right)$$

Parameters

x

An arithmetical expression

Return Values

Arithmetical expression

Algorithms

`erf`, `erfc`, and `erfi` are entire functions.

See Also

MuPAD Functions
`erfc` | `erfi` | `inverf` | `inverfc` | `stats::normalQuantile`

More About

- "Error Functions and Fresnel Functions"

erfc

Complementary error function

Syntax

```
erfc(x)
erfc(x, n)
```

Description

$\mathrm{erfc}(x) = 1 - \mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2}\, dt$ computes the complementary error function.

$\mathrm{erfc}(x, n) = \int_x^\infty \mathrm{erfc}(t, n-1)\, dt$ with $erfc(x, 0) = erfc(x)$ and $\mathrm{erfc}(x, -1) = \frac{2}{\sqrt{\pi}} e^{-x^2}$ returns the iterated integrals of the complementary error function.

This function is defined for all complex arguments `x`.

For floating-point arguments the error functions `erf`, `erfc`, and `erfi` return floating-point values. The implemented exact values are:

$erfc(0) = 1$, $erfc(\infty) = 0$, $erfc(-\infty) = 2$, $erfc(i\infty) = 1 - i\infty$, $erfc(-i\infty) = 1 + i\infty$,

$\mathrm{erfc}(0, n) = \frac{1}{2^n \Gamma\left(\frac{n}{2}+1\right)}$, $erfc(\infty, n) = 0$, $erfc(-\infty, n) = \infty$,

For all other arguments, the error function returns symbolic function calls.

The calls `erfc(x)` and `erfc(x, 0)` are equivalent.

If a numerical value of `n` is not an integer or if `n < -1`, the function call `erfc(x, n)` returns an error. The function also accepts symbolic values of `n`.

If `n` is a numerical value, you can use the `expand(erfc(x, n))` command to apply:

- The recurrence $\mathrm{erfc}(x, n) = -\frac{x\, \mathrm{erfc}(x, n-1)}{n} + \frac{\mathrm{erfc}(x, n-2)}{2n}$

- The reflection rule $\mathrm{erfc}(-x, n) = (-1)^{n+1}\,\mathrm{erfc}(x, n) + \frac{H(n, \mathrm{i}\,x)}{\mathrm{i}^n\, 2^{n-1}\, n!}$, where H = `orthpoly::hermite`

See “Example 3” on page 1-630.

For the function `erfc` with floating-point arguments of large absolute value, internal numerical underflow or overflow can happen. If a call to `erfc` causes underflow or overflow, this function returns:

- The result truncated to `0.0` if `x` is a large positive real number
- The result rounded to `2.0` if `x` is a large negative real number
- `RD_NAN` if `x` is a large complex number and MuPAD cannot approximate the function value

The error functions *erf*(*x*) = 1 - *erfc*(*x*) and *erfi*(*x*) = i (*erfc*(*x* i) - 1) return corresponding values for large arguments. See “Example 2” on page 1-629.

MuPAD can simplify expressions that contain error functions and their inverses. For real values `x`, the system applies the following simplification rules:

inverf(*erf*(*x*)) = *inverf*(1 - *erfc*(*x*)) = *inverfc*(1 - *erf*(*x*)) = *inverfc*(*erfc*(*x*)) = *x*,

inverf(- *erf*(*x*)) = *inverf*(*erfc*(*x*) - 1) = *inverfc*(1 + *erf*(*x*)) = *inverfc*(2 - *erfc*(*x*)) = - *x*

For any value `x`, the system applies the following simplification rules:

inverf(- *x*) = - *inverf*(*x*),

inverfc(2 - *x*) = - *inverfc*(*x*),

erf(*inverf*(*x*)) = *erfc*(*inverfc*(*x*)) = *x*.

erf(*inverfc*(*x*)) = *erfc*(*inverf*(*x*)) = 1 - *x*.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

You can call error functions with exact and symbolic arguments:

```
erf(0), erf(3/2), erf(sqrt(2)), erf(infinity)
```

$$0,\ \operatorname{erf}\left(\frac{3}{2}\right),\ \operatorname{erf}(\sqrt{2}),\ 1$$

```
erfc(0), erfc(x + 1), erfc(-infinity)
```

$$1,\ \operatorname{erfc}(x+1),\ 2$$

```
erfc(0, n), erfc(x + 1, -1), erfc(-infinity, 5)
```

$$\frac{1}{2^n\,\Gamma\left(\frac{n}{2}+1\right)},\ \frac{2\,e^{-(x+1)^2}}{\sqrt{\pi}},\ \infty$$

```
erfi(0), erfi(x + 1), erfi(-infinity)
```

$$0,\ \operatorname{erfi}(x+1),\ -\infty$$

```
inverf(-1), inverf(0), inverf(1), inverf(x + 1), inverf(1/5)
```

$$-\infty,\ 0,\ \infty,\ \operatorname{inverf}(x+1),\ \operatorname{inverf}\left(\frac{1}{5}\right)$$

```
inverfc(0), inverfc(1), inverfc(2), inverfc(15), inverfc(x/5)
```

$$\infty,\ 0,\ -\infty,\ \operatorname{inverfc}(15),\ \operatorname{inverfc}\left(\frac{x}{5}\right)$$

For floating-point arguments, the error functions return floating-point values:

```
erf(-7.2), erfc(2.0 + 3.5*I), erfc(3.0, 4), erfi(5.5 + 1.0*I)
```

$$-1.0,\ -420.8123327 - 343.6612334\,i,$$

$$0.000000009433438115,\ -93412361266.0 - 5.089108513\,10^{11}\,i$$

For floating-point arguments `x` from the interval `[-1, 1]`, `inverf` returns floating-point values:

```
inverf(-0.5), inverf(0.85)
```

$$-0.4769362762,\ 1.017902465$$

For floating-point arguments outside of this interval, `inverf` returns symbolic function calls:

```
inverf(-5.3), inverf(10.0)
```

$$-\mathrm{inverf}(5.3),\ \mathrm{inverf}(10.0)$$

For floating-point arguments `x` from the interval `[0, 2]`, `inverfc` returns floating-point values:

```
inverfc(0.5), inverfc(1.25)
```

$$0.4769362762,\ -0.225312055$$

For floating-point arguments outside of this interval, `inverfc` returns symbolic function calls:

```
inverfc(-1.25), inverfc(2.5)
```

$$-\mathrm{inverfc}(3.25),\ \mathrm{inverfc}(2.5)$$

Example 2

For large complex arguments, the error functions can return NaN:

```
erf(38000.0 + 3801.0*I),
erfi(38000.0 + 3801.0*I),
```

```
erfc(38000.0 + 3801.0*I)
```

$$\mathrm{NaN},\ \mathrm{NaN},\ \mathrm{NaN}$$

For large floating-point arguments with positive real parts, `erfc` can return values truncated to `0.0`:

```
erfc(27281.1), erfc(27281.2)
```

$$4.085187475\ 10^{-323227329},\ 0.0$$

Example 3

The functions `diff`, `float`, `limit`, `expand`, `rewrite`, and `series` handle expressions involving the error functions:

```
diff(erf(x), x, x, x)
```

$$\frac{8\,x^2\,e^{-x^2}}{\sqrt{\pi}} - \frac{4\,e^{-x^2}}{\sqrt{\pi}}$$

```
diff(erfc(x, 3), x, x)
```

$$\mathrm{erfc}(x,\ 1)$$

```
diff(inverf(x), x)
```

$$\frac{\sqrt{\pi}\,e^{\mathrm{inverf}(x)^2}}{2}$$

```
float(ln(3 + erfi(sqrt(PI)*I)))
```

$$1.150079617 + 0.3180894436\,i$$

```
limit(x/(1 + x)*erf(x), x = infinity)
```

$$1$$

```
expand(erfc(x, 3))
```

$$\frac{e^{-x^2}}{6\sqrt{\pi}} - \frac{x^3\,\mathrm{erfc}(x)}{6} - \frac{x\,\mathrm{erfc}(x)}{4} + \frac{x^2\,e^{-x^2}}{6\sqrt{\pi}}$$

```
rewrite(inverfc(x), inverf)
```

$$-\mathrm{inverf}(x-1)$$

```
series(erf(x), x = infinity, 3)
```

$$1 - \frac{e^{-x^2}}{\sqrt{\pi}\,x} + \frac{e^{-x^2}}{2\sqrt{\pi}\,x^3} + O\left(\frac{e^{-x^2}}{x^4}\right)$$

```
series(erfc(x), x = infinity, 3)
```

$$\frac{e^{-x^2}}{\sqrt{\pi}\,x} - \frac{e^{-x^2}}{2\sqrt{\pi}\,x^3} + O\left(\frac{e^{-x^2}}{x^4}\right)$$

```
series(erfi(x), x = I*infinity, 3)
```

$$i + \frac{e^{x^2}}{\sqrt{\pi}\,x} + \frac{e^{x^2}}{2\sqrt{\pi}\,x^3} + O\left(\frac{e^{x^2}}{x^4}\right)$$

Parameters

x

An arithmetical expression

n

An arithmetical expression representing an integer larger than or equal to - 1

Return Values

Arithmetical expression

Algorithms

`erf`, `erfc`, and `erfi` are entire functions.

See Also

MuPAD Functions

`erf` | `erfi` | `inverf` | `inverfc` | `stats::normalQuantile`

More About

- "Error Functions and Fresnel Functions"

erfi

Imaginary error function

Syntax

```
erfi(x)
```

Description

$\mathrm{erfi}(x) = -\mathrm{i}\,\mathrm{erf}(\mathrm{i}\,x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{t^2}\,dt$ computes the imaginary error function.

This function is defined for all complex arguments `x`.

For floating-point arguments the error functions `erf`, `erfc`, and `erfi` return floating-point values. The implemented exact values are:

erfi(0) = 0, *erfi*(∞) = ∞, *erfi*(- ∞) = - ∞, *erfi*(i ∞) = i, *erfi*(- i ∞) = - i

For all other arguments, the error function returns symbolic function calls.

For the function `erfi` with floating-point arguments of large absolute value, internal numerical underflow or overflow can happen.

The error functions *erf*(*x*) = 1 - *erfc*(*x*) and *erfi*(*x*) = i (*erfc*(*x* i) - 1) return corresponding values for large arguments. See “Example 2” on page 1-636.

MuPAD can simplify expressions that contain error functions and their inverses. For real values `x`, the system applies the following simplification rules:

inverf(*erf*(*x*)) = *inverf*(1 - *erfc*(*x*)) = *inverfc*(1 - *erf*(*x*)) = *inverfc*(*erfc*(*x*)) = *x*,

inverf(- *erf*(*x*)) = *inverf*(*erfc*(*x*) - 1) = *inverfc*(1 + *erf*(*x*)) = *inverfc*(2 - *erfc*(*x*)) = - *x*

For any value `x`, the system applies the following simplification rules:

inverf(- *x*) = - *inverf*(*x*),

$inverfc(2 - x) = - inverfc(x)$,

$erf(inverf(x)) = erfc(inverfc(x)) = x$.

$erf(inverfc(x)) = erfc(inverf(x)) = 1 - x$.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

You can call error functions with exact and symbolic arguments:

```
erf(0), erf(3/2), erf(sqrt(2)), erf(infinity)
```

$$0,\ \mathrm{erf}\left(\frac{3}{2}\right),\ \mathrm{erf}(\sqrt{2}),\ 1$$

```
erfc(0), erfc(x + 1), erfc(-infinity)
```

$$1,\ \mathrm{erfc}(x+1),\ 2$$

```
erfc(0, n), erfc(x + 1, -1), erfc(-infinity, 5)
```

$$\frac{1}{2^n\,\Gamma\left(\frac{n}{2}+1\right)},\ \frac{2\,e^{-(x+1)^2}}{\sqrt{\pi}},\ \infty$$

```
erfi(0), erfi(x + 1), erfi(-infinity)
```

$$0,\ \mathrm{erfi}(x+1),\ -\infty$$

```
inverf(-1), inverf(0), inverf(1), inverf(x + 1), inverf(1/5)
```

$$-\infty,\ 0,\ \infty,\ \text{inverf}(x+1),\ \text{inverf}\left(\frac{1}{5}\right)$$

```
inverfc(0), inverfc(1), inverfc(2), inverfc(15), inverfc(x/5)
```

$$\infty,\ 0,\ -\infty,\ \text{inverfc}(15),\ \text{inverfc}\left(\frac{x}{5}\right)$$

For floating-point arguments, the error functions return floating-point values:

```
erf(-7.2), erfc(2.0 + 3.5*I), erfc(3.0, 4), erfi(5.5 + 1.0*I)
```

$$-1.0,\ -420.8123327 - 343.6612334\ i,$$

$$0.000000009433438115,\ -93412361266.0 - 5.089108513\ 10^{11}\ i$$

For floating-point arguments `x` from the interval `[-1, 1]`, `inverf` returns floating-point values:

```
inverf(-0.5), inverf(0.85)
```

$$-0.4769362762,\ 1.017902465$$

For floating-point arguments outside of this interval, `inverf` returns symbolic function calls:

```
inverf(-5.3), inverf(10.0)
```

$$-\text{inverf}(5.3),\ \text{inverf}(10.0)$$

For floating-point arguments `x` from the interval `[0, 2]`, `inverfc` returns floating-point values:

```
inverfc(0.5), inverfc(1.25)
```

$$0.4769362762,\ -0.225312055$$

For floating-point arguments outside of this interval, `inverfc` returns symbolic function calls:

```
inverfc(-1.25), inverfc(2.5)
```

$$-\text{inverfc}(3.25),\ \text{inverfc}(2.5)$$

Example 2

For large complex arguments, the error functions can return NaN:

```
erf(38000.0 + 3801.0*I),
erfi(38000.0 + 3801.0*I),
erfc(38000.0 + 3801.0*I)
```

$$\text{NaN},\ \text{NaN},\ \text{NaN}$$

For large floating-point arguments with positive real parts, `erfc` can return values truncated to `0.0`:

```
erfc(27281.1), erfc(27281.2)
```

$$4.085187475\ 10^{-323227329},\ 0.0$$

Example 3

The functions `diff`, `float`, `limit`, `expand`, `rewrite`, and `series` handle expressions involving the error functions:

```
diff(erf(x), x, x, x)
```

$$\frac{8\,x^2\,e^{-x^2}}{\sqrt{\pi}} - \frac{4\,e^{-x^2}}{\sqrt{\pi}}$$

```
diff(erfc(x, 3), x, x)
```

$$\text{erfc}(x,\ 1)$$

```
diff(inverf(x), x)
```

$$\frac{\sqrt{\pi}\, e^{\mathrm{inverf}(x)^2}}{2}$$

```
float(ln(3 + erfi(sqrt(PI)*I)))
```

$$1.150079617 + 0.3180894436\, i$$

```
limit(x/(1 + x)*erf(x), x = infinity)
```

$$1$$

```
expand(erfc(x, 3))
```

$$\frac{e^{-x^2}}{6\sqrt{\pi}} - \frac{x^3\,\mathrm{erfc}(x)}{6} - \frac{x\,\mathrm{erfc}(x)}{4} + \frac{x^2\, e^{-x^2}}{6\sqrt{\pi}}$$

```
rewrite(inverfc(x), inverf)
```

$$-\mathrm{inverf}(x-1)$$

```
series(erf(x), x = infinity, 3)
```

$$1 - \frac{e^{-x^2}}{\sqrt{\pi}\,x} + \frac{e^{-x^2}}{2\sqrt{\pi}\,x^3} + O\left(\frac{e^{-x^2}}{x^4}\right)$$

```
series(erfc(x), x = infinity, 3)
```

$$\frac{e^{-x^2}}{\sqrt{\pi}\,x} - \frac{e^{-x^2}}{2\sqrt{\pi}\,x^3} + O\left(\frac{e^{-x^2}}{x^4}\right)$$

```
series(erfi(x), x = I*infinity, 3)
```

$$i + \frac{e^{x^2}}{\sqrt{\pi}\,x} + \frac{e^{x^2}}{2\sqrt{\pi}\,x^3} + O\left(\frac{e^{x^2}}{x^4}\right)$$

Parameters

x

An arithmetical expression

Return Values

Arithmetical expression

Algorithms

`erf`, `erfc`, and `erfi` are entire functions.

See Also

MuPAD Functions
`erf` | `erfc` | `inverf` | `inverfc` | `stats::normalQuantile`

More About

- "Error Functions and Fresnel Functions"

inverf

Inverse of the error function

Syntax

```
inverf(x)
```

Description

inverf(*x*) computes the inverse of the error function.

This function is defined for all complex arguments `x`.

The inverse function `inverf` is singular at the points `x = -1` and `x = 1`.

The inverses of the error functions return floating-point values only for floating-point arguments that belong to a particular interval. Thus, the inverse of error function `inverf(x)` returns floating-point values for real values `x` from the interval `[-1, 1]`. The inverse of the complementary error function `inverfc(x)` returns floating-point values for real values `x` from the interval `[0, 2]`. The implemented exact values are:

inverf(- 1) = - ∞, *inverf*(0) = 0, *inverf*(1) = ∞,

inverfc(0) = ∞, *inverfc*(1) = 0, *inverfc*(2) = - ∞.

For all other arguments, the error functions return symbolic function calls.

MuPAD can simplify expressions that contain error functions and their inverses. For real values `x`, the system applies the following simplification rules:

inverf(*erf*(*x*)) = *inverf*(1 - *erfc*(*x*)) = *inverfc*(1 - *erf*(*x*)) = *inverfc*(*erfc*(*x*)) = *x*,

inverf(- *erf*(*x*)) = *inverf*(*erfc*(*x*) - 1) = *inverfc*(1 + *erf*(*x*)) = *inverfc*(2 - *erfc*(*x*)) = - *x*

For any value `x`, the system applies the following simplification rules:

inverf(- *x*) = - *inverf*(*x*),

inverfc(2 - *x*) = - *inverfc*(*x*),

$erf(inverf(x)) = erfc(inverfc(x)) = x$.

$erf(inverfc(x)) = erfc(inverf(x)) = 1 - x$.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

You can call error functions with exact and symbolic arguments:

```
erf(0), erf(3/2), erf(sqrt(2)), erf(infinity)
```

$$0,\ \mathrm{erf}\left(\frac{3}{2}\right),\ \mathrm{erf}(\sqrt{2}),\ 1$$

```
erfc(0), erfc(x + 1), erfc(-infinity)
```

$$1,\ \mathrm{erfc}(x+1),\ 2$$

```
erfc(0, n), erfc(x + 1, -1), erfc(-infinity, 5)
```

$$\frac{1}{2^n\,\Gamma\left(\frac{n}{2}+1\right)},\ \frac{2\,e^{-(x+1)^2}}{\sqrt{\pi}},\ \infty$$

```
erfi(0), erfi(x + 1), erfi(-infinity)
```

$$0,\ \mathrm{erfi}(x+1),\ -\infty$$

```
inverf(-1), inverf(0), inverf(1), inverf(x + 1), inverf(1/5)
```

$$-\infty,\ 0,\ \infty,\ \mathrm{inverf}(x+1),\ \mathrm{inverf}\left(\frac{1}{5}\right)$$

```
inverfc(0), inverfc(1), inverfc(2), inverfc(15), inverfc(x/5)
```

$$\infty,\ 0,\ -\infty,\ \mathrm{inverfc}(15),\ \mathrm{inverfc}\left(\frac{x}{5}\right)$$

For floating-point arguments, the error functions return floating-point values:

```
erf(-7.2), erfc(2.0 + 3.5*I), erfc(3.0, 4), erfi(5.5 + 1.0*I)
```

$$-1.0,\ -420.8123327 - 343.6612334\ i,$$

$$0.000000009433438115,\ -93412361266.0 - 5.089108513\ 10^{11}\ i$$

For floating-point arguments `x` from the interval `[-1, 1]`, `inverf` returns floating-point values:

```
inverf(-0.5), inverf(0.85)
```

$$-0.4769362762,\ 1.017902465$$

For floating-point arguments outside of this interval, `inverf` returns symbolic function calls:

```
inverf(-5.3), inverf(10.0)
```

$$-\mathrm{inverf}(5.3),\ \mathrm{inverf}(10.0)$$

For floating-point arguments `x` from the interval `[0, 2]`, `inverfc` returns floating-point values:

```
inverfc(0.5), inverfc(1.25)
```

$$0.4769362762,\ -0.225312055$$

For floating-point arguments outside of this interval, `inverfc` returns symbolic function calls:

```
inverfc(-1.25), inverfc(2.5)
```

$$-\text{inverfc}(3.25),\ \text{inverfc}(2.5)$$

Example 2

For large complex arguments, the error functions can return NaN:

```
erf(38000.0 + 3801.0*I),
erfi(38000.0 + 3801.0*I),
erfc(38000.0 + 3801.0*I)
```

$$\text{NaN},\ \text{NaN},\ \text{NaN}$$

For large floating-point arguments with positive real parts, `erfc` can return values truncated to `0.0`:

```
erfc(27281.1), erfc(27281.2)
```

$$4.085187475\ 10^{-323227329},\ 0.0$$

Example 3

The functions `diff`, `float`, `limit`, `expand`, `rewrite`, and `series` handle expressions involving the error functions:

```
diff(erf(x), x, x, x)
```

$$\frac{8\,x^2\,e^{-x^2}}{\sqrt{\pi}} - \frac{4\,e^{-x^2}}{\sqrt{\pi}}$$

```
diff(erfc(x, 3), x, x)
```

$$\text{erfc}(x,\ 1)$$

```
diff(inverf(x), x)
```

$$\frac{\sqrt{\pi}\, e^{\mathrm{inverf}(x)^2}}{2}$$

```
float(ln(3 + erfi(sqrt(PI)*I)))
```

$$1.150079617 + 0.3180894436\, i$$

```
limit(x/(1 + x)*erf(x), x = infinity)
```

$$1$$

```
expand(erfc(x, 3))
```

$$\frac{e^{-x^2}}{6\sqrt{\pi}} - \frac{x^3\,\mathrm{erfc}(x)}{6} - \frac{x\,\mathrm{erfc}(x)}{4} + \frac{x^2\, e^{-x^2}}{6\sqrt{\pi}}$$

```
rewrite(inverfc(x), inverf)
```

$$-\mathrm{inverf}(x-1)$$

```
series(erf(x), x = infinity, 3)
```

$$1 - \frac{e^{-x^2}}{\sqrt{\pi}\,x} + \frac{e^{-x^2}}{2\sqrt{\pi}\,x^3} + O\left(\frac{e^{-x^2}}{x^4}\right)$$

```
series(erfc(x), x = infinity, 3)
```

$$\frac{e^{-x^2}}{\sqrt{\pi}\,x} - \frac{e^{-x^2}}{2\sqrt{\pi}\,x^3} + O\left(\frac{e^{-x^2}}{x^4}\right)$$

```
series(erfi(x), x = I*infinity, 3)
```

$$i + \frac{e^{x^2}}{\sqrt{\pi}\,x} + \frac{e^{x^2}}{2\sqrt{\pi}\,x^3} + O\left(\frac{e^{x^2}}{x^4}\right)$$

Parameters

x

An arithmetical expression

Return Values

Arithmetical expression

See Also

MuPAD Functions

`erf` | `erfc` | `erfi` | `inverfc` | `stats::normalQuantile`

More About

- "Error Functions and Fresnel Functions"

inverfc

Inverse of the complementary error function

Syntax

```
inverfc(x)
```

Description

inverfc(*x*) = *inverf*(1 - *x*) computes the inverse of the complementary error function.

This function is defined for all complex arguments `x`.

The inverse function `inverfc` is singular at the points `x = 0` and `x = 2`.

The inverses of the error functions return floating-point values only for floating-point arguments that belong to a particular interval. Thus, the inverse of error function `inverf(x)` returns floating-point values for real values `x` from the interval `[-1, 1]`. The inverse of the complementary error function `inverfc(x)` returns floating-point values for real values `x` from the interval `[0, 2]`. The implemented exact values are:

inverf(- 1) = - ∞, *inverf*(0) = 0, *inverf*(1) = ∞,

inverfc(0) = ∞, *inverfc*(1) = 0, *inverfc*(2) = - ∞.

For all other arguments, the error functions return symbolic function calls.

MuPAD can simplify expressions that contain error functions and their inverses. For real values `x`, the system applies the following simplification rules:

inverf(*erf*(*x*)) = *inverf*(1 - *erfc*(*x*)) = *inverfc*(1 - *erf*(*x*)) = *inverfc*(*erfc*(*x*)) = *x*,

inverf(- *erf*(*x*)) = *inverf*(*erfc*(*x*) - 1) = *inverfc*(1 + *erf*(*x*)) = *inverfc*(2 - *erfc*(*x*)) = - *x*

For any value `x`, the system applies the following simplification rules:

inverf(- *x*) = - *inverf*(*x*),

inverfc(2 - *x*) = - *inverfc*(*x*),

erf(*inverf*(*x*)) = *erfc*(*inverfc*(*x*)) = *x*.

erf(*inverfc*(*x*)) = *erfc*(*inverf*(*x*)) = 1 - *x*.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

You can call error functions with exact and symbolic arguments:

```
erf(0), erf(3/2), erf(sqrt(2)), erf(infinity)
```

$$0,\ \mathrm{erf}\left(\frac{3}{2}\right),\ \mathrm{erf}(\sqrt{2}),\ 1$$

```
erfc(0), erfc(x + 1), erfc(-infinity)
```

$$1,\ \mathrm{erfc}(x+1),\ 2$$

```
erfc(0, n), erfc(x + 1, -1), erfc(-infinity, 5)
```

$$\frac{1}{2^n\,\Gamma\left(\frac{n}{2}+1\right)},\ \frac{2\,e^{-(x+1)^2}}{\sqrt{\pi}},\ \infty$$

```
erfi(0), erfi(x + 1), erfi(-infinity)
```

$$0,\ \mathrm{erfi}(x+1),\ -\infty$$

```
inverf(-1), inverf(0), inverf(1), inverf(x + 1), inverf(1/5)
```

$$-\infty,\ 0,\ \infty,\ \text{inverf}(x+1),\ \text{inverf}\left(\frac{1}{5}\right)$$

```
inverfc(0), inverfc(1), inverfc(2), inverfc(15), inverfc(x/5)
```

$$\infty,\ 0,\ -\infty,\ \text{inverfc}(15),\ \text{inverfc}\left(\frac{x}{5}\right)$$

For floating-point arguments, the error functions return floating-point values:

```
erf(-7.2), erfc(2.0 + 3.5*I), erfc(3.0, 4), erfi(5.5 + 1.0*I)
```

$$-1.0,\ -420.8123327 - 343.6612334\ i,$$

$$0.000000009433438115,\ -93412361266.0 - 5.089108513\ 10^{11}\ i$$

For floating-point arguments `x` from the interval `[-1, 1]`, `inverf` returns floating-point values:

```
inverf(-0.5), inverf(0.85)
```

$$-0.4769362762,\ 1.017902465$$

For floating-point arguments outside of this interval, `inverf` returns symbolic function calls:

```
inverf(-5.3), inverf(10.0)
```

$$-\text{inverf}(5.3),\ \text{inverf}(10.0)$$

For floating-point arguments `x` from the interval `[0, 2]`, `inverfc` returns floating-point values:

```
inverfc(0.5), inverfc(1.25)
```

$$0.4769362762,\ -0.225312055$$

For floating-point arguments outside of this interval, `inverfc` returns symbolic function calls:

```
inverfc(-1.25), inverfc(2.5)
```

$$-\text{inverfc}(3.25),\ \text{inverfc}(2.5)$$

Example 2

For large complex arguments, the error functions can return NaN:

```
erf(38000.0 + 3801.0*I),
erfi(38000.0 + 3801.0*I),
erfc(38000.0 + 3801.0*I)
```

$$\text{NaN},\ \text{NaN},\ \text{NaN}$$

For large floating-point arguments with positive real parts, `erfc` can return values truncated to `0.0`:

```
erfc(27281.1), erfc(27281.2)
```

$$4.085187475\ 10^{-323227329},\ 0.0$$

Example 3

The functions `diff`, `float`, `limit`, `expand`, `rewrite`, and `series` handle expressions involving the error functions:

```
diff(erf(x), x, x, x)
```

$$\frac{8\,x^2\,e^{-x^2}}{\sqrt{\pi}} - \frac{4\,e^{-x^2}}{\sqrt{\pi}}$$

```
diff(erfc(x, 3), x, x)
```

$$\text{erfc}(x,\ 1)$$

```
diff(inverf(x), x)
```

$$\frac{\sqrt{\pi}\, e^{\mathrm{inverf}(x)^2}}{2}$$

```
float(ln(3 + erfi(sqrt(PI)*I)))
```

$$1.150079617 + 0.3180894436\, i$$

```
limit(x/(1 + x)*erf(x), x = infinity)
```

$$1$$

```
expand(erfc(x, 3))
```

$$\frac{e^{-x^2}}{6\sqrt{\pi}} - \frac{x^3\,\mathrm{erfc}(x)}{6} - \frac{x\,\mathrm{erfc}(x)}{4} + \frac{x^2\, e^{-x^2}}{6\sqrt{\pi}}$$

```
rewrite(inverfc(x), inverf)
```

$$-\mathrm{inverf}(x-1)$$

```
series(erf(x), x = infinity, 3)
```

$$1 - \frac{e^{-x^2}}{\sqrt{\pi}\,x} + \frac{e^{-x^2}}{2\sqrt{\pi}\,x^3} + O\left(\frac{e^{-x^2}}{x^4}\right)$$

```
series(erfc(x), x = infinity, 3)
```

$$\frac{e^{-x^2}}{\sqrt{\pi}\,x} - \frac{e^{-x^2}}{2\sqrt{\pi}\,x^3} + O\left(\frac{e^{-x^2}}{x^4}\right)$$

```
series(erfi(x), x = I*infinity, 3)
```

$$i + \frac{e^{x^2}}{\sqrt{\pi}\,x} + \frac{e^{x^2}}{2\sqrt{\pi}\,x^3} + O\left(\frac{e^{x^2}}{x^4}\right)$$

Parameters

x

An arithmetical expression

Return Values

Arithmetical expression

See Also

MuPAD Functions

`erf` | `erfc` | `erfi` | `inverf` | `stats::normalQuantile`

More About

- "Error Functions and Fresnel Functions"

error

Raise a user-specified exception

Syntax

```
error(message)
```

Description

`error(message)` aborts the current procedure, returns to the interactive level, and displays the error message `message`.

If the error is not caught via `traperror` by a procedure that has directly or indirectly called the current procedure, control is returned to the interactive level, and the string `message` is printed as an error message.

The printed error message has the form `Error: message [name]`, where `name` is the name of the procedure containing the call to `error`. See the examples.

Errors can be caught by the function `traperror`. If an error occurs while the arguments of `traperror` are evaluated, control is returned to the procedure containing the call to `traperror` and not to the interactive level. No error message is printed. The return value of `traperror` is `1028` when it catches an error raised by `error`; see “Example 2” on page 1-652.

The function `error` is useful to raise an error in the type checking part of a user-defined procedure, when this procedure is called with invalid arguments.

Examples

Example 1

If the divisor of the following simple division routine is `0`, then an error is raised:

```
mydivide := proc(n, d) begin
```

```
  if iszero(d) then
    error("Division by 0")
  end_if;
  n/d
end_proc:
mydivide(2, 0)
```

```
Error: Division by 0 [mydivide]
```

Example 2

When the error is raised in the following procedure p, control is returned to the interactive level immediately. The second call to `print` is never executed. Note that the procedure's name is printed in the error message:

```
p := proc() begin
  print("entering procedure p");
  error("oops");
  print("leaving procedure p")
end_proc:
p()
```

```
"entering procedure p"
```

```
Error: oops [p]
```

The following procedure q calls the procedure p and catches any error that is raised within p:

```
q := proc() begin
  print("entering procedure q");
  print("caught error: ", traperror(p()));
  print("leaving procedure q")
end_proc:
q()
```

```
"entering procedure q"
```

```
"entering procedure p"
```

"caught error: ", 1028

"leaving procedure q"

Parameters

`message`

The error message: a string

See Also

MuPAD Functions
`getlasterror` | `lasterror` | `traperror` | `warning`

euler

Euler numbers and polynomials

Syntax

```
euler(n)
```

```
euler(n, x)
```

Description

`euler(n)` returns the n-th Euler number.

`euler(n, x)` returns the n-th Euler polynomial in x.

The Euler polynomials are defined by the generating function

$$\frac{2\,e^{x\,t}}{e^{t}+1} = \sum_{n=0}^{\infty} \frac{\mathrm{euler}(n,x)}{n!}\,t^{n}$$

.

The Euler numbers are defined by `euler(n) = 2^n*euler(n,1/2)`.

An error occurs if `n` is a numerical value not representing a nonnegative integer.

If `n` is an integer larger than the value returned by `Pref::autoExpansionLimit()`, then the call `euler(n)` is returned symbolically. Use `expand(euler(n))` to get an explicit numerical result for large integers `n`.

If `n` contains non-numerical symbolic identifiers, then the call `euler(n)` is returned symbolically. In most cases, the same holds true for calls `euler(n, x)`. Some simplifications are implemented for special numerical values such as `x = 0`, `x = 1/2`, `x = 1` etc. for symbolic `n` . Cf. “Example 3” on page 1-656.

Note: Note that floating-point evaluation for high degree polynomials may be numerically unstable. See “Example 4” on page 1-657.

The floating-point evaluation on the standard interval $x \in [0, 1]$ is numerically stable for arbitrary `n`.

To use the Euler constant, call `E` or `exp(1)`. To use the Euler-Mascheroni constant, call `EULER`. See "Mathematical Constants Available in MuPAD". You can approximate these constants with floating-point numbers by using `float`.

Environment Interactions

When called with a floating-point value `x`, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

The first Euler numbers are:

```
euler(n) $ n = 0..11
```

$$1, 0, -1, 0, 5, 0, -61, 0, 1385, 0, -50521, 0$$

The first Euler polynomials:

```
euler(n, x) $ n = 0..4
```

$$1, x - \frac{1}{2}, x^2 - x, x^3 - \frac{3x^2}{2} + \frac{1}{4}, x^4 - 2x^3 + x$$

If `n` is symbolic, then a symbolic call is returned:

```
euler(n, x), euler(n + 3/2, x), euler(n + 5*I, x)
```

$$\text{euler}(n, x), \text{euler}\left(n + \frac{3}{2}, x\right), \text{euler}(n + 5\,i, x)$$

Example 2

If x is not an indeterminate, then the evaluation of the Euler polynomial at the point x is returned:

```
euler(50, 1 + I)
```

$$-1+620906113959162509876411268097228422281 71\,i$$

```
euler(3, 1 - y) = expand(euler(3, 1 - y))
```

$$\frac{1}{4}-(y-1)^3-\frac{3\,(y-1)^2}{2}=-y^3+\frac{3\,y^2}{2}-\frac{1}{4}$$

Example 3

Certain simplifications occur for some special numerical values of x, even if n is symbolic:

```
euler(n, 0), euler(n, 1/2), euler(n, 1)
```

$$-\frac{2\,\text{bernoulli}(n+1)\left(2^{n+1}-1\right)}{n+1},\ \frac{\text{euler}(n)}{2^n},\ \frac{2\,\text{bernoulli}(n+1)\left(2^{n+1}-1\right)}{n+1}$$

Calls with numerical arguments between $\frac{1}{2}$ and 1 are automatically rewritten in terms of calls with arguments between 0 and $\frac{1}{2}$:

```
euler(n, 2/3), euler(n, 0.7)
```

$$(-1)^n\,\text{euler}\left(n,\ \frac{1}{3}\right),\ (-1)^n\,\text{euler}(n,\ 0.3)$$

Calls with negative numerical arguments are automatially rewritten in terms of calls with positive arguments:

```
euler(n, -2)
```

$$(-1)^{n+1}\,\text{euler}(n,\ 2)+2\,(-2)^n$$

```
euler(n, -12.345)
```

$$2\,(-12.345)^{n} + (-1)^{n+1}\,\mathrm{euler}(n, 12.345)$$

Example 4

Float evaluation of high degree polynomials may be numerically unstable:

```
exact := euler(50, 1 + I): float(exact);
```

$$-1.0 + 6.20906114\ 10^{40}\,\mathrm{i}$$

```
euler(50, float(1 + I))
```

$$-1.774947368\ 10^{23} + 6.20906114\ 10^{40}\,\mathrm{i}$$

```
DIGITS := 40: euler(50, float(1 + I))
```

$$-0.9999999981373548507689370896315932124576$$
$$+\,6.2090611395916250987641126809722842228177\ 10^{40}\,\mathrm{i}$$

```
delete exact, DIGITS:
```

Example 5

Some system functions such as `diff` or `expand` handle symbolic calls of `euler`:

```
diff(euler(n, f(x)), x)
```

$$n\,\mathrm{euler}(n-1, f(x))\,\frac{\partial}{\partial x} f(x)$$

```
expand(euler(n, x + 2))
```

$$\mathrm{euler}(n, x) - 2\,x^{n} + 2\,(x+1)^{n}$$

```
expand(euler(n, -x))
```

$$2\,(-x)^n-(-1)^n\,\mathrm{euler}(n,x)$$

```
expand(euler(n, 3*x))
```

$$3^n\,\mathrm{euler}(n,x)-3^n\,\mathrm{euler}\left(n,x+\frac{1}{3}\right)+3^n\,\mathrm{euler}\left(n,x+\frac{2}{3}\right)$$

Parameters

n

An arithmetical expression representing a nonnegative integer

x

An arithmetical expression

Return Values

Arithmetical expression.

References

M. Abramowitz and I. Stegun, “Handbook of Mathematical Functions”, Dover Publications Inc., New York (1965).

See Also

MuPAD Functions

`bernoulli`

eval

Evaluate an object

Syntax

```
eval(object)
```

Description

`eval(object)` evaluates its argument `object` by recursively replacing the identifiers occurring in it by their values and executing function calls, and then evaluates the result again.

`eval` serves to request the evaluation of unevaluated or partially evaluated objects. *Evaluation* means that identifiers are replaced by their values and function calls are executed.

Usually, every system function automatically evaluates its arguments and returns a fully evaluated object, and using `eval` is only necessary in exceptional cases. For example, the functions `map`, `op`, and `subs` may return objects that are not fully evaluated. See “Example 1” on page 1-661.

Like most other MuPAD functions, `eval` first evaluates its argument. Then it evaluates the result again. At interactive level, the second evaluation usually has no effect, but this is different within procedures. See “Example 3” on page 1-661 and “Example 4” on page 1-663.

`eval` is sensitive to the value of the environment variable `LEVEL`, which determines the maximal depth of the recursive process that replaces an identifier by its value during evaluation. The evaluation of the argument and the subsequent evaluation of the result both take place with substitution depth `LEVEL`. See “Example 3” on page 1-661.

If a local variable or a formal parameter, of type `DOM_VAR`, of a `procedure` occurs in `object`, then it is always replaced by its value when `eval` evaluates its argument, independent of the value of `LEVEL`. At the subsequent second evaluation, the value of the local variable is evaluated with substitution depth given by `LEVEL`, which usually is `1`. Cf. “Example 4” on page 1-663.

The behavior of `eval` within a procedure may sometimes not be what you expect, since the default substitution depth within procedures is `1` and `eval` evaluates with this substitution depth. Use `level` to request a complete evaluation within a procedure; see the corresponding help page for details.

`eval` enforces the evaluation of expressions of the form `hold(x)`: `eval(hold(x))` is equivalent to `x`. Cf. “Example 2” on page 1-661.

`eval` accepts expression sequences as arguments. See “Example 3” on page 1-661. In particular, the call `eval()` returns the empty sequence `null()`.

`eval` does not recursively descend into `arrays`. Use the call `map(object, eval)` to evaluate the entries of an array. Cf. “Example 5” on page 1-664.

`eval` does not recursively descend into `tables`. Use the call `map(object, eval)` to evaluate the entries of a table.

However, it is not possible to evaluate the indices of a given table. If you want to do this, create a new table with the evaluated operands of the old one. Cf. “Example 6” on page 1-664.

`Polynomials` are not further evaluated by `eval`. Use `evalp` to substitute values for the indeterminates of a polynomial, and use the call `mapcoeffs(object, eval)` to evaluate all coefficients. Cf. “Example 7” on page 1-665.

The evaluation of elements of a user-defined domain depends on the implementation of the domain. Usually, domain elements remain unevaluated. If the domain has a slot `"evaluate"`, the corresponding slot routine is called with the domain element as argument at each evaluation, and hence it is called twice when `eval` is invoked. Cf. “Example 8” on page 1-666.

Environment Interactions

`eval` is sensitive to the value of the environment variable `LEVEL`, which determines the maximal substitution depth for identifiers.

Examples

Example 1

`subs` performs a substitution, but does not evaluate the result:

```
subs(ln(x), x = 1)
```

$$\ln(1)$$

An explicit call of `eval` is necessary to evaluate the result:

```
eval(subs(ln(x), x = 1))
```

$$0$$

`text2expr` does not evaluate its result either:

```
a := c:
text2expr("a + a"), eval(text2expr("a + a"))
```

$$a + a,\ 2\,c$$

Example 2

The function `hold` prevents the evaluation of its argument. A later evaluation can be forced with `eval`:

```
hold(1 + 1);  eval(%)
```

$$1 + 1$$

$$2$$

Example 3

When an object is evaluated, identifiers are replaced by their values recursively. The maximal recursion depth of this process is given by the environment variable `LEVEL`:

```
delete a0, a1, a2, a3, a4:
a0 := a1:  a1 := a2 + 2:  a2 := a3 + a4:  a3 := a4^2:  a4 := 5:

LEVEL := 1:  a0, a0 + a2;
LEVEL := 2:  a0, a0 + a2;
LEVEL := 3:  a0, a0 + a2;
LEVEL := 4:  a0, a0 + a2;
LEVEL := 5:  a0, a0 + a2;
```

$$a1,\ a1 + a3 + a4$$

$$a2 + 2,\ a4^2 + a2 + 7$$

$$a3 + a4 + 2,\ a3 + a4 + 32$$

$$a4^2 + 7,\ a4^2 + 37$$

$$32,\ 62$$

`eval` first evaluates its argument and then evaluates the result again. Both evaluations happen with substitution depth given by `LEVEL`:

```
LEVEL := 1:  eval(a0, a0 + a2);
LEVEL := 2:  eval(a0, a0 + a2);
LEVEL := 3:  eval(a0, a0 + a2);
```

$$a2 + 2,\ a4^2 + a2 + 7$$

$$a4^2 + 7,\ a4^2 + 37$$

$$32,\ 62$$

Since the default value of `LEVEL` is `100`, `eval` usually has no effect at interactive level:

```
delete LEVEL:
```

```
a0, eval(a0), a0 + a2, eval(a0 + a2)
```

$$32,\ 32,\ 62,\ 62$$

Example 4

This example shows the difference between the evaluation of identifiers and local variables. By default, the value of LEVEL is 1 within a procedure, i.e., a global identifier is replaced by its value when evaluated, but there is no further recursive evaluation. This changes when LEVEL is assigned a bigger value inside the procedure:

```
delete a0, a1, a2, a3:
a0 := a1 + a2:  a1 := a2 + a3:  a2 := a3^2 - 1:  a3 := 5:
p := proc()
       save LEVEL;
     begin
       print(a0, eval(a0)):
       LEVEL := 2:
       print(a0, eval(a0)):
     end_proc:

p()
```

$$a1 + a2,\ a3^2 + a3 + a2 - 1$$

$$a3^2 + a3 + a2 - 1,\ 53$$

In contrast, evaluation of a local variable replaces it by its value, without further evaluation. When eval is applied to an object containing a local variable, then the effect is an evaluation of the value of the local variable with substitution depth LEVEL:

```
q := proc()
       save LEVEL;
       local x;
     begin
       x := a0:
       print(x, eval(x)):
       LEVEL := 2:
       print(x, eval(x)):
```

```
     end_proc:
q()
```

$$a1 + a2,\ a3^2 + a3 + a2 - 1$$

$$a1 + a2,\ a3^2 + 28$$

The command `x:=a0` assigns the value of the identifier `a0`, namely the unevaluated expression `a1+a2`, to the local variable `x`, and `x` is replaced by this value every time it is evaluated, independent of the value of `LEVEL`:

Example 5

In contrast to lists and sets, evaluation of an array does not evaluate its entries. Thus `eval` has no effect for arrays either. Use `map` to evaluate all entries of an array:

```
delete a, b:
L := [a, b]:  A := array(1..2, L):  a := 1:   b := 2:
L, A, eval(A), map(A, eval)
```

$$[1, 2],\ (a\ \ b),\ (a\ \ b),\ (1\ \ 2)$$

The call `map(A, gamma)` does not evaluate the entries of the array `A` before applying the function `gamma`. Map the function `gamma@eval` to enforce the evaluation:

```
map(A, gamma), map(A, gamma@eval)
```

$$(\Gamma(a)\ \ \Gamma(b)),\ (1\ \ 1)$$

Example 6

Similarly, evaluation of a table does not evaluate its entries, and you can use `map` to achieve this. However, this does not affect the indices:

```
delete a, b:
T := table(a = b):  a := 1:  b := 2:
T, eval(T), map(T, eval)
```

$$\overline{a\,|b},\ \overline{a\,|b},\ \overline{a\,|2}$$

If you want a table with evaluated indices as well, create a new table from the evaluated operands of the old table. Using `eval` is necessary here since the operand function `op` does not evaluate the returned operands:

```
op(T), table(eval(op(T)))
```

$$a = b,\ \overline{1\,|2}$$

Example 7

`Polynomials` are inert when evaluated, and also `eval` has no effect:

```
delete a, x:  p := poly(a*x, [x]):  a := 2:  x := 3:
p, eval(p), map(p, eval)
```

$$\mathrm{poly}(a\,x,\,[x]),\ \mathrm{poly}(a\,x,\,[x]),\ \mathrm{poly}(a\,x,\,[x])$$

Use `mapcoeffs` to evaluate all coefficients:

```
mapcoeffs(p, eval)
```

$$\mathrm{poly}(2\,x,\,[x])$$

If you want to substitute a value for the indeterminate `x`, use `evalp`:

```
delete x:  evalp(p, x = 3)
```

$$3\,a$$

As you can see, the result of an `evalp` call may contain unevaluated identifiers, and you can evaluate them by an application of `eval`:

```
eval(evalp(p, x = 3))
```

$$6$$

Example 8

The evaluation of an element of a user-defined domains depends on the implementation of the domain. Usually, it is not evaluated further:

```
delete a:  T := newDomain("T"):
e := new(T, a):  a := 1:
e, eval(e), map(e, eval), val(e)
```

$$\mathrm{new}(T, a), \mathrm{new}(T, a), \mathrm{new}(T, a), \mathrm{new}(T, a)$$

If the slot `"evaluate"` exists, the corresponding slot routine is called for a domain element each time it is evaluated. We implement the routine `T::evaluate`, which simply evaluates all internal operands of its argument, for our domain `T`. The unevaluated domain element can still be accessed via `val`:

```
T::evaluate := x -> new(T, eval(extop(x))):
e, eval(e), map(e, eval), val(e)
```

$$\mathrm{new}(T, 1), \mathrm{new}(T, 1), \mathrm{new}(T, 1), \mathrm{new}(T, a)$$

Parameters

`object`

Any MuPAD object

Return Values

Evaluated object.

See Also

MuPAD Functions

`context` | `evalassign` | `evalp` | `freeze` | `hold` | `indexval` | `LEVEL` | `level` | `MAXDEPTH` | `MAXLEVEL` | `val`

More About

- “Evaluations in Symbolic Computations”
- “Level of Evaluation”
- “Actual and Displayed Results of Evaluations”

evalassign

Assignment with evaluation of the left hand side

Syntax

```
evalassign(x, value, i)

evalassign(x, value)
```

Description

`evalassign(x, value, i)` evaluates `x` with substitution depth `i` and assigns `value` to the result of the evaluation.

`evalassign(x, value, i)` evaluates `value`, as usual. Then it evaluates `x` with substitution depth `i`, and finally it assigns the evaluation of `value` to the evaluation of `x`.

The difference between `evalassign` and the assignment operator `:=` is that the latter does not evaluate its left hand side at all.

As usual, the evaluation of `value` takes place with substitution depth given by `LEVEL`. By default, it is `1` within a procedure.

See the help pages of `LEVEL` and `level` for the notion of substitution depth and for details about evaluation.

The third argument is optional. The calls `evalassign(x, value)`, `evalassign(x, value, 0)`, `x := value`, and `_assign(x, value)` are all equivalent.

The result of the evaluation of `x` must be a valid left hand side for an assignment. See the help page of `:=` for details.

The second argument is *not* flattened. Hence it may also be a sequence. Cf. “Example 2” on page 1-669.

Examples

Example 1

`evalassign` can be used in situations such as the following. Suppose that an identifier `a` has another identifier `b` as its value, and that we want to assign something to this *value* of `a`, not to `a` itself:

```
delete a, b: a := b:
evalassign(a, 100, 1): level(a, 1), a, b
```

```
b, 100, 100
```

This would not have worked with the assignment operator `:=`, which does not evaluate its left hand side:

```
delete a, b: a := b:
a := 100: level(a, 1), a, b
```

```
100, 100, b
```

Example 2

The second argument may also be a sequence:

```
a := b:
evalassign(a, (3,5), 1):
b
```

```
3, 5
```

Parameters

x

An object that evaluates to a valid left hand side of an assignment

value

Any MuPAD object

i

A nonnegative integer less than 2^{31}

Return Values

`value`.

Algorithms

The function `level` is used for the evaluation of `x`. Hence `i` may exceed the value of `LEVEL`.

All special rules for `_assign` apply: see there on further details on indexed assignments, assignments to slots, and the `protect` mechanism.

See Also

MuPAD Functions

`:=` | `_assign` | `assign` | `assignElements` | `delete` | `eval` | `LEVEL` | `level`

|, evalAt

Insert a value (evaluate at a point)

Syntax

```
 f | x = v

evalAt(f, x = v)

f | ( x1= v1, x2= v2, … )

evalAt(f, x1 = v1, x2 = v2, …)

evalAt(f, x1 = v1, x2 = v2, …)

f | [x1= v1, x2= v2, …]

evalAt(f, [x1 = v1, x2 = v2, …])

f | {x1= v1, x2= v2, …}

evalAt(f, {x1 = v1, x2 = v2, …})
```

Description

`evalAt(f, x = v)` substitutes `x = v` in the object `f` and evaluates.

The MuPAD statement `f | x = v` serves as a shortcut for calling `evalAt(f, x = v)`.

`evalAt(f, x = v)` evaluates the object `f` at the point `x = v`. Essentially, it is the same as `eval ( subs(f, x = v))`, but limited to free (as opposed to bound) variables.

Several substitutions of indeterminates by values can be done by `evalAt(f, x1 = v1, x2 = v2, ...)`. This is equivalent to `evalAt(... (evalAt(evalAt(f, x1 = v1), x2 = v2), ...), ...)`, i.e., `x1 = v1` is substituted in `f`, then `x2 = v2` is substituted in the result etc. E.g., `evalAt(x, x = y, y = 1)` yields 1.

Note that the three (equivalent) calls `evalAt(f, (x1 = v1, x2 = v2, ...))`, `evalAt(f, [x1 = v1, x2 = v2, ...])`, `evalAt(f, {x1 = v1, x2 =`

`v2, ...})` do parallel substitutions, i.e., the substitutions `x1 = v1`, `x2 = v2` are all performed on `f` simultaneously. Consequently, `evalAt(x, [x = y, y = 1])` yields `y`, not 1!

The operator `|` provides a shortcut for calling `evalAt`:

The command `f | x = v` is equivalent to calling `evalAt(f, x = v)`.

Similarly, `f | (x1=v1, x2=v2, ...)` is equivalent to `evalAt(f, (x1=v1, x2=v2, ...))`, `f | [x1=v1, x2=v2, ...]` is equivalent to `evalAt(f, [x1=v1, x2=v2, ...])`, `f | [x1=v1, x2=v2, ...}` is equivalent to `evalAt(f, {x1=v1, x2=v2, ...})`.

Note: The sequential substitution `evalAt(f, x1 = v1, x2 = v2, ...)` cannot be done via `f | x1 = v1, x2 = v2, ...`: this produces the sequence `evalAt(f, x1 = v1)`, x2 = v2, ... Use `f | x1 = v1 | x2 = v2 | ...` for sequential substitution. E.g., the statement `x | x = y | y = 1` yields 1.

Examples

Example 1

Calls to `evalAt` and corresponding statements using the operator `|` are equivalent:

```
evalAt(x^2 + sin(x), x = 1);
x^2 + sin(x) | x = 1
```

$$\sin(1) + 1$$

$$\sin(1) + 1$$

We use the operator `|` to evaluate an expression `f` representing a function of `x` at several points:

```
f := x + exp(x):
f | x = 3;
f | x = 5.0;
```

```
f | x = y;
```

$$e^3 + 3$$

$$153.4131591$$

$$y + e^y$$

We create a matrix with symbolic entries and evaluate the matrix with various values for the symbols:

```
A := matrix([[x, sin(PI*x)], [2, y]]);
A | x = a;
A | [x = a, y = b]
```

$$\begin{pmatrix} x & \sin(\pi x) \\ 2 & y \end{pmatrix}$$

$$\begin{pmatrix} a & \sin(\pi a) \\ 2 & y \end{pmatrix}$$

$$\begin{pmatrix} a & \sin(\pi a) \\ 2 & b \end{pmatrix}$$

```
delete f, A:
```

Example 2

We do several substitutions simultaneously:

```
f := cos(y) + sin(x) + x*y;
f | (x = 1, y = 2);
f | [x = 1, y = 2];
f | {x = 1, y = 2};
```

$$\cos(y) + \sin(x) + x\,y$$

$$\cos(2) + \sin(1) + 2$$

$$\cos(2) + \sin(1) + 2$$

$$\cos(2) + \sin(1) + 2$$

```
delete f:
```

Parameters

`f`

An arbitrary MuPAD object.

`x, x1, x2, …`

indeterminates or indexed indeterminates.

`v, v1, v2, …`

The values for `x, x1, x2, ….`

Return Values

Copy of the input object `f` with replaced operands.

Overloaded By

f

See Also

MuPAD Functions

`evalp | subs | subsex | subsop`

evalp

Evaluate a polynomial at a point

Syntax

```
evalp(p, x = v, …)

evalp(p, [x = v, …])

evalp(f, <vars>, x = v, …)

evalp(f, <vars>, [x = v, …])
```

Description

`evalp(p, x = v)` evaluates the polynomial `p` in the variable `x` at the point `v`.

An error occurs if `x` is not an indeterminate of `p`. The value `v` may be any object that could also be used as coefficient. The result is an element of the coefficient ring of `p` if `p` is univariate. If `p` is multivariate, the result is a polynomial in the remaining variables.

If several evaluation points are given, the evaluations take place in succession from left to right. Each evaluation follows the rules above.

For a polynomial `p` in the variables `x1,x2,...`, the syntax `p(v1,v2,...)` can be used instead of `evalp(p, x1 = v1, x2 = v2, ...)`.

`evalp(f, vars, x = v, ...)` first converts the polynomial expression `f` to a polynomial with the variables given by `vars`. If no variables are given, they are searched for in `f`. See `poly` about details of the conversion. `FAIL` is returned if `f` cannot be converted to a polynomial. A successfully converted polynomial is evaluated as above. The result is converted to an expression.

Horner's rule is used to evaluate the polynomial. The evaluation of variables at the point `0` is most efficient and should take place first. After that, the remaining main variable should be evaluated first.

The result of `evalp` is not evaluated further. One may use `eval` to fully evaluate the result.

Instead of `evalp(p, x1 = v1, x2 = v2, ...)` one may also use the equivalent form `evalp(p, [x1 = v1, x2 = v2, ...])`.

Examples

Example 1

`evalp` is used to evaluate the polynomial expression $x^2 + 2x + 3$ at the point $x = a + 2$. The form of the resulting expression reflects the fact that Horner's rule was used:

```
evalp(x^2 + 2*x + 3, x = a + 2)
```

$$(a+2)\,(a+4)+3$$

Example 2

`evalp` is used to evaluate a polynomial in the indeterminates x and y at the point $x = 3$. The result is a polynomial in the remaining indeterminate y:

```
p := poly(x^2 + x*y + 2, [x, y]): evalp(p, x = 3)
```

$$\mathrm{poly}(3\,y+11,\,[y])$$

```
delete p:
```

Example 3

Polynomials may be called like functions in order to evaluate all variables:

```
p := poly(x^2 + x*y, [x, y]): evalp(p, x = 3, y = 2) = p(3, 2)
```

$$15 = 15$$

```
delete p:
```

Example 4

If not all variables are replaced by values, the result is a polynomial in the remaining variables:

```
evalp(poly(x*y*z + x^2 + y^2 + z^2, [x, y, z]), x = 1, y = 1)
```

$$\mathrm{poly}(z^2 + z + 2, [z])$$

Example 5

The result of `evalp` is not evaluated further. We first define a polynomial `p` with coefficient `a` and then change the value of `a`. The change is not reflected by `p`, because polynomials do not evaluate their coefficients implicitly. One must map the function `eval` onto the coefficients in order to enforce evaluation:

```
p := poly(x^2 + a*y + 1, [x,y]): a := 2:
p, mapcoeffs(p, eval)
```

$$\mathrm{poly}(x^2 + a\,y + 1, [x, y]),\ \mathrm{poly}(x^2 + 2\,y + 1, [x, y])$$

If we use `evalp` to evaluate `p` at the point $x = 1$, the result is not fully evaluated. One must use `eval` to get fully evaluated coefficients:

```
r := evalp(p, x = 1):
r, mapcoeffs(r, eval)
```

$$\mathrm{poly}(a\,y + 2, [y]),\ \mathrm{poly}(2\,y + 2, [y])$$

```
delete p, a, r:
```

Parameters

p

A polynomial of type `DOM_POLY`

x

An indeterminate

v

The value for `x`: an element of the coefficient ring of the polynomial

f

A polynomial expression

vars

A list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

Return Values

Element of the coefficient ring, or a polynomial, or a polynomial expression, or `FAIL`

Overloaded By

f, p

See Also

MuPAD Functions
`eval` | `evalAt` | `poly`

exp

Exponential function

Syntax

```
exp(x)
```

Description

`exp(x)` represents the value of the exponential function at the point x.

The exponential function is defined for all complex arguments.

For most exact arguments, an unevaluated function call is returned subject to some simplifications:

- Calls of the form $e^{q\,\pi\,i}$ with integer or rational q are rewritten such that q lies in the interval $[0, 2)$. Explicit results are returned if the denominator of q is 1, 2, 3, 4, 5, 6, 8, 10, or 12.
- Further, the following special values are implemented: $e^0 = 1$, $e^{\infty} = \infty$, $e^{-\infty} = 0$.
- A call of the form $e^{c\,\ln(y)}$ with an unevaluated $ln(y)$ and a constant c (i.e., of type `Type::Constant`) yields the result y^c.
- The call $e^{f(y)}$ yields the result $\frac{y}{f(y)}$, if f is `lambertW`.

Floating point results are computed, when the argument is a floating-point number.

Note: Numerical exceptions may happen, when the absolute value of the real part of a floating-point argument x is large. If $\Re(x) < -7.4\,10^8$, then `exp(x)` may return the truncated result `0.0` (protection against underflow). If $\Re(x) > 7.4\,10^8$, then `exp(x)` may return the floating-point equivalent RD_INF of `infinity`. See “Example 2” on page 1-681.

For arguments of type DOM_INTERVAL, the return value is another interval containing the image set of the exponential function over the input interval. See “Example 4” on page 1-682.

The protected identifier E is an alias for exp(1).

Environment Interactions

When called with a floating-point argument, the function is sensitive to the environment variable DIGITS which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
exp(1), exp(2), exp(-3), exp(1/4), exp(1 + I), exp(x^2)
```

$$e,\ e^2,\ e^{-3},\ e^{\frac{1}{4}},\ e^{1+i},\ e^{x^2}$$

Floating point values are computed for floating-point arguments:

```
exp(1.23), exp(4.5 + 6.7*I), exp(1.0/10^20), exp(123456.7)
```

$$3.421229536,\ 82.31014791 + 36.44342846\,i,\ 1.0,\ 3.660698702\ 10^{53616}$$

Some special symbolic simplifications are implemented:

```
exp(I*PI), exp(x - 22*PI*I), exp(3 + I*PI)
```

$$-1,\ e^x,\ -e^3$$

```
exp(ln(-2)), exp(ln(x)*PI), exp(lambertW(5))
```

$$-2,\ x^{\pi},\ \frac{5}{W_0(5)}$$

Example 2

The truncated result `0.0` may be returned for floating-point arguments with negative real parts. This prevents numerical underflow:

```
exp(-742261118.6)
```

$$1.14996558\ 10^{-322359908}$$

```
exp(-744261118.7)
```

$$0.0$$

```
exp(-742261118.6 + 10.0^10*I), exp(-744261118.7 + 10.0^10*I)
```

$$1.004057513\ 10^{-322359908} - 5.606151488\ 10^{-322359909}\ i,\ 0.0$$

When internal numerical overflow occurs, the floating-point equivalent RD_INF of `infinity` is returned:

```
exp(744261117.2)
```

$$1.972919601\ 10^{323228496}$$

```
exp(744261117.3)
```

$$RD\_INF$$

Example 3

System functions such as `limit`, `series`, `expand`, `combine` etc. handle expressions involving `exp`:

```
limit(x*exp(-x), x = infinity), series(exp(x/(x + 1)), x = 0)
```

$$0,\ 1 + x - \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} - \frac{19\,x^5}{120} + O(x^6)$$

```
expand(exp(x + y + (sqrt(2) + 5)*PI*I))
```

$$-e^{\pi\sqrt{2}\,i}\,e^{x}\,e^{y}$$

```
combine(%, exp)
```

$$-e^{x+y+\pi\sqrt{2}\,i}$$

Example 4

`exp` transforms intervals (of type `DOM_INTERVAL`) to intervals:

```
exp(-1 ... 1)
```

$$0.3678794411 \ldots 2.718281829$$

Note that the MuPAD floating-point numbers cannot be arbitrarily large. In the context of floating-point intervals, all values larger than a machine-dependent constant are regarded as “infinite”:

```
exp(1 ... 1e1000)
```

$$2.718281828 \ldots \mathrm{RD\_INF}$$

Finally, we would like to mention that you can also use `exp` on disjunct unions of intervals:

```
exp((1 ... PI) union (10 ... 20))
```

$$2.718281828 \ldots 23.14069264 \cup 22026.46579 \ldots 485165195.5$$

Parameters

x

An arithmetical expression or a floating-point interval

Return Values

Arithmetical expression or a floating-point interval

Overloaded By

x

See Also

MuPAD Functions
`ln` | `log`

expand

Expand an expression

Syntax

```
expand(f, options)
```

```
expand(f, g1, g2, …, options)
```

Description

`expand(f)` expands the arithmetical expression `f`.

The most important use of `expand` is the application of the distributivity law to rewrite products of sums as sums of products. In this respect, `expand` is the inverse function of `factor`.

The numerator of a fraction is expanded, and then the fraction is rewritten as a sum of fractions with simpler numerators; see “Example 1” on page 1-685. In a certain sense, this is the inverse functionality of `normal`. Use `partfrac` for a more powerful way to rewrite a fraction as a sum of simpler fractions.

`expand(f)` applies the following rules when rewriting powers occurring as subexpressions in `f`:

- $x^{a+b} = x^a x^b$.
- If `b` is an integer, or $x \geq 0$ or $y \geq 0$, then $(x\,y)^b = x^b\,y^b$.
- If `b` is an integer, then $(x^a)^b = x^{a\,b}$.

Except for the third rule, this behavior of `expand` is the inverse functionality of `combine`. See “Example 2” on page 1-686.

`expand` works recursively on the subexpressions of an expression `f`. If `f` is of the container type `array` or `table`, `expand` only returns `f` and does not map on the entries. To expand all entries of one of the containers, use `map`. See “Example 3” on page 1-686.

If optional arguments `g1, g2, ...` are present, then any subexpression of `f` that is equal to one of these additional arguments is not expanded; see “Example 4” on page 1-687. See section “Background” for a description how this works.

Properties of identifiers are taken into account (see `assume`). Identifiers without any properties are assumed to be complex. See “Example 9” on page 1-690.

`expand` also handles various types of special mathematical functions. It rewrites a single call of a special function with a complicated argument as a sum or a product of several calls of the same function or related functions with simpler arguments. In this respect, `expand` is the inverse function of `combine`.

In particular, `expand` implements the functional equations of the exponential function and the logarithm, the gamma function and the polygamma function, and the addition theorems for the trigonometric functions and the hyperbolic functions. See “Example 10” on page 1-691.

Environment Interactions

`expand` is sensitive to properties of identifiers set via `assume`.

Examples

Example 1

`expand` expands products of sums by multiplying out:

```
expand((x + 1)*(y + z)^2)
```

$$2yz + xy^2 + xz^2 + y^2 + z^2 + 2xyz$$

After expansion of the numerator, a fraction is rewritten as a sum of fractions:

```
expand((x + 1)^2*y/(y + z)^2)
```

$$\frac{y}{y^2 + 2yz + z^2} + \frac{2xy}{y^2 + 2yz + z^2} + \frac{x^2 y}{y^2 + 2yz + z^2}$$

Example 2

A power with a sum in the exponent is rewritten as a product of powers:

```
expand(x^(y + z + 2))
```

$$x^y\, x^z\, x^2$$

Example 3

`expand` works in a recursive fashion. In the following example, the power $(x + y)^{z+2}$ is first expanded into a product of two powers. Then the power $(x + y)^2$ is expanded into a sum. Finally, the product of the latter sum and the remaining power $(x + y)^z$ is multiplied out:

```
expand((x + y)^(z + 2))
```

$$x^2\,(x+y)^z + y^2\,(x+y)^z + 2\,x\,y\,(x+y)^z$$

Here is another example:

```
expand(2^((x + y)^2))
```

$$2^{x^2}\, 2^{y^2}\, 2^{2\,x\,y}$$

`expand` maps on the entries of lists, sets, and matrices:

```
expand([(a + b)^2, (a - b)^2]);
expand({(a + b)^2, (a - b)^2});
expand(matrix([[(a + b)^2, 0],[0, (a - b)^2]]))
```

$$[a^2 + 2\,a\,b + b^2,\ a^2 - 2\,a\,b + b^2]$$

$$\{a^2 - 2\,a\,b + b^2,\ a^2 + 2\,a\,b + b^2\}$$

$$\begin{pmatrix} a^2 + 2\,a\,b + b^2 & 0 \\ 0 & a^2 - 2\,a\,b + b^2 \end{pmatrix}$$

`expand` does not map on the entries of tables or arrays:

```
expand(table((a + b)^2=(c + 1)^2)),
expand(array(1..1, [(a + b)^2]))
```

$$\begin{array}{|c|}\hline (a+b)^2 \\ \hline (c+1)^2 \\ \hline \end{array}, \left((a+b)^2 \right)$$

Use `map` in order to expand all entries of a container:

```
map(table((a + b)^2=(c + 1)^2), expand),
map(array(1..1, [(a + b)^2]), expand)
```

$$\begin{array}{|c|}\hline (a+b)^2 \\ \hline 2\,c+c^2+1 \\ \hline \end{array}, \left(a^2+2\,a\,b+b^2 \right)$$

Note that this call expands only the *entries* in a table, not the keys. In the (rare) case that you want the keys expanded as well, transform the table to a list or set of equations first:

```
T := table((a + b)^2=(c + 1)^2):
table(expand([op(T)]))
```

$$\begin{array}{|c|}\hline 2\,a\,b+a^2+b^2 \\ \hline 2\,c+c^2+1 \\ \hline \end{array}$$

Example 4

If additional arguments are provided, `expand` performs only a partial expansion. These additional expressions, such as `x + 1` in the following example, are not expanded:

```
expand((x + 1)*(y + z))
```

$$y+z+x\,y+x\,z$$

```
expand((x + 1)*(y + z), x + 1)
```

$$y\,(x+1)+z\,(x+1)$$

Example 5

By default, `expand` works on all subexpressions including trigonometric subexpressions:

```
e := (sin(2*x) + 1)*(1 - cos(2*x)):
expand(e)
```

$$-4\sin(x)\cos(x)^3 - 2\cos(x)^2 + 4\sin(x)\cos(x) + 2$$

To prevent expansion of subexpressions, use the `ArithmeticOnly` option:

```
expand(e, ArithmeticOnly)
```

$$\sin(2x) - \cos(2x) - \cos(2x)\sin(2x) + 1$$

The option does not prevent expansion of powers and roots:

```
expand((sin(2*x) + 1)^3, ArithmeticOnly)
```

$$\sin(2x)^3 + 3\sin(2x)^2 + 3\sin(2x) + 1$$

To keep subexpressions with integer powers unexpanded, use the `MaxExponent` option.

Example 6

The `IgnoreAnalyticConstraints` option applies a set of purely algebraic simplifications including the equality of sum of logarithms and a logarithm of a product. Using the `IgnoreAnalyticConstraints` option, you get a simpler result, but one that might be incorrect for some of the values of variables:

```
expand(ln(a*b*c*d), IgnoreAnalyticConstraints)
```

$$\ln(a) + \ln(b) + \ln(c) + \ln(d)$$

Without using this option, you get a mathematically correct result:

```
expand(ln(a*b*c*d))
```

$$\ln(a\,b\,c\,d)$$

Example 7

If the additional `MaxExponent` provided, `expand` performs only a partial expansion. Powers with an integer exponent larger than the given bound, are not expanded:

```
expand((a + b)^3, MaxExponent = 2)
```

$$(a+b)^3$$

If the exponent is smaller or equal the given bound, the power is expanded:

```
expand((a + b)^2, MaxExponent = 2)
```

$$a^2+2\,a\,b+b^2$$

Example 8

The `expand` function can accept several options simultaneously. Suppose you want to expand the following expression:

```
e := (sin(2*x) + 1)*(x + 1)^3
```

$$(\sin(2\,x)+1)\,(x+1)^3$$

`expand` without any options works recursively. The function expands all subexpressions including trigonometric functions and powers:

```
expand(e)
```

$$3\,x+2\,\cos(x)\,\sin(x)+3\,x^2+x^3+6\,x\,\cos(x)\,\sin(x)+6\,x^2\,\cos(x)\,\sin(x)+2\,x^3\,\cos(x)\,\sin(x)+1$$

The `ArithmeticOnly` option prevents the expansion of the term `sin(2x)`. The `MaxExponent` option prevents the expansion of $(x + 1)^3$:

```
expand(e, ArithmeticOnly);
expand(e, MaxExponent = 2)
```

$$3\,x+\sin(2\,x)+3\,x\,\sin(2\,x)+3\,x^2\,\sin(2\,x)+x^3\,\sin(2\,x)+3\,x^2+x^3+1$$

$$(x+1)^3 + 2\cos(x)\sin(x)(x+1)^3$$

Combining these options in one call of the `expand` function, you apply both restrictions for the expansion:

```
expand(e, MaxExponent = 2, ArithmeticOnly)
```

$$(x+1)^3 + \sin(2x)(x+1)^3$$

Example 9

The following expansions are not valid for all values `a`, `b` from the complex plane. Therefore, MuPAD does not expand these expressions:

```
expand(ln(a^2)), expand(ln(a*b)), expand((a*b)^n)
```

$$\ln(a^2),\ \ln(a\,b),\ (a\,b)^n$$

The expansions are valid under the assumption that `a` is a positive real number:

```
assume(a > 0):  expand(ln(a^2)), expand(ln(a*b)), expand((a*b)^n)
```

$$2\ln(a),\ \ln(a)+\ln(b),\ a^n\,b^n$$

Clear the assumption for further computations:

```
unassume(a):
```

Alternatively, to get the expanded result for the third expression, assume that `n` is an integer:

```
expand((a*b)^n) assuming n in Z_
```

$$a^n\,b^n$$

Use the `IgnoreAnalyticConstraints` option to expand these expressions without explicitly specified assumptions:

```
expand(ln(a^2), IgnoreAnalyticConstraints),
expand(ln(a*b), IgnoreAnalyticConstraints),
```

```
expand((a*b)^n, IgnoreAnalyticConstraints)
```

$$2\ln(a),\ \ln(a)+\ln(b),\ a^n\,b^n$$

Example 10

The addition theorems of trigonometry are implemented by `"expand"`-slots of the trigonometric functions `sin` and `cos`:

```
expand(sin(a + b)), expand(sin(2*a))
```

$$\cos(a)\sin(b)+\cos(b)\sin(a),\ 2\cos(a)\sin(a)$$

The same is true for the hyperbolic functions `sinh` and `cosh`:

```
expand(cosh(a + b)), expand(cosh(2*a))
```

$$\sinh(a)\sinh(b)+\cosh(a)\cosh(b),\ 2\cosh(a)^2-1$$

The exponential function with a sum as argument is expanded via `exp::expand`:

```
expand(exp(a + b))
```

$$e^a\,e^b$$

Here are some more expansion examples for the functions `sum`, `fact`, `abs`, `coth`, `sign`, `binomial`, `beta`, `gamma`, `cot`, `tan`, `exp` and `psi`:

```
sum(f(x) + g(x),x);  expand(%)
```

$$\sum_x (f(x)+g(x))$$

$$\left(\sum_x f(x)\right)+\left(\sum_x g(x)\right)$$

```
fact(x + 1);  expand(%)
```

$$(x+1)!$$

$$x!\,(x+1)$$

```
abs(a*b);  expand(%)
```

$$|a\,b|$$

$$|a|\,|b|$$

```
coth(a + b);  expand(%)
```

$$\coth(a+b)$$

$$\frac{\coth(a)\,\coth(b)+1}{\coth(a)+\coth(b)}$$

```
coth(a*b);  expand(%)
```

$$\coth(a\,b)$$

$$\coth(a\,b)$$

```
sign(a*b);  expand(%)
```

$$\operatorname{sign}(a\,b)$$

$$\operatorname{sign}(a)\,\operatorname{sign}(b)$$

```
binomial(n, m);  expand(%)
```

$$\binom{n}{m}$$

$$-\frac{n\,\Gamma(n)}{m^2\,\Gamma(n-m)\,\Gamma(m)-m\,n\,\Gamma(n-m)\,\Gamma(m)}$$

```
beta(n, m);  expand(%)
```

$$\beta(m, n)$$

$$\frac{\Gamma(m)\,\Gamma(n)}{\Gamma(m+n)}$$

```
gamma(x + 1);  expand(%)
```

$$\Gamma(x+1)$$

$$x\,\Gamma(x)$$

```
tan(a + b); expand(%)
```

$$\tan(a+b)$$

$$-\frac{\tan(a)+\tan(b)}{\tan(a)\,\tan(b)-1}$$

```
cot(a + b); expand(%)
```

$$\cot(a+b)$$

$$\frac{\cot(a)\,\cot(b)-1}{\cot(a)+\cot(b)}$$

```
exp(x + y);  expand(%)
```

$$e^{x+y}$$

$$e^{x}\,e^{y}$$

```
psi(x + 2);  expand(%)
```

$$\psi(x+2)$$

$$\psi(x)+\frac{1}{x+1}+\frac{1}{x}$$

In contrast to previous versions of MuPAD, `expand` does not rewrite `tan` in terms of `sin` and `cos`:

```
expand(tan(a))
```

$$\tan(a)$$

Example 11

This example illustrates how to extend the functionality of `expand` to user-defined mathematical functions. As an example, we consider the sine function. (Of course, the system function `sin` already has an `"expand"` slot; see “Example 10” on page 1-691.)

We first embed our function into a function environment, which we call `Sin`, in order not to overwrite the system function `sin`. Then we implement the addition theorem $sin(x + y) = sin(x)\,cos(y) + sin(y)\,cos(x)$ in the `"expand"` slot of the function environment, i.e., the slot routine `Sin::expand`:

```
Sin := funcenv(Sin):
Sin::expand := proc(u) // compute expand(Sin(u))
  local x, y;
begin
  // recursively expand the argument u
  u := expand(op(u));

   if type(u) = "_plus" then // u is a sum
    x := op(u, 1); // the first term
    y := u - x;    // the remaining terms

    // apply the addition theorem and
    // expand the result again
    expand(Sin(x)*cos(y) + cos(x)*Sin(y))
  else
    Sin(u)
  end_if
```

```
end_proc:
```

Now, if `expand` encounters a subexpression of the form `Sin(u)`, it calls `Sin::expand(u)` to expand `Sin(u)`. The following command first expands the argument `a*(b+c)` via the recursive call in `Sin::expand`, then applies the addition theorem, and finally `expand` itself expands the product of the result with `z`:

```
expand(z*Sin(a*(b + c)))
```

$$z\,\mathrm{Sin}(a\,b)\,\cos(a\,c) + z\,\mathrm{Sin}(a\,c)\,\cos(a\,b)$$

The expansion after the application of the addition theorem in `Sin::expand` is necessary to handle the case when `u` is a sum with more than two terms: then `y` is again a sum, and `cos(y)` and `Sin(y)` are expanded recursively:

```
expand(Sin(a + b + c))
```

$$\mathrm{Sin}(a)\,\cos(b)\,\cos(c) + \mathrm{Sin}(b)\,\cos(a)\,\cos(c) + \mathrm{Sin}(c)\,\cos(a)\,\cos(b) - \mathrm{Sin}(a)\,\sin(b)\,\sin(c)$$

Parameters

`f, g1, g2, …`

arithmetical expressions

Options

`ArithmeticOnly`

Expand arithmetic part of an expression without expanding trigonometric, hyperbolic, logarithmic and special functions.

This option does not prevent expansion of powers and roots. Technically, the option omits overloading the `expand` function for each term of the original expression. See “Example 5” on page 1-688.

`IgnoreAnalyticConstraints`

With this option `expand` applies the following rules when expanding expressions:

- $ln(a) + ln(b) = ln(a\,b)$ for all values of a and b. In particular:

  $(a\,b)^c = e^{c\,\ln(a\,b)} = e^{c\,(\ln(a)+\ln(b))} = a^c\,b^c$ for all values of a, b, and c
- $ln(a^b) = b\,ln(a)$ for all values of a and b. In particular:

  $(a^b)^c = e^{b\,c\,\ln(a)} = e^{\ln(a)^{b\,c}} = a^{b\,c}$ for all values of a, b, and c
- If f and g are standard mathematical functions and $f(g(x)) = x$ for all small positive numbers, $f(g(x)) = x$ is assumed to be valid for all complex x. In Particular:
  - $\ln(e^x) = x$
  - $arcsin(sin(x)) = x$, $arccos(cos(x)) = x$, $arctan(tan(x)) = x$
  - $arcsinh(sinh(x)) = x$, $arccosh(cosh(x)) = x$, $arctanh(tanh(x)) = x$
  - $W_k(x\,e^x) = x$ for all values of k

Using the option can give you simpler results for the expressions for which the default call to `expand` returns complicated results. With this option the function does not guarantee the equality of the initial expression and the result for all symbolic parameters. See “Example 6” on page 1-688.

`MaxExponent`

Option, specified as `MaxExponent = n`

Do not expand powers with integer exponents larger than `n`.

If you call `expand` with this option, the function expands does not expand powers with integer exponents larger than `n`. See “Example 7” on page 1-689.

Return Values

arithmetical expression.

Overloaded By

f

Algorithms

With optional arguments `g1, g2, ...`, the expansion of certain subexpressions of `f` can be prevented. This works as follows: every occurrence of `g1, g2, ...` in `f` is replaced by an auxiliary variable before the expansion, and afterwards the auxiliary variables are replaced by the original subexpressions.

Users can extend the functionality of `expand` to their own special mathematical functions via overloading. To this end, embed your function into a function environment `g` and implement the behavior of `expand` for this function in the `"expand"` slot of the function environment.

Whenever `expand` encounters a subexpression of the form `g(u,..)`, it issues the call `g::expand(g(u,..))` to the slot routine to expand the subexpression, passing the not yet expanded arguments `g(u,..)` as arguments. The result of this call is not expanded any further by `expand`. See “Example 11” on page 1-694 above.

Similarly, an `"expand"` slot can be defined for a user-defined library domain`T`. Whenever `expand` encounters a subexpression `d` of domain type`T`, it issues the call `T::expand(d)` to the slot routine to expand `d`. The result of this call is not expanded any further by `expand`. If `T` has no `"expand"` slot, then `d` remains unchanged.

See Also

MuPAD Functions

`collect` | `combine` | `denom` | `factor` | `normal` | `numer` | `partfrac` | `rationalize` | `rectform` | `rewrite` | `simplify`

expose

Display the source code of a procedure or the entries of a domain

Syntax

```
expose(f)
```

Description

`expose(f)` displays the source code of the MuPAD procedure `f` or the entries of the domain `f`.

Usually, procedures and domains are printed in abbreviated form. `expose` serves to display the complete source code of a procedure and all entries of a domain, respectively. However, you cannot use `expose` to look at the source code of kernel functions.

If `f` is a domain, then `expose` returns a symbolic `newDomain` call. The arguments of the call are equations of the form `index = value`, where `value` equals the value of `f::index`. `expose` is not recursively applied to `f::index`; hence, the source code of domain methods is not displayed.

Although `expose` returns a syntactically valid MuPAD object, this return value is intended for screen output only, and further processing of it is deprecated.

Examples

Example 1

Using `expose`, you can inspect the source code of procedures of the MuPAD library:

```
sin
```

```
sin
```

```
expose(%)

proc(x)   name sin;   local f, y;   option
noDebug; begin   if args(0) = 0 then     error("no arguments given")
  else   ... end_proc
```

Example 2

When applied to a domain, `expose` shows the entries of that domain:

```
expose(DOM_NULL)


domain DOM_NULL

  new := proc new() ... end;

  new_extelement := proc new_extelement(d) ... end;

  Content := proc DOM_NULL::Content(Out, x) ... end;
end_domain
```

Example 3

Applying `expose` to other objects is legal but generally useless:

```
expose(3)

     3
```

Parameters

`f`

Any object; typically, a procedure, a function environment, or a domain

Overloaded By

f

Algorithms

In addition to the usual overloading mechanism for domain elements, a domain method overloading `expose` must handle the following case: it will be called with zero arguments when the domain itself is to be exposed.

If `f` is a procedure, then `expose` returns an object of the domain `stdlib::Exposed`. The only purpose of this domain is its "print" method; manipulating its elements should never be necessary. Therefore it remains undocumented.

See Also

MuPAD Functions
`print`

expr

Convert into an element of a basic domain

Syntax

```
expr(object)
```

Description

`expr(object)` converts `object` into an element of a basic domain, such that all sub-objects are elements of basic domains as well.

`expr` is a type conversion function, for converting an element of a more complex library domain, such as a polynomial or a matrix, into an element of a basic kernel domain.

`expr` proceeds recursively, such that all sub-objects of the returned object are elements of basic domains, or infinities, or `undefined`. See “Example 1” on page 1-702 and “Example 2” on page 1-702.

The two special objects `infinity` and `complexInfinity` are translated into identifiers with the same name by `expr`. Evaluating these identifiers yields the original objects. See “Example 1” on page 1-702.

If `object` already belongs to a basic domain other than `DOM_POLY`, then `expr` is only applied recursively to the operands of `object`, if any.

If `object` is a polynomial of domain type`DOM_POLY`, then `expr` is applied recursively to the coefficients of `object`, and afterwards the result is converted into an identifier, a number, or an expression. See “Example 1” on page 1-702.

If `object` belongs to a library domain `T` with an `"expr"` slot, then the corresponding slot routine `T::expr` is called with `object` as argument, and the result is returned.

This can be used to extend the functionality of `expr` to elements of user-defined domains. If the slot routine is unable to perform the conversion, it must return `FAIL`. See “Example 6” on page 1-704.

If the domain `T` does not have an `"expr"` slot, then `expr` returns `FAIL`.

The result of `expr` is not evaluated further. Use `eval` to evaluate it. See "Example 4" on page 1-703.

Examples

Example 1

`expr` converts a polynomial into an expression, an identifier, or a number:

```
expr(poly(x^2 + y, [x])), expr(poly(x)), expr(poly(2, [x]));
map(%, domtype)
```

$$x^2 + y,\ x,\ 2$$

DOM_EXPR, DOM_IDENT, DOM_INT

`expr` also works with the objects `infinity`, `complexInfinity`, and `undefined`:

```
expr(infinity), expr(complexInfinity), expr(undefined);
map(%, domtype)
```

$$\infty,\ \text{complexInfinity},\ \text{undefined}$$

stdlib::Infinity, stdlib::CInfinity, stdlib::Undefined

Example 2

This example shows that `expr` works recursively on expressions. All subexpressions which are domain elements are converted into expressions. The construction with `hold(_plus)(..)` is necessary since `x + i(1)` would evaluate to `FAIL`:

```
i := Dom::IntegerMod(7):
hold(_plus)(x, i(1));  expr(%)
```

$$x + (1 \bmod 7)$$

$$x + 1$$

Example 3

The function `series` returns an element of the domain `Series::Puiseux`, which is not a basic domain:

```
s := series(sin(x), x);
domtype(s)
```

$$x - \frac{x^3}{6} + \frac{x^5}{120} + O(x^7)$$

Series::Puiseux

Use `expr` to convert the result into an element of domain type `DOM_EXPR`:

```
e := expr(s); domtype(e)
```

$$\frac{x^5}{120} - \frac{x^3}{6} + x$$

DOM_EXPR

Note that the information about the order term is lost after the conversion.

Example 4

`expr` does not evaluate its result. In this example the polynomial `p` has a parameter `a` and the global variable `a` has a value. `expr` applied on the polynomial `p` returns an expression containing `a`. If you want to insert the value of `a` use the function `eval`:

```
p := poly(a*x, [x]):  a := 2:  expr(p);  eval(%)
```

$$a\,x$$

$$2\,x$$

Example 5

A is an element of type `Dom::Matrix(Dom::Integer)`:

```
A := Dom::Matrix(Dom::Integer)([[1, 2], [3, 2]]);
domtype(A)
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 2 \end{pmatrix}$$

Dom::Matrix(Dom::Integer)

In this case, `expr` converts A into an element of type `DOM_ARRAY`:

```
a := expr(A); domtype(a)
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 2 \end{pmatrix}$$

DOM_ARRAY

However, it is not guaranteed that the result is of type `DOM_ARRAY` in future versions of MuPAD as well. For example, the internal representation of matrices might change in the future. Use `coerce` to request the conversion into a particular data type:

```
coerce(A, DOM_ARRAY)
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 2 \end{pmatrix}$$

A nested `list` is an alternative representation for a matrix:

```
coerce(A, DOM_LIST)
```

[[1, 2], [3, 2]]

Example 6

If a sub-object belongs to a domain without an `"expr"` slot, then `expr` returns `FAIL`:

```
T := newDomain("T"):
d := new(T, 1, 2);
expr(d)
```

$$\mathrm{new}(T, 1, 2)$$

FAIL

You can extend the functionality of `expr` to your own domains. We demonstrate this for the domain `T` by implementing an `"expr"` slot, which returns a list with the internal operands of its argument:

```
T::expr := x -> [extop(x)]:
```

If now `expr` encounters a sub-object of type `T` during the recursive process, it calls the slot routine `T::expr` with the sub-object as argument:

```
expr(d), expr([d, 3])
```

$$[1, 2], [[1, 2], 3]$$

Parameters

`object`

An arbitrary object

Return Values

Element of a basic domain.

Overloaded By

object

See Also

MuPAD Functions
`coerce` | `domtype` | `eval` | `testtype` | `type`

expr2text

Convert objects into character strings

Syntax

```
expr2text(object)
```

Description

`expr2text(object)` converts `object` into a character string. The result usually corresponds to the screen output of `object` when `PRETTYPRINT` is set to `FALSE`.

If the function is called without arguments, then an empty character string is created. If more than one argument is given, the arguments are interpreted as an expression sequence and are converted into a single character string.

Like most other MuPAD function, `expr2text` evaluates its arguments before the conversion.

If strings occur in `object`, they will be quoted in the result.

Examples

Example 1

Expressions are converted into character strings:

```
expr2text(a + b)
```

```
"a + b"
```

`expr2text` quotes strings. Note that the quotation marks are preceded by a backslash when they are printed on the screen:

```
expr2text(["text", 2])
```

```
"["text", 2]"
```

Example 2

If more than one argument is given, the arguments are treated as a single expression sequence:

```
expr2text(a, b, c)
```

```
"a, b, c"
```

If no argument is given, an empty string is generated:

```
expr2text()
```

```
""
```

Example 3

`expr2text` evaluates its arguments:

```
a := b: c := d: expr2text(a, c)
```

```
"b, d"
```

Use `hold` to prevent evaluation:

```
expr2text(hold(a, c));
delete a, c:
```

```
"a, c"
```

Here is another example:

```
expr2text((a := b; c := d));
delete a, c:
```

```
"d"
```

```
e := expr2text(hold((a := b; c := d)))
```

```
"(a := b;  c := d)"
```

The last string contains a newline character "\n". Use `print` with option `Unquoted` to expand this into a new line:

```
print(Unquoted, e):
```

```
(a := b;
c := d)
```

Example 4

`expr2text` is overloadable. It uses a default output for elements of a domain if the domain has neither a `"print"` slot nor an `"expr2text"` slot:

```
T := newDomain("T"): e := new(T, 1):
e;
print(e):
expr2text(e)
```

```
new(T, 1)
```

```
new(T, 1)
```

```
"new(T, 1)"
```

If a `"print"` slot exists, it will be called by `expr2text` to generate the output:

```
T::print := proc(x) begin
  _concat("foo: ", expr2text(extop(x)))
end_proc:
e;
print(e):
expr2text(e)
```

```
foo: 1
```

```
foo: 1
```

```
"foo: 1"
```

If you want `expr2text` to generate an output differing from the usual output generated by `print`, you can supply an `"expr2text"` method:

```
T::expr2text := proc(x) begin
  _concat("bar: ", expr2text(extop(x)))
end_proc:
e;
print(e):
expr2text(e)
```

```
foo: 1
```

```
foo: 1
```

```
"bar: 1"
```

Parameters

`object`

Any MuPAD object

Return Values

`string`.

Overloaded By

object

Algorithms

When processing a domain element `e`, `expr2text` first tries to call the `"expr2text"` method of the corresponding domain `T`. If it exists, `T::expr2text(e)` is called and the result is returned. If no `"expr2text"` method exists, `expr2text` tries to call the `"print"` method in the same way. If no `"print"` method exists either, `expr2text` will generate a default output. Cf. “Example 4” on page 1-709.

An `"expr2text"` method or a `"print"` method may return an arbitrary MuPAD object, which will be processed recursively by `expr2text`.

Note: The returned object must not contain the domain element `e` as a sub-object. Otherwise, the MuPAD kernel runs into infinite recursion and emits an error message.

For expressions, the result returned by `expr2text` always coincides with the output produced by `print`. If the 0th operand of the expression is a function environment, the result of `expr2text` is computed by the second operand of the function environment.

See Also

MuPAD Functions

`coerce` | `fprint` | `int2text` | `print` | `tbl2text` | `text2expr` | `text2int` | `text2list` | `text2tbl`

extnops

Number of operands of the internal representation a domain element

Syntax

```
extnops(object)
```

Description

`extnops(object)` returns the number of operands of the object's internal representation.

For objects of a basic data type such as expressions, sets, lists, tables, arrays, hfarrays etc., `extnops` yields the same result as the function `nops`. The only difference to the function `nops` is that `extnops` cannot be overloaded by domains implemented in the MuPAD language.

Internally, a domain element may consist of an arbitrary number of data objects; `extnops` returns the actual number of *internal* operands. Since every domain should provide interface methods, `extnops` should only be used from inside these methods. “From the outside”, the function `nops` should be used.

Examples

Example 1

`extnops` returns the number of entries of a domain element:

```
d := newDomain("demo"): e := new(d, 1, 2, 3, 4): extnops(e)
```

```
    4
```

```
delete d, e:
```

Example 2

For kernel domains, `extnops` is equivalent to `nops`:

```
extnops([1, 2, 3, 4]), nops([1, 2, 3, 4])
```

```
4, 4
```

Example 3

We define a domain of lists. Its internal representation is a single object (a list of kernel type `DOM_LIST`):

```
myList := newDomain("lists"):
myList::new := proc(l : DOM_LIST) begin new(myList, l) end_proc:
```

We want the functionality of `nops` for this domain to be the same as for the kernel type `DOM_LIST`. To achieve this, we overload the function `nops`. The internal list is accessed via `extop(l, 1)`:

```
myList::nops := l -> nops(extop(l, 1)):
```

We create an element of this domain:

```
mylist := myList([1, 2, 3])
```

```
new(lists, [1, 2, 3])
```

Since `nops` was overloaded, `extnops` provides the only way of determining the number of operands of the internal representation of `mylist`. In contrast to `nops`, `extnops` always returns 1, because the internal representation consists of exactly one list:

```
nops(mylist), extnops(mylist)
```

```
3, 1
```

```
delete myList, mylist:
```

Parameters

`object`

An arbitrary MuPAD object

Return Values

Nonnegative integer.

See Also

MuPAD Domains
`DOM_DOMAIN`

MuPAD Functions
`extop` | `extsubsop` | `new` | `nops` | `op` | `subsop`

extop

Internal operands of a domain element

Syntax

```
extop(object)

extop(object, i)

extop(object, i .. j)
```

Description

`extop(object)` returns all operands of the domain element `object`.

`extop(object, i)` returns the `i`-th operand.

`extop(object, i..j)` returns the `i`-th to `j`-th operand.

For objects of a basic data type such as expressions, sets, lists, tables, arrays, hfarrays etc., `extop` yields the same operands as the function `op`. See the corresponding documentation for details on operands. The main difference to the function `op` is that `extop` cannot be overloaded. Therefore, it guarantees direct access to the operands of the *internal representation* of elements of a library domain. Typically, `extop` is used in the implementation of the `"op"` method of a library domain that overloads the system's `op` function.

A domain element consists of a reference to the corresponding domain and a sequence of values representing its contents. The function `extop` allows access to the domain and the operands of this internal data sequence.

`extop(object)` returns a sequence of all internal operands except the 0-th one. This call is equivalent to `extop(object, 1..extnops(object))`.

`extop(object, i)` returns the `i`-th internal operand. In particular, the domain of the object is returned by `extop(object, 0)` if `object` is an element of a library domain. If `object` is an element of a kernel domain, the call `extop(object, 0)` is equivalent to `op(object, 0)`.

`extop(object, i..j)` returns the `i`-th to `j`-th internal operands of `object` as an expression sequence; `i` and `j` must be nonnegative integers with `i` smaller or equal to `j`. This sequence is equivalent to `extop(object, k) $k = i..j`.

`extop` returns `FAIL` if a specified operand does not exist. Cf. “Example 4” on page 1-718.

The operands of an expression sequence are its elements. Note that such sequences are not flattened by `extop`.

Examples

Example 1

We create a new domain `d` and use the function `new` to create an element of this type. Its internal data representation is the sequence of arguments passed to `new`:

```
d := newDomain("demo"): e := new(d, 1, 2, 3): extop(e)
```

```
1, 2, 3
```

Individual operands can be selected:

```
extop(e, 2)
```

```
2
```

Ranges of operands can be selected:

```
extop(e, 1..2)
```

```
1, 2
```

The 0-th operand of a domain element is its domain:

```
extop(e, 0)
```

```
demo
```

```
delete d, e:
```

Example 2

First, a new domain `d` is defined via `newDomain`. The `"new"` method serves for creating elements of this type. The internal representation of the domain is a sequence of all arguments of this `"new"` method:

```
d := newDomain("d"): d::new := () -> new(dom, args()):
```

The system's `op` function is overloaded by the following `"op"` method of this domain. It is to return the elements of a sorted copy of the internal data sequence. In the implementation of the `"op"` method, the function `extop` is used to access the internal data:

```
d::op := proc(x, i = null())
         local internalData;
         begin internalData := extop(x);
               op(sort([internalData]), i)
         end_proc:
```

Due to this overloading, `op` returns different operands than `extop`:

```
e := d(3, 7, 1): op(e); extop(e)
```

$$1, 3, 7$$

$$3, 7, 1$$

```
delete d, e:
```

Example 3

For kernel data types such as sets, lists etc., `extop` always returns the same operands as `op`:

```
extop([a, b, c]) = op([a, b, c])
```

$$(a, b, c) = (a, b, c)$$

Expressions are of kernel data type DOM_EXPR, thus extop(sin(x), 0) is equivalent to op(sin(x), 0):

```
domtype(sin(x)), extop(sin(x), 0) = op(sin(x), 0)
```

```
DOM_EXPR, sin = sin
```

Expression sequences are not flattened:

```
extop((1, 2, 3), 0), extop((1, 2, 3))
```

```
_exprseq, 1, 2, 3
```

Example 4

Non-existing operands are returned as FAIL:

```
extop([1, 2], 4),  extop([1, 2], 1..4)
```

```
FAIL, FAIL
```

Parameters

object

An arbitrary MuPAD object

i, j

Nonnegative integers

Return Values

sequence of operands or the specified operand. FAIL is returned if no corresponding operand exists.

See Also

MuPAD Domains
DOM_DOMAIN

MuPAD Functions
extnops | extsubsop | new | nops | op | subsop

extsubsop

Substitute operands of a domain element

Syntax

```
extsubsop(d, i1 = new1, i2 = new2, …)
```

Description

`extsubsop(d, i = new)` returns a copy of the domain element `d` with the `i`-th operand of the internal representation replaced by `new`.

Internally, a domain element may consist of an arbitrary number of objects. `extsubsop` replaces one or more of these objects, without checking whether the substitution is meaningful.

Note: The operands of elements of domains of the MuPAD library must meet certain (undocumented) conditions; use `extsubsop` only for your own domains. It is good programming style to use `extsubsop` only inside low-level domain methods.

`extsubsop` returns a modified copy of the object, but does not change the object itself.

The numbering of operands is the same as the one used by `extop`.

If the 0-th operand is to be replaced, the corresponding new value must be a domain of type `DOM_DOMAIN`; `extsubsop` then replaces the domain of `d` by this new domain.

When trying to replace the `i`-th operand with `i` exceeding the actual number of operands, `extsubsop` first increases the number of operands by appending as many `NIL`'s as necessary and then performs the substitution. Cf. “Example 3” on page 1-722.

When the `i`-th operand is replaced by an expression sequence of `k` elements, each of these elements becomes an individual operand of the result, indexed from `i` to `i+k-1`. The remaining operands of `d` are shifted to the right accordingly. This new numbering is already in effect for the remaining substitutions in the same call to `extsubsop`. Cf. “Example 4” on page 1-722.

The void object `null()` becomes an operand of the result when it is substituted into an object.

After performing the substitution, `extsubsop` does not evaluate the result once more. Cf. “Example 5” on page 1-722.

In contrast to the function `subsop`, `extsubsop` cannot be overloaded.

Like `extop` and `extnops`, `extsubsop` can be applied to objects of a kernel domain. In this case `extsubsop` behaves like `subsop`.

Examples

Example 1

We create a domain element and then replace its first operand:

```
d := newDomain("1st"): e := new(d, 1, 2, 3): extsubsop(e, 1 = 5)
```

$$\text{new}(1\text{st}, 5, 2, 3)$$

This does not change the value of `e`:

```
e
```

$$\text{new}(1\text{st}, 1, 2, 3)$$

```
delete d, e:
```

Example 2

The domain type of an element can be changed by replacing its 0-th operand:

```
d := newDomain("some_domain"): e := new(d, 2):
extsubsop(e, 0 = Dom::IntegerMod(5))
```

$$2 \bmod 5$$

```
delete d, e:
```

Example 3

We substitute the sixth operand of a domain element that has less than six operands. In such cases, an appropriate number of `NIL`'s is inserted:

```
d := newDomain("example"): e := new(d, 1, 2, 3, 4):
extsubsop(e, 6 = 8)
```

$$\text{new}(\text{example}, 1, 2, 3, 4, \text{NIL}, 8)$$

```
delete d, e:
```

Example 4

We substitute the first operand of a domain element `e` by a sequence with three elements. These become the first three operands of the result; the second operand of `e` becomes the fourth operand of the result, and so on. This new numbering is already in effect when the second substitution is carried out:

```
d := newDomain("example"): e := new(d, 1, 2, 3, 4):
extsubsop(e, 1 = (11, 13, 17), 2 = (29, 99))
```

$$\text{new}(\text{example}, 11, 29, 99, 17, 2, 3, 4)$$

```
delete d, e:
```

Example 5

We define a domain with its own evaluation method. This method prints out its argument such that we can see whether it is called. Then we define an element of our domain.

```
d := newDomain("anotherExample"):
d::evaluate := x -> (print("Argument:", x); x):
e := new(d, 3)
```

$$\text{new}(\text{anotherExample}, 3)$$

We can now watch all evaluations that happen: `extsubsop` evaluates its arguments, performs the desired substitution, but does not evaluate the result of the substitution:

```
extsubsop(e, 1 = 0)
```

```
"Argument:", new(anotherExample, 3)

new(anotherExample, 0)
```

```
delete d, e:
```

Example 6

`extsubsop` applied to an object from a kernel type yields the same result as `subsop`:

```
extsubsop([1,2,3], 2=4), subsop([1,2,3], 2=4)
```

```
[1, 4, 3], [1, 4, 3]
```

Parameters

`d`

Arbitrary MuPAD object

`i1, i2, …`

Nonnegative integers

`new1, new2, …`

Arbitrary MuPAD objects

Return Values

Input object with replaced operands.

See Also

MuPAD Domains
DOM_DOMAIN

MuPAD Functions
extnops | extop | new | nops | op | subs | subsex | subsop

!, fact

Factorial function

Syntax

```
n !
```

```
fact(n)
```

Description

`fact(n)` represents the factorial $n! = 1\,2\,3\,\ldots$ of an integer.

The short hand call `n!` is equivalent to `fact(n)`.

If `n` is a nonnegative integer smaller than the value returned by `Pref::autoExpansionLimit()`, then an integer is returned. If `n` is a numerical value that is not an integer, then an error occurs. If `n` is a symbolic expression, then a symbolic call of `fact` is returned.

Use `expand(n!)` to compute an explicit result for large integers `n` equal to or larger than `Pref::autoExpansionLimit()`.

The `gamma` function generalizes the factorial function to arbitrary complex arguments. It satisfies `gamma(n+1) = n!` for nonnegative integers `n`. Expressions involving symbolic `fact` calls can be rewritten by `rewrite(expression, gamma)`. Cf. “Example 3” on page 1-726.

The operator `!` can also be used in prefix notation with an entirely different meaning: `! command` is equivalent to `system("command")`.

Examples

Example 1

Integer numbers are produced if the argument is a nonnegative integer:

```
fact(0), fact(5), fact(2^5)
```

$$1,\ 120,\ 263130836933693530167218012160000000$$

A symbolic call is returned if the argument is a symbolic expression:

```
fact(n), fact(n - sin(x)), fact(3.0*n + I)
```

$$n!,\ (n-\sin(x))!,\ (3.0\,n+\mathrm{i})!$$

The calls `fact(n)` and `n!` are equivalent:

```
5! = fact(5), fact(n^2 + 3)
```

$$120 = 120,\ (n^2+3)!$$

Example 2

Use `gamma(float(n+1))` rather than `float(fact(n))` for floating-point approximations of large factorials. This avoids the costs of computing large integer numbers:

```
float(fact(2^13)) = gamma(float(2^13 + 1))
```

$$1.275885799\ 10^{28503} = 1.275885799\ 10^{28503}$$

Example 3

The functions `expand`, `limit`, `rewrite` and `series` handle expressions involving `fact`:

```
expand(fact(n^2 + 4))
```

$$(n^2)!\,(n^2+1)\,(n^2+2)\,(n^2+3)\,(n^2+4)$$

```
limit(fact(n)/exp(n), n = infinity)
```

$$\infty$$

```
rewrite(fact(2*n^2 + 1)/fact(n - 1), gamma)
```

$$\frac{\Gamma(2\,n^2+2)}{\Gamma(n)}$$

The Stirling formula is obtained as an asymptotic series:

```
series(fact(n), n = infinity, 3)
```

$$\frac{\sqrt{2}\,\sqrt{\pi}\,\sqrt{n}\,e^{-n}}{\left(\frac{1}{n}\right)^n}+\frac{\sqrt{2}\,\sqrt{\pi}\,e^{-n}}{12\,\sqrt{n}\left(\frac{1}{n}\right)^n}+O\left(\frac{e^{-n}}{n^{3/2}\left(\frac{1}{n}\right)^n}\right)$$

Parameters

n

An arithmetical expression representing a nonnegative integer

Return Values

Arithmetical expression.

Overloaded By

n

See Also

MuPAD Functions

`beta` | `binomial` | `gamma` | `igamma` | `pochhammer` | `psi`

!!, fact2

Double factorial function

Syntax

```
n !!
```

```
fact2(n)
```

Description

`fact2(n)` represents the double factorial of an integer. The double factorial is defined as $n!! = 2\ 4\ \ldots\ n$ for even positive integers and $n!! = 1\ 3\ \ldots\ n$ for odd positive integers.

The short hand call `n!!` is equivalent to `fact2(n)`.

`0!!` and `(-1)!!` both return `1`.

If `n` is an integer greater or equal to `-1` and smaller than the value given by `Pref::autoExpansionLimit`(), then an integer is returned. If `n` is an integer smaller than `-1` or a non-integer numerical value then an error occurs. If `n` is a symbolic expression, then a symbolic call of `fact2` is returned.

Use `expand(n!!)` to compute an explicit result for large integers `n` equal to or larger than `Pref::autoExpansionLimit`().

Expressions involving symbolic calls of `fact2` can be rewritten in terms of the `gamma` function by `rewrite(expression, gamma)`. Cf. “Example 2” on page 1-729.

Note that the double factorial `n!!` does *not* equal the iterated factorial `(n!)!`.

Examples

Example 1

Integer numbers are produced if the argument is an integer greater than or equal to - 1:

```
fact2(-1), fact2(0), fact2(5), fact2(16)
```

$$1,\ 1,\ 15,\ 10321920$$

A symbolic call is returned if the argument is a symbolic expression:

```
fact2(n), fact2(4.7*I*n)
```

$$n!!,\ (4.7\,n\,\mathrm{i})!!$$

The calls `fact2(n)` and `n!!` are equivalent:

```
5!! = fact2(5), fact2(n^2 + 3)
```

$$15 = 15,\ (n^2+3)!!$$

Example 2

The function `rewrite` can be used to rewrite expressions involving `fact2` in terms of the `gamma` function. In most cases, `Simplify` has to be used to obtain a simple result:

```
rewrite(n!!, gamma)
```

$$2^{\frac{n}{2}-\frac{(-1)^n}{4}+\frac{1}{4}}\,\pi^{\frac{(-1)^n}{4}-\frac{1}{4}}\,\Gamma\left(\frac{n}{2}+1\right)$$

```
rewrite(fact2(2*n)/fact2(2*n - 1), gamma)
```

$$\frac{2^{\frac{(-1)^{2n-1}}{4}-n+\frac{1}{4}}\,2^{n-\frac{(-1)^{2n}}{4}+\frac{1}{4}}\,\pi^{\frac{(-1)^{2n}}{4}-\frac{1}{4}}\,\pi^{\frac{1}{4}-\frac{(-1)^{2n-1}}{4}}\,\Gamma(n+1)}{\Gamma\left(n+\frac{1}{2}\right)}$$

```
assume(n, Type::Integer): Simplify(%2)
```

$$\frac{2^{2n}\,n\,(n-1)!^2}{2\,\Gamma(2n)}$$

Example 3

For efficiency, the double factorial should be rewritten in terms of `gamma` if a floating-point evaluation for large arguments is desired. The following call produces a huge exact integer that is finally converted to a float:

```
float(fact2(2^17))
```

$$1.03441219\ 10^{306922}$$

The following call is much faster because no exact intermediate result is computed:

```
float(subs(rewrite(fact2(n), gamma), n = 2^17))
```

$$1.03441219\ 10^{306922}$$

Parameters

n

An arithmetical expression representing an integer greater than or equal to - 1.

Return Values

Arithmetical expression.

Overloaded By

n

See Also

MuPAD Functions

`beta` | `binomial` | `fact` | `gamma` | `igamma` | `psi`

factor

Factor a polynomial into irreducible polynomials

Syntax

```
factor(f, <Adjoin = adjoin>, <MaxDegree = n>)
```

```
factor(f, F | Domain = F | Full)
```

Description

`factor(f)` computes a factorization $f = u\, f_1^{e_1} \ldots f_r^{e_r}$ of the polynomial f, where u is the content of f, f_1, …, f_r are the distinct primitive irreducible factors of f, and e_1, …, e_r are positive integers.

`factor` rewrites its argument as a product of as many terms as possible. In a certain sense, it is the complementary function of `expand`, which rewrites its argument as a sum of as many terms as possible.

If `f` is a polynomial whose coefficient ring is not `Expr`, then `f` is factored over its coefficient ring. See “Example 10” on page 1-739.

If `f` is a polynomial with coefficient ring `Expr`, then `f` is factored over the smallest ring containing the coefficients. Mathematically, this *implied coefficient ring* always contains the ring $\mathbb{Z}$ of integers. See “Example 4” on page 1-736. If the coefficient ring `R` of `f` is not `Expr`, then we say that the implied coefficient ring is `R`. Elements of the implied coefficient ring are considered to be constants and are not factored any further. In particular, the content u is an element of the implied coefficient ring.

With the option `Adjoin`, the elements of `adjoin` are also adjoined to the coefficient ring.

If the second argument `F` or, alternatively, `Domain = F`is given, then f is factored over the real numbers $\mathbb{R}$ or the complex numbers $\mathbb{C}$. Factorization over $\mathbb{R}$ or $\mathbb{C}$ is performed using numerical calculations and the results will contain floating-point numbers. See “Example 5” on page 1-737.

If `f` is an arithmetical expression but not a number, it is considered as a rational expression. Non-rational subexpressions such as `sin(x)`, `exp(1)`, `x^(1/3)` etc., but

not constant algebraic subexpressions such as `I` and `(sqrt(2)+1)^3`, are replaced by auxiliary variables before factoring. Algebraic dependencies of the subexpressions, such as the equation $cos(x)^2 = 1 - sin(x)^2$, are not necessarily taken into account. See "Example 7" on page 1-738.

The resulting expression is then written as a quotient of two polynomial expressions in the original and the auxiliary indeterminates. The numerator and the denominator are converted into polynomials with coefficient ring `Expr` via `poly`, and the implied coefficient ring is the smallest ring containing the coefficients of the numerator polynomial and the denominator polynomial. Usually, this is the ring of integers. Then both polynomials are factored over the implied coefficient ring, and the multiplicities e_i corresponding to factors of the denominator are negative integers; see "Example 3" on page 1-736. After the factorization, the auxiliary variables are replaced by the original subexpressions. See "Example 6" on page 1-738.

If `f` is an integer, then it is decomposed into a product of primes, and the result is the same as for `ifactor`. If `f` is a rational number, then both the numerator and the denominator are decomposed into a product of primes. In this case, the multiplicities e_i corresponding to factors of the denominator are negative integers. See "Example 2" on page 1-735.

If `f` is a floating point number or a complex number, then `factor` returns a factorization with the single factor `f`.

The result of `factor` is an object of the domain type `Factored`. Let `g:=factor(f)` be such an object.

It is represented internally by the list `[u, f1, e1, ..., fr, er]` of odd length $2r + 1$. Here, `f1` through `fr` are of the same type as the input (either polynomials or expressions); `e1` through `er` are integers; and `u` is an arithmetical expression.

One may extract the content u and the terms $f_i^{e_i}$ by the ordinary index operator `[ ]`, i.e., `g[1] = f1^e1, g[2] = e1^e2, ...` if $u = 1$ and `g[1] = u, g[2] = f1^e1, g[3] = e1^e2, ...`, respectively, if $u \neq 1$.

The call `Factored::factors(g)` yields the list `[f1, f2, ...]` of factors, the call `Factored::exponents(g)` returns the list `[e1, e2, ...]` of exponents.

The call `coerce(g,DOM_LIST)` returns the internal representation of a factored object, i.e., the list `[u, f1, e1, f2, e2, ...]`.

Note that the result of `factor` is printed as an expression, and it is implicitly converted into an expression whenever it is processed further by other MuPAD functions. As an example, the result of `q:=factor(x^2+2*x+1)` is printed as `(x+1)^2`, which is an expression of type `"_power"`.

See “Example 1” on page 1-733 for illustrations, and the help page of `Factored` for details.

If `f` is not a number, then each of the polynomials $p_1, \ldots, p_r$ is primitive, i.e., the greatest common divisor of its coefficients (see `content` and `gcd`) over the implied coefficient ring (see above for a definition) is one.

Currently, factoring polynomials is possible over the following implied coefficient rings: integers, real numbers, complex numbers and rational numbers, finite fields—represented by `IntMod(n)` or `Dom::IntegerMod(n)` for a prime number `n`, or by a `Dom::GaloisField`—, and rings obtained from these basic rings by taking polynomial rings (see `Dom::DistributedPolynomial`, `Dom::MultivariatePolynomial`, `Dom::Polynomial`, and `Dom::UnivariatePolynomial`), fields of fractions (see `Dom::Fraction`), and algebraic extensions (see `Dom::AlgebraicExtension`).

If the input `f` is an arithmetical expression that is not a number, all occurring floating-point numbers are replaced by continued fraction approximations. The result is sensitive to the environment variable `DIGITS`, see `numeric::rationalize` for details.

Examples

Example 1

To factor the polynomial $x^3 + x$, enter:

```
g := factor(x^3+x)
```

$$x\,(x^2+1)$$

Usually, expressions are factored over the ring of integers, and factors with non-integral coefficients, such as `x - I` in the example above, are not considered.

One can access the internal representation of this factorization with the ordinary index operator:

```
g[1], g[2]
```

$$x,\ x^2+1$$

The internal representation of g, as described above, is given by the following command:

```
coerce(g, DOM_LIST)
```

$$[1,\ x,\ 1,\ x^2+1,\ 1]$$

The result of the factorization is an object of domain type `Factored`:

```
domtype(g)
```

Factored

Some of the functionality of this domain is described in what follows.

One may extract the factors and exponents of the factorization also in the following way:

```
Factored::factors(g), Factored::exponents(g)
```

$$[x,\ x^2+1],\ [1,\ 1]$$

One can ask for the type of factorization:

```
Factored::getType(g)
```

"irreducible"

This output means that all f_i are irreducible. Other possible types are `"squarefree"` (see `polylib::sqrfree`) or `"unknown"`.

One may multiply factored objects, which preserves the factored form:

```
g2 := factor(x^2 + 2*x + 1)
```

$$(x+1)^2$$

```
g * g2
```

$$x\,(x^2+1)\,(x+1)^2$$

It is important to note that one can apply (almost) any function working with arithmetical expressions to an object of type `Factored`. However, the result is then usually not of domain type `Factored`:

```
expand(g);
domtype(%)
```

$$x^3+x$$

DOM_EXPR

For a detailed description of these objects, please refer to the help page of the domain `Factored`.

Example 2

`factor` splits an integer into a product of prime factors:

```
factor(8)
```

$$2^3$$

For rational numbers, both the numerator and the denominator are factored:

```
factor(10/33)
```

$$2\,5\,3^{-1}\,11^{-1}$$

Note that, in contrast, constant polynomials are *not* factored:

```
factor(poly(8, [x]))
```

$$8$$

Example 3

Factors of the denominator are indicated by negative multiplicities:

```
factor((z^2 - 1)/z^2)
```

$$\frac{(z-1)\,(z+1)}{z^2}$$

```
Factored::factors(%), Factored::exponents(%)
```

$$[z-1,\ z+1,\ z],\ [1,\ 1,\ -2]$$

Example 4

If some coefficients are irrational but algebraic, the factorization takes place over the smallest field extension of the rationals that contains all of them. Hence, `x^2+1` is considered irreducible while its `I`-fold is considered reducible:

```
factor(x^2 + 1), factor(I*x^2 + I)
```

$$x^2+1,\ \mathrm{i}\,(x-\mathrm{i})\,(x+\mathrm{i})$$

MuPAD does not automatically factor over the field of algebraic numbers; only the coefficients of the input are adjoined to the rationals:

```
factor(sqrt(2)*x^4 - sqrt(2)*x^2 - sqrt(2)*2)
```

$$\sqrt{2}\,(x+\sqrt{2})\,(x-\sqrt{2})\,(x^2+1)$$

```
factor(I*x^4 - I*x^2 - I*2)
```

$$i\,(x-i)\,(x+i)\,(x^2-2)$$

```
factor(sqrt(2)*I*x^4 - sqrt(2)*I*x^2 - sqrt(2)*I*2)
```

$$(\sqrt{2}\,i)\,(x+\sqrt{2})\,(x+i)\,(x-i)\,(x-\sqrt{2})$$

Example 5

With the option *Adjoin*, additional elements can be adjoined to the implied coefficient ring:

```
factor(x^2 + 1, Adjoin = [I])
```

$$(x-i)\,(x+i)$$

```
factor( x^2-2, Adjoin = {sqrt(2)} )
```

$$(x-\sqrt{2})\,(x+\sqrt{2})$$

With the option `Full`, a complete factorization into linear factors can be computed.

```
factor( x^2-2, Full)
```

$$(x-\sqrt{2})\,(x+\sqrt{2})$$

If the argument R_ or C_ is given, factorization is done over the real or complex numbers using numeric calculations:

```
factor( x^2-2, R_ )
```

$$(x+1.414213562)\,(x-1.414213562)$$

```
factor(x^2 + 1, C_)
```

$$(x-1.0\,i)\,(x+1.0\,i)$$

Example 6

Transcendental objects are treated as indeterminates:

```
delete x:
factor(7*(cos(x)^2 - 1)*sin(1)^3)
```

$$7\sin(1)^3(\cos(x)-1)(\cos(x)+1)$$

```
Factored::factors(%), Factored::exponents(%)
```

$$[\sin(1), \cos(x)-1, \cos(x)+1], [3, 1, 1]$$

Example 7

`factor` regards transcendental subexpressions as algebraically independent of each other. Sometimes, the dependence is recognized:

```
factor(x + 2*sqrt(x) + 1)
```

$$(\sqrt{x}+1)^2$$

In many cases, however, the algebraic dependence is not recognized:

```
factor(x^2 + (2^y*3^y + 6^y)* x + (6^y)^2)
```

$$6^{2\,y} + 6^y x + x^2 + 2^y 3^y x$$

Example 8

`factor` replaces floating-point numbers by continued fraction approximations, factors the resulting polynomial, and finally applies `float` to the coefficients of the factors:

```
factor(x^2 + 2.0*x - 8.0)
```

$$1.0(x+4.0)(x-2.0)$$

Example 9

`factor` with the option `Full` can use `RootOf` to symbolically represent the roots of a polynomial:

```
factor(x^5 + x^2 + 1, Full)
```

$$\left(x-(\sigma_1)_1\right)\left(x-(\sigma_1)_2\right)\left(x-(\sigma_1)_3\right)\left(x-(\sigma_1)_4\right)\left(x-(\sigma_1)_5\right)$$

where

$$\sigma_1 = \mathrm{RootOf}(z^5+z^2+1,\ z)$$

Example 10

Polynomials with a coefficient ring other than `Expr` are factored over their coefficient ring. We factor the following polynomial modulo 17:

```
R := Dom::IntegerMod(17): f:= poly(x^3 + x + 1, R):
factor(f)
```

$$\mathrm{poly}(x+6,\ [x],\ \mathrm{Dom::IntegerMod}(17))\ \mathrm{poly}(x^2+11\,x+3,\ [x],\ \mathrm{Dom::IntegerMod}(17))$$

For every `p`, the expression `IntMod(p)` may be used instead of `Dom::IntegerMod(p)`:

```
R := IntMod(17): f:= poly(x^3 + x + 1, R):
factor(f)
```

$$\mathrm{poly}(x+6,\ [x],\ \mathrm{IntMod}(17))\ \mathrm{poly}(x^2-6\,x+3,\ [x],\ \mathrm{IntMod}(17))$$

Example 11

More complex domains are allowed as coefficient rings, provided they can be obtained from the rational numbers or from a finite field by iterated construction of algebraic extensions, polynomial rings, and fields of fractions. In the following example, we factor the univariate polynomial u^2 - x^3 in u over the coefficient field $F = \mathbb{Q}(x, \sqrt{x})$:

```
Q := Dom::Rational:
Qx := Dom::Fraction(Dom::DistributedPolynomial([x], Q)):
F := Dom::AlgebraicExtension(Qx, poly(z^2 - x, [z])):
f := poly(u^2 - x^3, [u], F)
```

$$\text{poly}(u^2 - x^3, [u], \text{Dom::AlgebraicExtension}(\text{Dom::Fraction}(\text{Dom::DistributedPolynomial}([x], \text{Dom::Rational}, \text{LexOrder})), -x + z^2 = 0, z))$$

```
factor(f)
```

$$\text{poly}(u + x\,z, [u], \text{Dom::AlgebraicExtension}(\text{Dom::Fraction}(\text{Dom::DistributedPolynomial}([x], \text{Dom::Rational}, \text{LexOrder})), -x + z^2 = 0, z))\ \text{poly}(u - x\,z, [u], \text{Dom::AlgebraicExtension}(\text{Dom::Fraction}(\text{Dom::DistributedPolynomial}([x], \text{Dom::Rational}, \text{LexOrder})), -x + z^2 = 0, z))$$

Parameters

f

A polynomial or an arithmetical expression

F

R_ or C_

Options

MaxDegree

Option, specified as `MaxDegree = n`

Only algebraic numbers of a maximum degree `n` will be adjoined to the rational numbers. If not specified, all coefficients up to degree 2 are adjoined. `n` must be a positive integer.

`Adjoin`

Option, specified as `Adjoin = adjoin`

In addition to the coefficients of `f`, the elements of `adjoin` are adjoined to the rational numbers. Elements of algebraic degree larger than the value of the option `MaxDegree` are not adjoined. `adjoin` must be a set or list.

`Domain`

Option, specified as `Domain = F`

Compute a numerical factorization over $\mathbb{R}$ or $\mathbb{C}$, respectively.

`Full`

Compute the full factorization of `f` into linear factors. This option has no effect on multivariate polynomials.

Return Values

Object of the domain type `Factored`.

Overloaded By

f

Algorithms

The factoring algorithms are collected in a separate library domain `faclib`; it should not be necessary to call these routines directly.

The implemented algorithms include Cantor-Zassenhaus (over finite fields) and Hensel lifting (over the rational numbers and in the multivariate case).

See Also

MuPAD Functions

`collect` | `content` | `denom` | `div` | `divide` | `expand` | `Factored` | `gcd` | `icontent` | `ifactor` | `igcd` | `ilcm` | `indets` | `irreducible` | `isprime` | `lcm` | `normal` | `numer` | `partfrac` | `polylib::decompose` | `polylib::divisors` | `polylib::primpart` | `polylib::sqrfree` | `rationalize` | `simplify`

factorout

Factor out a given expression

Syntax

```
factorout(x, f, <list>)
```

Description

`factorout(x,f)` factors out a given expression `f` from the expression `x`.

The result is a product of the form $f\ \left(\frac{x}{f}\right)$.

If the optional parameter *list* is set to `TRUE`, a list of the factors is returned. See "Example 2" on page 1-744

Examples

Example 1

```
factorout(2*x+4, 2)
```

$$2\,(x+2)$$

```
factorout(a+a*2, a)
```

$$3\,a$$

```
factorout(a+a*3, 2)
```

$$2\,(2\,a)$$

```
factorout(a*b + b*c, b)
```

$$b\,(a+c)$$

```
factorout(a*sin(b) + c*sin(b), sin(b))
```

$$\sin(b)\,(a+c)$$

```
factorout(sqrt(50)*x^2+5*x-sqrt(10)*x-sqrt(10), sqrt(5))
```

$$\sqrt{5}\,(\sqrt{10}\,x^2+(\sqrt{5}-\sqrt{2})\,x-\sqrt{2})$$

```
factorout((a*b + b*c)/(d*c+c), b/c)
```

$$\frac{b}{c}\,\frac{a+c}{d+1}$$

Example 2

With the optional parameter 'list' set to true, a list of all factors is returned:

```
factorout(a*b + b*c, b, TRUE)
```

$$[b,\ a+c]$$

Parameters

x

An expression.

f

The expression to be factored out.

list

A boolean value. If list is TRUE, then a list is returned. By default, an expression is returned.

Return Values

Expression or a list.

FAIL

Indicate a failed computation

Syntax

```
FAIL
```

Description

FAIL is a keyword of the MuPAD language. Many functions of the library use the return value FAIL to indicate failed computations or non-existing elements.

FAIL is the only element of the domain DOM_FAIL.

FAIL is used as the return value for computations that failed. Also, requesting non-existing slots of domains or function environments yields FAIL. Due to this behavior, library functions can try computations without provoking errors.

A function should return FAIL or an error if at least one of its inputs is FAIL.

Examples

Example 1

The following attempt to convert sqrt(3) to an integer of a residue class ring must fail:

```
poly(sqrt(3)*x, [x], Dom::IntegerMod(3))
```

```
FAIL
```

The following matrix is not invertible. You can try to invert it without producing an error:

```
A := matrix([[1, 1], [1, 1]]): 1/A
```

```
FAIL
```

The `"inverse"` slot of a function environment yields the inverse of the function. The inverse of the sine function is implemented, but MuPAD does not know the inverse of the dilogarithm function:

```
sin::inverse, dilog::inverse
```

```
"arcsin", FAIL
```

```
delete A:
```

Example 2

Most functions return `FAIL` or an error on input of `FAIL`:

```
poly(FAIL)
```

```
FAIL
```

```
sin(FAIL)
```

```
Error: An arithmetical expression is expected. [sin]
```

Example 3

`FAIL` evaluates to itself:

```
FAIL, eval(FAIL), level(FAIL, 5)
```

```
FAIL, FAIL, FAIL
```

See Also

MuPAD Functions

`error` | `NIL` | `null`

fclose

Close a file

Syntax

```
fclose(n)
```

Description

`fclose(n)` closes the file specified by the file descriptor `n`.

The file must have been opened with `fopen`. The call to `fopen` yields the file descriptor `n` representing the file.

Only a limited number of file descriptors is available. The user should use `fclose` to close a file which is no longer needed because this releases the file descriptor. The exact number of file descriptors available depends on the used operating system.

Examples

Example 1

We open a file `test` for writing. This yields the file descriptor `n`:

```
fid := fopen(TempFile, Write, Text):
file := fname(fid):
n := fopen(file, Write):
```

We close the file:

```
fclose(n):
delete n:
```

Parameters

n

A file descriptor returned by `fopen`: a positive integer

Return Values

Void object of type `DOM_NULL`.

See Also

MuPAD Functions

`FILEPATH` | `finput` | `fname` | `fopen` | `fprint` | `fread` | `ftextinput` | `import::readbitmap` | `import::readdata` | `pathname` | `print` | `protocol` | `read` | `readbytes` | `READPATH` | `write` | `writebytes` | `WRITEPATH`

FILEPATH

Pathname of a file that is currently loaded

Description

FILEPATH is a variable containing the path of a currently read file.

Possible values: String

The variable FILEPATH represents the pathname of a file. It only has a value while reading a file via read or fread and corresponds to the path specified in read or fread. It can only be accessed from inside the file that is currently read. Using this variable, the read file can access its own pathname and read other files via absolute pathnames, even if it only knows their relative locations with respect to itself.

The value of FILEPATH is a string containing the operating system dependent path to the file that is currently read. The path string terminates with a path separator and, under Windows®, starts with the name of the current volume if this was specified in the read/fread command. Cf. “Example 1” on page 1-750.

Examples

Example 1

Assume that the file C:\TEMP\file.mu contains the following lines of code. It queries its own location via FILEPATH (= C:\TEMP) and reads two files installed relative to the location of file.mu via their absolute pathnames C:\TEMP\SubFolder\file1.mu and C:\TEMP\SubFolder\file2.mu, respectively:

```
print(Unquoted, "FILEPATH" = FILEPATH):
read(FILEPATH.pathname("SubFolder")."file1.mu"):
read(FILEPATH.pathname("SubFolder")."file2.mu"):
```

When reading the file file.mu, the part C:\TEMP\ of the specified path is accessed by file.mu via FILEPATH. It finds the files file1.mu and file2.mu if they were installed correctly relative to the path of file.mu:

```
read("C:".pathname(Root, "TEMP"), "file.mu")
```

```
 FILEPATH = C:\TEMP\
```

It is good programming style to use platform independent path strings. For this reason, we used the function `pathname` rather than a mere string concatenation to append appropriate path delimiters.

See Also

MuPAD Functions

`fclose` | `fopen` | `fread` | `pathname` | `read` | `READPATH`

finput

Read objects from file

Syntax

```
finput(filename | n, <Encoding = "encodingValue">)
```

```
finput(filename | n, <Encoding = "encodingValue">, x1, x2, …)
```

Description

`finput(filename, x)` reads a MuPAD object from a file and assigns it to the identifier `x`.

`finput(n, x)` reads from the file associated with the file descriptor `n`.

`finput` can read MuPAD binary files as well as ASCII text files. `finput` recognizes the format of the file automatically.

`finput(..., Encoding = "encodingValue", ...)` uses the specified encoding. For supported encodings, see “Options” on page 1-757. You can use this option with any of the previously specified syntaxes.

Binary files may be created via `fprint` or `write`. Text files can also be created in a MuPAD session via these functions (using the `Text` option; see the corresponding help pages for details). Alternatively, text files can be created and edited directly using your favorite text editor. The file must consist of syntactically correct MuPAD objects or statements, separated by semicolons or colons. An object may extend over more than one line.

`finput(filename)` reads the first object in the file and returns it to the MuPAD session.

`finput(filename, x1, x2, ...)` reads the contents of a file object by object. The i-th object is assigned to the identifier x_i. The identifiers are not evaluated while executing `finput`; previously assigned values are overwritten. The objects are not evaluated. Evaluation can be enforced with the function `eval`. Cf. “Example 2” on page 1-754.

Instead of a file name, also a file descriptor `n` of a file opened via `fopen` can be used. The functionality is as described above. However, there is one difference: With a file name, the file is closed automatically after the data were read. A subsequent call to `finput` starts at the beginning of the file. With a file descriptor, the file remains open (use `fclose` to close the file). The next time data are read from this file, the reading continues at the current position. Consequently, a file descriptor should be used if the individual objects in the file are to be read via several subsequent calls of `finput`. Cf. “Example 3” on page 1-755.

Files in `gzip` compressed format with a filename ending in “`.gz`” are automatically and transparently decompressed while reading.

If the number of identifiers specified in the `finput` call is larger than the number of objects in the file, the additional identifiers are assigned the value `null()`.

`finput` interprets the file name as a pathname relative to the “working folder.”

Note that the meaning of “working folder” depends on the operating system. On Windows systems and on Mac OS X systems, the “working folder” is the folder where MATLAB® is installed. On UNIX systems, it is the current working folder in which MATLAB was started. When started from a menu or desktop item, this is typically the user's home folder.

Also absolute path names are processed by `finput`.

Expression sequences are not flattened by `finput` and cannot be used to pass several identifiers to `finput`. Cf. “Example 4” on page 1-756.

Examples

Example 1

Create a new file in the system's temporary folder. The name of the temporary folder varies for different platforms. The `fopen` command with the `TempFile` option creates a file in any system's temporary folder (if such folder exists):

```
fid := fopen(TempFile, Write, Text):
```

Write the numbers 11, 22, 33 and 44 into a file:

```
fprint(fid, 11, 22, 33, 44):
```

Use `fname` to return the name of the temporary file you created:

```
file := fname(fid):
```

Read this file with `finput`:

```
finput(file, x1, x2, x3, x4)
```

```
    44
```

```
x1, x2, x3, x4
```

```
    11, 22, 33, 44
```

If you try to read more objects than stored in the file, `finput` returns the void object of type `DOM_NULL`:

```
finput(file, x1, x2, x3, x4, x5); domtype(%)
```

```
    DOM_NULL
```

```
delete x1, x2, x3, x4, x5:
```

Example 2

Objects read from a file are not evaluated:

```
fid := fopen(TempFile, Write, Text):
file := fname(fid):
fprint(file, x1):
x1 := 23:
finput(file)
```

```
    x1
```

```
eval(%)
```

```
    23
```

```
delete x1:
```

Example 3

Read some data from a file using several calls of `finput`. You have to use a file descriptor for reading from the file. The file is opened for reading with `fopen`:

```
fid := fopen(TempFile, Write, Text):
fprint(fid, 11, 22, 33, 44):
file := fname(fid):
n := fopen(file):
```

The file descriptor returned by `fopen` can be passed to `finput` for reading the data:

```
finput(n, x1, x2): x1, x2
```

```
     11, 22
```

```
finput(n, x3, x4):
x3, x4
```

```
     33, 44
```

Close the file and delete the identifiers:

```
fclose(n):
delete n, x1, x2, x3, x4:
```

Alternatively, the contents of a file can be read into a MuPAD session in the following way:

```
n := fopen(file):
for i from 1 to 4 do
   x.i := finput(n)
end_for:
x1, x2, x3, x4
```

```
     11, 22, 33, 44
```

```
fclose(n):
delete n, i, x1, x2, x3, x4:
```

Example 4

Expression sequences are not flattened by `finput` and cannot be used to pass identifiers to `finput`:

```
fid := fopen(TempFile, Write, Text):
fprint(fid, 11, 22, 33):
file := fname(fid):
finput(file, (x1, x2), x3)
```

```
Error: The argument is invalid. [finput]
```

The following call does not lead to an error because the identifier `x12` is not evaluated. Consequently, only one object is read from the file and assigned to `x12:`

```
x12 := x1, x2:
finput(file, x12):
x1, x2, x12
```

```
     x1, x2, 11
```

```
delete x1, x2, x12:
```

Example 5

To specify the encoding to write data, use Encoding. The Encoding option applies only to text files that are opened using a file name and not a file descriptor. Create a temporary file and store the values `"abcäöü"`, `11` and `22` in the encoding “`UTF-8`”:

```
fprint(Text, Encoding="UTF-8", "finput_test", "abcäöü", 11, 22):
```

Specify the encoding to read the stored values correctly:

```
finput("finput_test", Encoding="UTF-8", x1, x2, x3):
x1, x2, x3
```

```
     "abcäöü", 11, 22
```

If you do not specify an encoding, the default system encoding is used. Thus, your output might vary from that shown next. Characters unrecognized by the default system encoding are replaced by the default substitution character for that encoding:

```
finput("finput_test", x1, x2, x3):
x1, x2, x3

     "abc??????", 11, 22
```

Parameters

`filename`

The name of a file: a character string

`n`

A file descriptor provided by `fopen`: a positive integer

x_1, x_2, …

identifiers

Options

`Encoding`

This option lets you specify the character encoding to use. The allowed encodings are:

`"Big5"`	`"ISO-8859-1"`	`"windows-932"`
`"EUC-JP"`	`"ISO-8859-2"`	`"windows-936"`
`"GBK"`	`"ISO-8859-3"`	`"windows-949"`
`"KSC_5601"`	`"ISO-8859-4"`	`"windows-950"`
`"Macintosh"`	`"ISO-8859-9"`	`"windows-1250"`
`"Shift_JIS"`	`"ISO-8859-13"`	`"windows-1251"`
`"US-ASCII"`	`"ISO-8859-15"`	`"windows-1252"`
`"UTF-8"`		`"windows-1253"`
		`"windows-1254"`
		`"windows-1257"`

The default encoding is system dependent. If you specify the encoding incorrectly, characters might read incorrectly. Characters unrecognized by the encoding are replaced by the default substitution character for the specified encoding.

Encodings not listed here can be specified but might not produce correct results.

Return Values

Last object that was read from the file.

See Also

MuPAD Functions

`fclose` | `fname` | `fopen` | `fprint` | `fread` | `ftextinput` | `input` | `pathname` | `print` | `protocol` | `read` | `READPATH` | `textinput` | `write` | `WRITEPATH`

float

Convert to a floating-point number

Syntax

```
float(object)
```

```
float(object, n)
```

Description

`float(object)` converts the object or numerical subexpressions of the object to floating-point numbers.

`float` converts numbers and numerical expressions such as `sqrt(sin(2))` or `sqrt(3) + sin(PI/17)*I` to real or complex floating-point numbers of type `DOM_FLOAT` or `DOM_COMPLEX`, respectively. If symbolic objects other than the special constants CATALAN, E, EULER, and PI are present, only *numerical* subexpressions are converted to floats. In particular, identifiers and indexed identifiers are returned unchanged by `float`. Cf. “Example 1” on page 1-761.

A `float` call is mapped recursively to the operands of an expression. When numbers (or constants such as PI) are found, they are converted to floating-point approximations. The number of significant decimal digits is given by the environment variable `DIGITS`; the default value is 10. The converted operands are combined by arithmetical operations or function calls according to the structure of the expression. E.g., a call such as `float(PI - 314/100)` may be regarded as a sequence of numerical operations:

```
t1 := float(PI); t2 := float(314/100); result := t1 - t2
```

Consequently, float evaluation via `float` may be subject to error propagation. Cf. “Example 2” on page 1-761.

The second argument `n` in `float(object, n)` temporarily overwrites the current setting for `DIGITS`. See “Example 3” on page 1-762.

`float` is automatically mapped to the elements of sets and lists. However, it is not automatically mapped to the entries of arrays, hfarrays, tables, and operands of function

calls. Use `map(object, float)` for a fast floating-point conversion of all entries of an array or a table. Use `mapcoeffs(p, float)` to convert the coefficients of a polynomial `p` of type `DOM_POLY`. To control the behavior of `float` on a function call, use a function environment providing a `"float"` slot. Cf. “Example 4” on page 1-763 and “Example 5” on page 1-764.

The preferences `Pref::floatFormat` and `Pref::trailingZeroes` can be used to modify the screen output of floating-point numbers.

Rational approximations of floating-point numbers may be computed by the function `numeric::rationalize`.

MuPAD special functions such as `sin`, `exp`, `besselJ` etc. are implemented as function environments. Via overloading, the `"float"` attribute (slot) of a function environment `f`, say, is called for the float evaluation of symbolic calls `f(x1, x2, ...)` contained in an expression.

The user may extend the functionality of the system function `float` to his own functions. For this, the function `f` to be processed must be declared as a function environment via `funcenv`. A `"float"` attribute must be written, which is called by the system function `float` in the form `f::float(x1, x2, ...)` whenever a symbolic call `f(x1, x2, ...)` inside an expression is found. The arguments passed to `f::float` are not converted to floats, neither is the return value of the slot subject to any further float evaluation. Thus, the float conversion of symbolic functions calls of `f` is entirely determined by the slot routine. Cf. “Example 5” on page 1-764.

Also a domain `d`, say, written in the MuPAD language, can overload `float` to define the float evaluation of its elements. A slot `d::float` must be implemented. If an element `x`, say, of this domain is subject to a float evaluation, the slot is called in the form `d::float(x)`. As for function environments, neither `x` nor the return value of the slot are subject to any further float evaluation.

If a domain does not have a `"float"` slot, the system function `float` returns its elements unchanged.

Note that MuPAD floating-point numbers are restricted in size. On 32 bit architectures, an overflow/underflow occurs if numbers of absolute size larger/smaller than about $10.0^{\pm 2525222}$ are encountered. On 64 bit architectures, the limits are about $10.0^{\pm 42366205509363}$.

See the documentation for `DIGITS` for further information.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We convert some numbers and numerical expressions to floats:

```
float(17), float(PI/7 + I/4), float(4^(1/3) + sin(7))
```

$$17.0,\ 0.4487989505 + 0.25\,i,\ 2.244387651$$

`float` is sensitive to `DIGITS`:

```
DIGITS := 20:
float(17), float(PI/7 + I/4), float(4^(1/3) + sin(7))
```

$$17.0,\ 0.44879895051282760549 + 0.25\,i,\ 2.2443876506869885651$$

Symbolic objects such as identifiers are returned unchanged:

```
DIGITS := 10: float(2*x + sin(3))
```

$$2.0\,x + 0.1411200081$$

Example 2

We illustrate error propagation in numerical computations. The following rational number approximates `exp(2)` to 17 decimal digits:

```
r := 738905609893065023/100000000000000000:
```

The following `float` call converts `exp(2)` and `r` to floating-point approximations. The approximation errors propagate and are amplified in the following numerical expression:

```
DIGITS := 10: float(10^20*(r - exp(2)))
```

```
0.0
```

None of the digits in this result is correct! To obtain a better result, use the second argument in `float` to increase the number of digits for this particular function call:

```
float(10^20*(r - exp(2)), 20)
```

```
276.9572539
```

For further calculations, free the variable `r`:

```
delete r:
```

Example 3

The second argument in `float` lets you temporarily overwrite the current setting for the number of significant decimal digits. For example, compute the following expression with 10 and 30 significant decimal digits. To display floating-point numbers with the number of digits that MuPAD used to compute them, set the value of `Pref::outputDigits` to `InternalPrecision`:

```
Pref::outputDigits(InternalPrecision):
```

Compute the following expression with the default value of `DIGITS = 10`:

```
x := 10^8:
float(sqrt(x^2 + 1) - x)
```

```
0.000000004889443517
```

Compute the same expression with 30 significant decimal digits:

```
float(sqrt(x^2 + 1) - x, 30)
```

```
0.000000004999999999999999875000638189 77
```

After evaluating `float`, MuPAD restores the value of `DIGITS`:

```
DIGITS
```

```
      10
```

For further calculations, restore the output precision and free the variable `x`:

```
Pref::outputDigits(UseDigits):
delete x
```

Example 4

`float` is mapped to the elements of sets and lists:

```
float([PI, 1/7, [1/4, 2], {sin(1), 7/2}])
```

```
      [3.141592654, 0.1428571429, [0.25, 2.0], {0.8414709848, 3.5}]
```

For tables and arrays, the function `map` must be used to forward `float` to the entries:

```
T := table("a" = 4/3, 3 = PI):
float(T), map(T, float)
```

```
      3  | π,          3  | 3.141592654
      "a"| 4/3         "a"| 1.333333333
```

```
A := array(1..2, [1/7, PI]):
float(A), map(A, float)
```

```
      ( 1/7 π ), ( 0.1428571429  3.141592654 )
```

Matrix domains overload the function `float`. In contrast to arrays, `float` works directly on a matrix:

```
float(matrix(A))
```

```
      ( 0.1428571429 )
      ( 3.141592654  )
```

Use `mapcoeffs` to apply `float` to the coefficients of a polynomial generated by `poly`:

```
p := poly(9/4*x^2 + PI, [x]): float(p), mapcoeffs(p, float)
```

$$\mathrm{poly}\left(\frac{9\,x^2}{4}+\pi,\,[x]\right),\ \mathrm{poly}(2.25\,x^2+3.141592654,\,[x])$$

```
delete A, T, p:
```

Example 5

We demonstrate overloading of `float` by a function environment. The following function `Sin` is to represent the sine function. In contrast to the `sin` function in MuPAD, `Sin` measures its argument in degrees rather than in radians (i.e., `Sin(x) = sin(PI/180*x)`). The only functionality of `Sin` is to produce floating point values if the argument is a real float. For all other kinds of arguments, a symbolic function call is to be returned:

```
Sin := proc(x)
begin
  if domtype(x) = DOM_FLOAT then
    return(Sin::float(x));
  else return(procname(args()))
  end_if;
end_proc:
```

The function is turned into a function environment via `funcenv`:

```
Sin := funcenv(Sin):
```

Finally, the `"float"` attribute is implemented. If the argument can be converted to a real floating-point number, a floating-point result is produced. In all other cases, a symbolic call of `Sin` is returned:

```
Sin::float := proc(x)
begin
  x := float(x):
  if domtype(x) = DOM_FLOAT then
    return(float(sin(PI/180*x)));
  else return(Sin(x))
  end_if;
end_proc:
```

Now, float evaluation of arbitrary expressions involving `Sin` is possible:

```
Sin(x), Sin(x + 0.3), Sin(120)
```

$$\mathrm{Sin}(x),\ \mathrm{Sin}(x+0.3),\ \mathrm{Sin}(120)$$

```
Sin(120.0), float(Sin(120)), float(Sin(x + 120))
```

$$0.8660254038,\ 0.8660254038,\ \mathrm{Sin}(x+120.0)$$

```
float(sqrt(2) + Sin(120 + sqrt(3)))
```

$$2.264730594$$

```
delete Sin:
```

Parameters

`object`

Any MuPAD object

`n`

An integer greater than 1

Return Values

Floating point number of type `DOM_FLOAT` or `DOM_COMPLEX`, or the input object with exact numbers replaced by floating-point numbers.

Overloaded By

object

See Also

MuPAD Functions

`DIGITS` | `isolate` | `Pref::floatFormat` | `Pref::outputDigits` | `Pref::trailingZeroes`

fname

Get a file's name

Syntax

```
fname(n)
```

Description

`fname(n)` returns the name of the file specified by the file descriptor `n`.

The file must have been opened with `fopen`. The call to `fopen` yields the file descriptor `n` representing the file.

The special file descriptor `0` represents no file but output to the user interface instead; `fname(0)` returns `NIL`.

Examples

Example 1

We open a temporary file for writing. This yields the file descriptor `n`:

```
n := fopen(TempFile);
```

```
    16
```

We get the file's name. Note that the name depends on the operating system:

```
fname(n);
```

```
    "/tmp/mtxM9fPT"
```

Parameters

n

A file descriptor returned by `fopen`: a positive integer

Return Values

the name of the file: a character string of type `DOM_STRING`, or `NIL`.

See Also

MuPAD Functions
`fclose` | `finput` | `fopen` | `fprint` | `fread` | `ftextinput`

fopen

Open file

Syntax

```
fopen(filename | TempFile, <Read | Write | Append>, <Bin | Text | Raw>, <Encoding = "er
```

Description

`fopen(filename, format)` opens an existing file for reading in the specified format. An error is raised if no file with the specified name is found or the format of the file does not coincide with the specified `format`. If the file is in `gzip`-compressed format and its name ends in "`.gz`", it will be transparently uncompressed upon reading.

`fopen(filename)` opens an existing file for reading. The file must hold data in text or MuPAD binary format (optionally compressed), `fopen` automatically identifies the file format in this case. The file must not be used as raw file.

`fopen(filename, mode, format)` opens the file for writing in the specified format if the mode is given as `Read` or `Append`. If no file with the specified name exists, a new file is created. If the filename ends in "`.gz`", all data written to the file will be transparently compressed in `gzip` compatible format.

`fopen(TempFile, format)` creates and opens a temporary file for writing in the specified format. The option `Read` and `Append` are not allowed in this case. If no format is given, `Bin` is used. Use `fname` to query the actual name and location of the temporary file. Cf. "Example 3" on page 1-772.

`fopen(..., Encoding = "encodingValue")` uses the specified encoding. For supported encodings, see "Options" on page 1-773. You can use this option with the previously specified syntaxes for text files.

In write mode (using one of the options `Write` or `Append`), the environment variable `WRITEPATH` is considered if no temporary file is created. If it has a value, a new file is created (or an existing file is searched for) in the corresponding folder. Otherwise, it is created/searched for in the "working folder."

Note that the meaning of “working folder” depends on the operating system. On Windows systems and on Mac OS X systems, the “working folder” is the folder where MATLAB is installed. On UNIX systems, it is the current working folder in which MATLAB was started. When started from a menu or desktop item, this is typically the user's home folder.

Note: In read mode, `fopen` does not search for files in the folders given by `READPATH` and the library path.

A temporary file is created in a special folder. This folder and the name of the file are system dependent.

Also absolute path names are processed by `fopen`.

The file descriptor returned by `fopen` can be used by various functions such as `fname`, `fclose`, `fread`, `fprint`, `read`, `write` etc.

A file opened by `fopen` should be closed by `fclose` after use. This holds also for temporary files.

`fopen` accepts its arguments in any order, not only in the order used above.

Environment Interactions

The function is sensitive to the environment variable `WRITEPATH` when creating files that are not temporary (temporary files are created via `TempFile`). If `WRITEPATH` has a value, in write mode (using the options `Write` or `Append`), the file is created in the corresponding folder. Otherwise, the file is created in the “working folder.” A temporary file is created in a special folder.

When using `Write` or `Append`, `fopen` creates a new file if no file under the given name exists.

Examples

Example 1

Open the file `test` for writing. With the option `Write`, it is not necessary that the file `test` exists. By default, the file is opened as a binary file:

```
fid := fopen("test", Write):
```

Write a string to the file and close it:

```
fprint(fid, "a string"):
fclose(fid):
```

Append another string to the file:

```
fid := fopen("test", Append):
fprint(fid, "another string"):
fclose(fid):
```

The binary file cannot be opened as a text file for appending data:

```
fid := fopen("test", Append, Text)
```

```
    FAIL
```

However, it may be opened as a text file with the option `Write`. The existing binary file is overwritten with a text file:

```
fid := fopen("test", Write, Text):
fclose(fid):
delete fid:
```

Example 2

`fopen` fails to open non-existing files for reading. Here, assume the file "`xyz`" does not exist:

```
n := fopen("xyz")
```

```
    FAIL
```

Assume the file "test" created in "Example 1" on page 1-771 exists. It can be opened for reading successfully:

```
n := fopen("test")
```

```
     119
```

```
fclose(n):
delete n:
```

Example 3

Open a temporary file, write 10 binary data bytes into it and close it. fname is used to query the name of the file:

```
fd := fopen(TempFile, Raw):
writebytes(fd, [i $ i=1..10]):
fn := fname(fd):
fclose(fd):
fn
```

```
     "/tmp/mupad.7aYAp4"
```

Re-open the file and read the data:

```
fd := fopen(fn, Read, Raw):
readbytes(fd);
```

```
     [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
fclose(fd):
delete fd, fn:
```

Example 4

To specify the encoding to read and write data, use Encoding. The Encoding option applies only to text files that are opened using a file name and not a file descriptor. Create a temporary file and write the string "abcäöü" in the encoding "UTF-8":

```
fid := fopen(TempFile, Text, Write, Encoding="UTF-8"):
file := fname(fid):
fprint(Unquoted, fid, "abcäöü"):
```

```
fclose(fid):
```

Use `ftextinput` to read the data with the specified encoding:

```
ftextinput(file, Encoding="UTF-8")
```

```
"abcäöü"
```

If you do not specify an encoding, the default system encoding is used. Thus, your output might vary from that shown next. Characters unrecognized by the default system encoding are replaced by the default substitution character for that encoding:

```
fid := fopen(TempFile, Text, Write):
file := fname(fid):
fprint(Unquoted, fid, "abcäöü"):
fclose(fid):
ftextinput(file)
```

```
"abc"
```

Parameters

`filename`

The name of a file: a character string or the flag `TempFile`

Options

`TempFile`

`fopen` creates a temporary file in the systems “temp” folder. The name of this file can be queried using `fname`.

`Append`, `Read`, `Write`

With `Read`, the file is opened for reading; with `Write` or `Append`, it is opened for writing. If a file opened for writing does not yet exist, it is created. With `Write`, existing files are overwritten. With `Append`, new data may be appended to an existing file. Note that in the `Append` mode, the specified format must coincide with the format of the existing file;

otherwise, the file cannot be opened and `fopen` returns `FAIL`. If the flag `TempFile` is given, the default mode is `Write`. Otherwise, the default mode is `Read`.

`Bin, Raw, Text`

With `Bin`, the data is stored in MuPAD internal binary format. With `Text`, the data may be strings or MuPAD objects stored as text. Newlines are handled according to the conventions of the operating system at hand. With `Raw`, the data is interpreted as binary machine numbers. See the functions `readbytes` and `writebytes`.

If the mode is `Read` or `Append`, the default is the format of the data in the existing file. If the mode is `Write`, the default is `Bin`.

`Encoding`

This option lets you specify the character encoding to use. The allowed encodings are:

`"Big5"`	`"ISO-8859-1"`	`"windows-932"`
`"EUC-JP"`	`"ISO-8859-2"`	`"windows-936"`
`"GBK"`	`"ISO-8859-3"`	`"windows-949"`
`"KSC_5601"`	`"ISO-8859-4"`	`"windows-950"`
`"Macintosh"`	`"ISO-8859-9"`	`"windows-1250"`
`"Shift_JIS"`	`"ISO-8859-13"`	`"windows-1251"`
`"US-ASCII"`	`"ISO-8859-15"`	`"windows-1252"`
`"UTF-8"`		`"windows-1253"`
		`"windows-1254"`
		`"windows-1257"`

The default encoding is system dependent. If you specify the encoding incorrectly, characters might read incorrectly. Characters unrecognized by the encoding are replaced by the default substitution character for the specified encoding.

Encodings not listed here can be specified but might not produce correct results.

Return Values

a positive integer: the file descriptor. `FAIL` is returned if the file cannot be opened.

See Also

MuPAD Functions

`fclose` | `FILEPATH` | `finput` | `fname` | `fprint` | `fread` | `ftextinput` | `import::readbitmap` | `import::readdata` | `pathname` | `print` | `protocol` | `read` | `readbytes` | `READPATH` | `write` | `writebytes` | `WRITEPATH`

for, from, to, step, end_for, _for_in, downto, _for_downto

For loop

Syntax

```
for i from start to stop do
  body
end_for

for i from start to stop step stepwidth do
  body
end_for

_for(i, start, stop, stepwidth, body)

for i from start downto stop do
  body
end_for

for i from start downto stop step stepwidth do
  body
end_for

_for_down(i, start, stop, stepwidth, body)

for x in object do
  body
end_for

_for_in(x, object, body)
```

Description

`for - end_for` is a repetition statement providing a loop for automatic iteration over a range of numbers or objects.

When entering an incrementing loop

`for i from start to stop step stepwidth do body end_for`,

the assignment `i := start` is made. The body is executed with this value of `i` (the body may reassign a new value to `i`). After all statements inside the body are executed, the loop returns to the beginning of the body, increments `i := i + stepwidth` and checks the stopping criterion `i > stop`. If `FALSE`, the body is executed again with the new value of `i`. If `TRUE`, the loop is terminated immediately without executing the body again.

The decrementing loop

```
for i from start downto stop step stepwidth do body end_for
```

implements a corresponding behavior. The only difference is that upon return to the beginning of the body, the loop variable is decremented by `i := i - stepwidth` before the stopping criterion `i < stop` is checked.

The loop `for x in object do body end_for` iterates `x` over all operands of the object. This loop is equivalent to

```
for i from 1 to nops(object) do
x := op(object, i);          body        end_for
```

Typically, `object` may be a list, an expression sequence, an array or an hfarray. Note that other container objects such as finite sets or tables do not have a natural internal ordering, i.e., care must be taken, if the loop expects a certain ordering of the iterative steps.

The body of a loop may consist of any number of statements which must be separated either by a colon `:` or a semicolon `;`. The last evaluated result inside the body is printed on the screen as the return value of the loop. Use `print` inside the loop to see intermediate results.

The loop variable `i`, respectively `x`, may have a value before the loop starts. After the loop is terminated, it has the value that was assigned in the last step of the loop. Typically, in an incrementing or decrementing loop with integer values of `start`, `stop`, and `stepwidth`, this is `i` = `stop` plus or minus `stepwidth`.

The arguments `start`, `stop`, `stepwidth`, and `object` are evaluated only once at the beginning of the loop and not after every iteration. E.g., if `object` is changed in a step of the loop, `x` still runs through all operands of the original object.

Loops can be exited prematurely using the `break` statement. Steps of a loop can be skipped using the `next` statement. Cf. “Example 2” on page 1-780.

The keyword `end_for` may be replaced by the keyword `end`. Cf. “Example 3” on page 1-780.

Instead of the the imperative loop statements, the equivalent calls of the functions `_for`, `_for_down`, or `_for_in` may be used. Cf. “Example 4” on page 1-781.

The `$`-operator is often a more elegant notation for `for`-loops.

`_for`, `_for_down` and `_for_in` are functions of the system kernel.

Examples

Example 1

The body of the following loop consists of several statements. The value of the loop variable `i` is overwritten when the loop is entered:

```
i := 20:
for i from 1 to 3 do
  a := i;
  b := i^2;
  print(a, b)
end_for:
```

```
    1, 1

    2, 4

    3, 9
```

The loop variable now has the value that satisfied the stopping criterion `i > 3`:

```
i
```

```
    4
```

The iteration range is not restricted to integers:

```
for i from 2.2 downto 1 step 0.5 do
  print(i)
end_for:
```

2.2

1.7

1.2

The following loop sums up all elements in a list. The return value of the loop is the final sum. It can be assigned to a variable:

```
s := 0: S := for x in [c, 1, d, 2] do s := s + x end_for
```

$c+d+3$

Note that for sets, the internal ordering is not necessarily the same as printed on the screen:

```
S := {c, d, 1}
```

$\{1, c, d\}$

```
for x in S do print(x) end_for:
```

c

d

1

```
delete a, b, i, s, S, x:
```

Example 2

Loops can be exited prematurely using the `break` statement:

```
for i from 1 to 3 do
  print(i);
  if i = 2 then break end_if
end_for:
```

```
    1
```

```
    2
```

With the `next` statement, the execution of commands in a step can be skipped. The evaluation continues at the beginning of the body with the incremented value of the loop variable:

```
a := 0:
for i from 1 to 3 do
  a := a + 1;
  if i = 2 then next end_if;
  print(i, a)
end_for:
```

```
    1, 1
```

```
    3, 3
```

```
delete i, a:
```

Example 3

Loops can be closed with the keyword `end` instead of `end_for`. The parser recognizes the scope of `end` statements automatically.

```
s:= 0:
for i from 1 to 3 do
  for j from 1 to 3 do
    s := i + j;
    if i + j > 4 then
```

```
      break;
    end
  end
end

    5

delete s, i, j:
```

Example 4

This example demonstrates the correspondence between the functional and the imperative form of `for` loops:

```
hold(
  _for(i, start, stop, stepwidth, (statement1; statement2))
)


for i from start to stop step stepwidth do
  statement1;
  statement2
end_for
```

The optional `step` clause is omitted by specifying the value `NIL` for the step width:

```
hold(
  _for_down(i, 10, 1, NIL, (x := i^2; x := x - 1))
)


for i from 10 downto 1 do
  x := i^2;
  x := x - 1
end_for

hold(
  _for_in(x, object, body)
)


for x in object do
  body
end_for
```

Parameters

i, x

The loop variable: an identifier or a local variable (DOM_VAR) of a procedure

start

The starting value for i: a real number. This may be an integer, a rational number, or a floating point number.

stop

The stopping value for i: a real number. This may be an integer, a rational number, or a floating point number.

stepwidth

The step width: a positive real number. This may be an integer, a rational number, or a floating-point number. The default value is 1.

object

An arbitrary MuPAD object

body

The body of the loop: an arbitrary sequence of statements

Return Values

Value of the last command executed in the body of the loop. If no command was executed, the value NIL is returned. If the iteration range is empty, the void object of type DOM_NULL is returned.

See Also

MuPAD Functions
$ | break | next | repeat | while

More About

- “Loops”

forceGarbageCollection

Force a garbage collection

Syntax

```
forceGarbageCollection()
```

Description

`forceGarbageCollection()` forces a garbage collection to be performed. This function serves a highly technical purpose. Usually, there should be no need for a user to call this function.

Each time the interactive level is reached, the garbage collection routine is called. A heuristic algorithmn decides whether a garbage collection is really performed. After a call to `forceGarbageCollection`, a garbage collection will be forced on the next call of the garbage collection routine.

Note: `forceGarbageCollection` does not cause an immediate garbage collection; it is only executed on returning to the interactive level. Therefore, it cannot be used in procedures to release memory during a longer computation.

Examples

Example 1

When the interactive level is reached, a garbage collection is performed:

```
forceGarbageCollection()
```

Return Values

Void object of type `DOM_NULL`.

See Also

MuPAD Functions
`bytes`

forget

Clear the remember table of a procedure

Syntax

```
forget(f)
```

Description

`forget(f)` clears the remember table of a procedure `f`. The `forget` function clears only remember tables created by the option remember.

The `forget` function clears only remember tables created by the option remember. The function does not affect the remember tables created by `prog::remember`.

Do not call the `forget` function for predefined MuPAD functions. Many predefined MuPAD functions have special values stored in their remember tables. The `forget` function does not throw an error when you call it for a predefined MuPAD function.

The `forget` function does not work recursively. If an inner procedure in a nested procedure uses the option remember, the `forget` function does not clear the remember table created for the inner procedure.

Examples

Example 1

If you use the option remember in a procedure, MuPAD stores all input arguments you used in the procedure calls as indices of the remember table, and the corresponding results as values of these entries. For example, create the following procedure `f` as a wrapper for the MuPAD `sign` function. Use the option remember to enable the remember mechanism for the procedure `f`:

```
f := proc(x)
option remember;
```

```
begin
  sign(x)
end:
```

Now compute the `sign` function for the values -1, 0, and 1:

```
f(-1), f(0), f(1)
```

$$-1, 0, 1$$

You can define a different value for `sign(0)`. First use the `unprotect` function to be able to overwrite the value of `sign`. Then assign the new value to `sign(0)`:

```
unprotect(sign):
sign(0):= 1/2:
```

Although you specified the new value for `sign(0)`, MuPAD does not recalculate the result of the function call `f(0)`. Instead, the system returns the result stored in the remember table:

```
f(0)
```

$$0$$

To clear a remember table created by the option `remember`, use the `forget` function:

```
forget(f):
f(0)
```

$$\frac{1}{2}$$

If you assign a value to a function call, calling the `forget` function also clears that value:

```
f(2) := 1/3:
f(2)
```

$$\frac{1}{3}$$

```
forget(f):
f(2)
```

```
    1
```

For further computations, restore the `sign` function to its default definition. Use the `protect` function with the `ProtectLevelError` option to prevent further changes to `sign`. Also, delete the procedure `f`:

```
sign(0):= 0:
protect(sign, ProtectLevelError):
delete f
```

Parameters

`f`

A procedure or function environment

Return Values

Void object of domain type `DOM_NULL`

See Also

MuPAD Functions

`proc` | `prog::remember`

More About

- “Clear Remember Tables”

fourier

Fourier transform

Syntax

```
fourier(f, t, w)
```

Description

`fourier(f, t, w)` computes the Fourier transform of the expression `f = f(t)` with respect to the variable `t` at the point `w` and is defined as follows:

$$F(w) = c \int_{-\infty}^{\infty} f(t)\, e^{i\, s\, w\, t}\, dt$$

.

`c` and `s` are parameters of the Fourier transform. By default, `c = 1` and `s = -1`.

To change the parameters `c` and `s` of the Fourier transform, use `Pref::fourierParameters`. See “Example 3” on page 1-791. Common choices for the parameter `c` are 1, $\frac{1}{2\pi}$, or $\frac{1}{\sqrt{2\pi}}$. Common choices for the parameter `s` are -1, 1, - 2 π, or 2 π.

If `fourier` cannot find an explicit representation of the transform, it returns an unevaluated function call. See “Example 4” on page 1-791.

If `f` is a matrix, `fourier` applies the Fourier transform to all components of the matrix.

To compute the inverse Fourier transform, use `ifourier`.

To compute the discrete Fourier transform, use `numeric::fft`.

Environment Interactions

Results returned by `fourier` depend on the current `Pref::fourierParameters` settings.

Examples

Example 1

Compute the Fourier transform of this expression with respect to the variable `t`:

```
fourier(exp(-t^2), t, w)
```

$$\sqrt{\pi}\, e^{-\frac{w^2}{4}}$$

Example 2

Compute the Fourier transform of this expression with respect to the variable `t` for positive values of the parameter w_0:

```
assume(w_0 > 0):
F := fourier(t*exp(-w_0^2*t^2), t, w)
```

$$-\frac{\sqrt{\pi}\, w\, e^{-\frac{w^2}{4 w_0^2}}\, i}{2 w_0^3}$$

Evaluate the Fourier transform of the expression at the points $w = 2\, w_0$ and $w = 5$. You can evaluate the resulting expression `F` using `|` (or its functional form `evalAt`):

```
F | w = 2*w_0
```

$$-\frac{\sqrt{\pi}\, e^{-1}\, i}{w_0^2}$$

Also, you can evaluate the Fourier transform at a particular point directly:

```
fourier(t*exp(-w_0^2*t^2), t, 5)
```

$$-\frac{5\sqrt{\pi}\,e^{-\frac{25}{4\,w_0^{\,2}}}\,i}{2\,w_0^{\,3}}$$

Example 3

The default parameters of the Fourier transform are `c = 1` and `s = -1`.

```
fourier(t*exp(-t^2), t, w)
```

$$-\frac{\sqrt{\pi}\,w\,e^{-\frac{w^2}{4}}\,i}{2}$$

To change these parameters, use `Pref::fourierParameters` before calling `fourier`:

```
Pref::fourierParameters(1, 1):
```

Evaluate the transform of the same expression with the new parameters:

```
fourier(t*exp(-t^2), t, w)
```

$$\frac{\sqrt{\pi}\,w\,e^{-\frac{w^2}{4}}\,i}{2}$$

For further computations, restore the default values of the Fourier transform parameters:

```
Pref::fourierParameters(NIL):
```

Example 4

If `fourier` cannot find an explicit representation of the transform, it returns an unevaluated call:

```
fourier(besselJ(1, 1/(1 + t^2)), t, w)
```

$$\text{fourier}\left(J_1\left(\frac{1}{t^2+1}\right), t, w\right)$$

`ifourier` returns the original expression:

```
ifourier(%, w, t)
```

$$J_1\left(\frac{1}{t^2+1}\right)$$

Example 5

Compute the following Fourier transforms that involve the Dirac and the Heaviside functions:

```
fourier(t^3, t, w)
```

$$-2\,\pi\,\delta'''(w)\,i$$

```
fourier(heaviside(t - t_0), t, w)
```

$$e^{-t_0\,w\,i}\left(\pi\,\delta(w) - \frac{i}{w}\right)$$

Example 6

The Fourier transform of a function is related to the Fourier transform of its derivative:

```
fourier(diff(f(t), t), t, w)
```

$$w\,\text{fourier}(f(t), t, w)\,i$$

Parameters

f

Arithmetical expression or unevaluated function call of type `fourier`. If the first argument is a matrix, the result is returned as a matrix.

t

Identifier or indexed identifier representing the transformation variable

w

Arithmetical expression representing the evaluation point

Return Values

Arithmetical expression or matrix of such expressions

Overloaded By

f

References

F. Oberhettinger, “Tables of Fourier Transforms and Fourier Transforms of Distributions”, Springer, 1990.

See Also

MuPAD Functions

`fourier::addpattern` | `ifourier` | `ifourier::addpattern` | `numeric::fft` | `numeric::invfft` | `Pref::fourierParameters`

fourier::addpattern

Add patterns for the Fourier transform

Syntax

```
fourier::addpattern(pat, t, w, res, <vars, <conds>>)
```

Description

`fourier::addpattern(pat, t, w, res)` teaches `fourier` to return `res` for the expression `pat`.

The `fourier` function uses a set of patterns for computing Fourier transforms. You can extend the set by adding your own patterns. To add a new pattern to the pattern matcher, use `fourier::addpattern`. MuPAD does not save custom patterns permanently. The new patterns are available in the *current* MuPAD session only.

After the call `fourier::addpattern(pat, t, w, res)`, the `fourier` function returns `res` for the expression `pat`. Note that the Fourier transform is defined as $c \int_{-\infty}^{\infty} \mathrm{pat}\, \mathrm{e}^{s\,i\,w\,t}\, \mathrm{d}\,t$, where `c` and `s` are the parameters specified by `Pref::fourierParameters`. If you add a new pattern, and then change the Fourier transform parameters, the result returned by `fourier(pat, t, w)` will also change. See “Example 2” on page 1-796.

Variable names that you use when calling `fourier::addpattern` can differ from the names that you use when calling `fourier`. See “Example 3” on page 1-796.

You can include a list of free parameters and a list of conditions on these parameters. These conditions and the result are protected from premature evaluation. This means that you can use `not iszero(a^2 - b)` instead of `hold( _not @ iszero )(a^2 - b)`.

The following conditions treat assumptions on identifiers differently:

- `a^2 - b <> 0` takes into account assumptions on identifiers.
- `not iszero(a^2 - b)` disregards assumptions on identifiers.

See “Example 4” on page 1-796 and “Example 5” on page 1-797.

Environment Interactions

The Fourier pair `(pat, res)` holds only for the current values of the Fourier transform parameters specified by `Pref::fourierParameters`.

Calling `fourier::addpattern` can change the expressions returned by future calls to `fourier` and `ifourier` in the current MuPAD session.

Examples

Example 1

Compute the Fourier transform of the function `foo`. By default, MuPAD does not have a pattern for this function:

```
fourier(foo(t), t, w)
```

$$\text{fourier}(\text{foo}(t), t, w)$$

Add a pattern for the Fourier transform of `foo` using `fourier::addpattern`:

```
fourier::addpattern(foo(t), t, w, bar(w)):
```

Now `fourier` returns the Fourier transform of `foo`:

```
fourier(foo(t), t, w)
```

$$\text{bar}(w)$$

After you add a new transform pattern, MuPAD can use that pattern indirectly:

```
fourier(t^3 + a*foo(2*t - 4), t, w)
```

$$-2\,\pi\,\delta'''(w)\,\mathrm{i} + \frac{a\,\text{bar}\left(\frac{w}{2}\right)\,\mathrm{e}^{-2\,w\,\mathrm{i}}}{2}$$

Example 2

Add this new Fourier transform pattern for the function `foo`:

```
fourier::addpattern(foo(t), t, w, bar(w)):
fourier(foo(t), t, w)
```

$$\mathrm{bar}(w)$$

Now change the Fourier transform parameters using `Pref::fourierParameters`:

```
Pref::fourierParameters(a, b):
```

Evaluate the transform with the new parameters:

```
fourier(foo(t), t, w)
```

$$a\,\mathrm{bar}(-b\,w)$$

For further computations, restore the default values of the Fourier transform parameters:

```
Pref::fourierParameters(NIL):
```

Example 3

Define the Fourier transform of `foo(x)` using the variables `x` and `y` as parameters:

```
fourier::addpattern(foo(x), x, y, bar(y)):
```

The `fourier` function recognizes the added pattern even if you use other variables as parameters:

```
fourier(foo(t), t, w)
```

$$\mathrm{bar}(w)$$

Example 4

Use assumptions when adding the following pattern for the Fourier transform:

```
fourier::addpattern(foo(x, t), t, w, bar(x, w), [x], [abs(x) < 1]):
fourier(foo(x, t), t, w) assuming -1 < x < 1
```

$$\mathrm{bar}(x, w)$$

If $|x| \geq 1$, you cannot apply this pattern:

```
fourier(foo(x, t), t, w) assuming x > 1
```

$$\mathrm{fourier}(\mathrm{foo}(x, t), t, w)$$

If MuPAD cannot determine whether the conditions are satisfied, it returns a `piecewise` object:

```
fourier(foo(x, t), t, w)
```

$$\begin{cases} \mathrm{bar}(x, w) & \text{if } |x| < 1 \end{cases}$$

Example 5

Add this pattern for the Fourier transform of `f`:

```
fourier::addpattern(f(a, t), t, w, g(a, w)/a):
fourier(f(a, T), T, W)
```

$$\frac{g(a, W)}{a}$$

This pattern holds only when the first argument of `f` is the symbolic parameter `a`. If you use any other value of this parameter, `fourier` ignores the pattern:

```
fourier(f(b, T), T, W);
fourier(f(2, T), T, W)
```

$$\mathrm{fourier}(f(b, T), T, W)$$

$$\mathrm{fourier}(f(2, T), T, W)$$

To use the pattern for arbitrary values of the parameter, declare the parameter `a` as an additional pattern variable:

```
fourier::addpattern(f(a, t), t, w, g(a, w)/a, [a]):
```

Now `fourier` applies the specified pattern for an arbitrary value of `a`:

```
fourier(f(2, T), T, W)
```

$$\frac{g(2, W)}{2}$$

```
fourier(f(a^2 + 1, T), T, W)
```

$$\frac{g(a^2 + 1, W)}{a^2 + 1}$$

Note that the resulting expression `g(a, w)/a` defining the Fourier transform of `f(a, t)` implicitly assumes that the value of `a` is not zero. A strict definition of the pattern is:

```
fourier::addpattern(f(a, t), t, w, g(a, w)/a, [a], [a <> 0]):
```

For this particular pattern, you can omit specifying the assumption `a <> 0` explicitly. If `a = 0`, MuPAD throws an internal "Division by zero." error and ignores the pattern:

```
fourier(f(0, T), T, W)
```

$$\text{fourier}(f(0, T), T, W)$$

Parameters

`pat`

Arithmetical expression in the variable `t` representing the pattern to match

`t`

Identifier or indexed identifier used as a variable in the pattern

`w`

Identifier or indexed identifier used as a variable in the result

`res`

Arithmetical expression in the variable `w` representing the pattern for the result of the transform

`vars`

List of identifiers or indexed identifiers used as "pattern variables" (placeholders in `pat` and `res`). You can use pattern variables as placeholders for almost any MuPAD expressions not containing `t` or `w`. You can restrict them by conditions given in the optional parameter `conds`.

`conds`

List of conditions on the pattern variables

Return Values

Object of type `DOM_NULL`

See Also

MuPAD Functions

`fourier` | `ifourier` | `ifourier::addpattern`

fprint

Write data to file

Syntax

```
fprint(<Unquoted | NoNL>, <Bin | Text>, <Encoding = "encodingValue">, filename, <object
```

```
fprint(<Unquoted | NoNL>, <Encoding = "encodingValue">, n, <object1, object2, …>)
```

Description

`fprint(f, objects)` writes MuPAD objects to the file `f`. The objects are evaluated, the results are stored in the file. These data can be read into another MuPAD session via the functions `finput` and `ftextinput`, respectively.

`fprint(Encoding = "encodingValue", f, objects)` uses the specified encoding. For supported encodings, see “Options” on page 1-805.

The file may be specified directly by its name. In this case, `fprint` creates a new file or overwrites an existing file. `fprint` opens and closes the file automatically.

If `WRITEPATH` does not have a value, `fprint` interprets the file name as a path name relative to the “working folder.”

Note that the meaning of “working folder” depends on the operating system. On Windows systems and on Mac OS X systems, the “working folder” is the folder where MATLAB is installed. On UNIX systems, it is the current working folder in which MATLAB was started; when started from a menu or desktop item, this is typically the user's home folder.

Also absolute path names are processed by `fprint`.

If the filename given ends in “`.gz`”, MuPAD automatically writes a compressed file in `gzip` format. These files are transparently uncompressed when read in again by MuPAD. The `gzip` format is supported by many other programs as well. See “Example 5” on page 1-804.

Instead of a file name, also a file descriptor of a file opened via `fopen` can be used. See “Example 2” on page 1-802. In this case, the data written by `fprint` are appended to the corresponding file. The file is not closed automatically by `fprint` and must be closed by a subsequent call to `fclose`.

Note that `fopen(filename)` opens the file in read-only mode. A subsequent `fprint` command to this file causes an error. Use the `Write` or `Append` option of `fopen` to open the file for writing.

Note: The file descriptor `0` represents the screen. See “Example 4” on page 1-803.

Text output occurs without the Pretty-Printer. A call to `fprint` writes all specified objects into a single line of the text file. A newline character is appended to this line, unless the option `NoNL` is used. By default, the written objects are separated by colons without any further white space. The resulting text data consists of syntactically correct MuPAD code and can be read again using `finput`. With the options `Unquoted` and `NoNL`, neither white space no colons are inserted to separate the objects. The resulting text data cannot be read again using `finput`. See “Example 3” on page 1-802.

Environment Interactions

The function is sensitive to the environment variable `WRITEPATH`. If this variable has a value, the file is created in the corresponding folder. Otherwise, the file is created in the “working folder”.

Examples

Example 1

Write some data to the file “`test`”. By default, this file is created as a binary file:

```
fid := fopen(TempFile, Write, Text):
d := 5:
fprint(fid, d, d*3):
file := fname(fid):
fclose(fid)
```

The file is read into the MuPAD session:

```
finput(file, e, f): d, e, f;
```

```
     5, 5, 15
```

```
delete d, e, f:
```

Example 2

Use a file descriptor to access the file `test`. Several calls to `fprint` append data to the file:

```
n := fopen(file, Write):
fprint(n, (d := 5), d*3):
fprint(n, "more data"):
```

Using a file descriptor, call `fclose` to close the file:

```
fclose(n):
```

The file is read into the MuPAD session, assigning the stored values to the identifiers `e`, `f`, and `g`:

```
finput(file, e, f, g ): e, f, g;
```

```
     5, 15, "more data"
```

```
delete n, d, e, f, g:
```

Example 3

With the option `Unquoted`, character strings are written without quotation marks:

```
fid1 := fopen(TempFile):
fid2 := fopen(TempFile):
file1 := fname(fid1):
file2 := fname(fid2):
fprint(Text, file1, "Hello World!", MuPAD + 1):
fprint(Unquoted, Text, file2, "Hello World!", MuPAD + 1):
```

Creates temporary files have the following content:

```
"Hello World!":MuPAD + 1:

Hello World!MuPAD + 1
```

Use `finput` or `ftextinput` to read the data from the file:

```
finput(file1, a, b):
a, b;

     "Hello World!", MuPAD + 1

ftextinput(file2, c): c

     "Hello World!MuPAD + 1"

delete a, b, c:
```

Example 4

Typically, the `print` function serves for displaying objects on screen. If the object produces a line that is longer than the `TEXTWIDTH` setting, `print` breaks that line into shorter lines and inserts the line continuation characters. To avoid inserting line continuation characters, display long objects on screen by using the `fprint` function with the file descriptor 0. For example, convert the following expression to a TeX formatted string. When you use the `print` function, the resulting string contains the line continuation character (\):

```
print(Unquoted, generate::TeX(diff(1/ln(1/x), x$4)))


\frac{22}{x^4\, {\ln\!\left(\frac{1}{x}\right)}^3} - \frac{6}{x^4\, {\ln\
\!\left(\frac{1}{x}\right)}^2} - \frac{36}{x^4\, {\ln\!\left(\frac{1}{x}\
\right)}^4} + \frac{24}{x^4\, {\ln\!\left(\frac{1}{x}\right)}^5}
```

If you want to use the generated string in TeX, you must remove these additional characters. Also, you can generate the string without these characters by using the `fprint` function:

```
fprint(Unquoted, 0, generate::TeX(diff(1/ln(1/x), x$4)))

\frac{22}{x^4\, {\ln\!\left(\frac{1}{x}\right)}^3} - \frac{6}{x^4\, {\ln\!\left(\frac{
```

Another way to avoid line continuation characters is to increase the TEXTWIDTH setting:

```
defaultWidth := TEXTWIDTH:
TEXTWIDTH := 250:
print(Unquoted, generate::TeX(diff(1/ln(1/x), x$4)));
TEXTWIDTH := defaultWidth:
```

```
\frac{22}{x^4\, {\ln\!\left(\frac{1}{x}\right)}^3} - \frac{6}{x^4\, {\ln\!\left(\frac{
```

Example 5

When writing to a file with a name ending in ".gz", MuPAD creates a compressed file automatically. On a UNIX system, the file command can be used to verify this:

```
fprint(Text, "test.gz", "test"):
system("file test.gz"):
```

```
test.gz: gzip compressed data, from Unix
```

Reading the file from MuPAD does not show a difference, because gzip-compressed files are automatically uncompressed in memory by MuPAD:

```
ftextinput("test.gz")
```

```
""test"."
```

Example 6

To specify the encoding to write data, use Encoding. The Encoding option applies only to text files that are opened using a file name and not a file descriptor. Create a file and write the string "abcäöü" in the encoding "UTF-8":

```
fprint(Unquoted, Text, Encoding="UTF-8", "fprint_test", "abcäöü"):
```

Use ftextinput to read the data with the specified encoding:

```
ftextinput("fprint_test", Encoding="UTF-8")
```

```
"abcäöü"
```

If you do not specify an encoding, the default system encoding is used. Thus, your output might vary from that shown next. Characters unrecognized by the default system encoding are replaced by the default substitution character for that encoding:

```
fprint(Unquoted, Text, "fprint_test", "abcäöü"):
ftextinput("fprint_test")
```

```
    "abc"
```

Parameters

`filename`

The name of a file: a character string

`object`$_1$`, object`$_2$`, …`

Arbitrary MuPAD objects

`n`

A file descriptor provided by `fopen`: a nonnegative integer

Options

`Unquoted`

With this option, character strings are written without quotation marks. Additionally, the control characters `'\n'`, `'\b'`, and `'\t'` in strings are expanded. Furthermore, no colons are inserted between the objects. A newline character is appended to the line written by `fprint`.

This option is relevant for text files only. It is useful for writing user-formated text files. Data written with this option cannot be read again via `finput`.

`NoNL`

This option has the same functionality as `Unquoted`, with the only difference that no newline character is appended to the line written by `fprint`.

Bin, Text

With Bin, the data is stored in MuPAD binary format. With Text, standard ASCII format is used. The default is Bin.

Encoding

This option lets you specify the character encoding to use. The allowed encodings are:

"Big5"	"ISO-8859-1"	"windows-932"
"EUC-JP"	"ISO-8859-2"	"windows-936"
"GBK"	"ISO-8859-3"	"windows-949"
"KSC_5601"	"ISO-8859-4"	"windows-950"
"Macintosh"	"ISO-8859-9"	"windows-1250"
"Shift_JIS"	"ISO-8859-13"	"windows-1251"
"US-ASCII"	"ISO-8859-15"	"windows-1252"
"UTF-8"		"windows-1253"
		"windows-1254"
		"windows-1257"

The default encoding is system dependent. If you specify the encoding incorrectly, characters might read incorrectly. Characters unrecognized by the encoding are replaced by the default substitution character for the specified encoding.

Encodings not listed here can be specified but might not produce correct results.

Return Values

Void object of type DOM_NULL.

See Also

MuPAD Functions

doprint | expr2text | fclose | finput | fname | fopen | fread | ftextinput | import::readbitmap | import::readdata | pathname | print | protocol | read | READPATH | write | WRITEPATH

frac

Fractional part of a number

Syntax

```
frac(x)
```

Description

`frac(x)` represents the "fractional part" `x-floor(x)` of the number `x`.

For complex arguments, `frac` is applied separately to the real and imaginary part.

For real numbers, the value `x-floor(x)` represented by `frac(x)` is a number from the interval $[0, 1)$. For positive arguments, you may think of `frac` as truncating all digits before the decimal point.

For integer arguments, `0` is returned. For rational arguments, a rational number is returned. For arguments that contain symbolic identifiers, symbolic function calls are returned. For floating-point arguments or non-rational exact expressions, floating-point values are returned.

Note: If the argument is a floating-point number of absolute value larger than 10^{DIGITS}, then the result is affected by internal non-significant digits! Cf. "Example 2" on page 1-808.

Note: Exact numerical data that are neither integers nor rational numbers are approximated by floating-point numbers. For such arguments, the result depends on the present value of `DIGITS`! Cf. "Example 3" on page 1-809.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate the fractional part of real and complex numbers:

```
frac(1234), frac(123/4), frac(1.234)
```

$$0,\ \frac{3}{4},\ 0.234$$

```
frac(-1234), frac(-123/4), frac(-1.234)
```

$$0,\ \frac{1}{4},\ 0.766$$

```
frac(3/2 + 7/4*I), frac(4/3 + 1.234*I)
```

$$\frac{1}{2}+\frac{3\,i}{4},\ 0.3333333333+0.234\,i$$

The fractional part of a symbolic numerical expression is returned as a floating-point value:

```
frac(exp(123)), frac(3/4*sin(1) + I*tan(3))
```

$$0.7502040793,\ 0.6311032386+0.8574534569\,i$$

Expressions with symbolic identifiers produce symbolic function calls:

```
frac(x), frac(sin(1) + x^2), frac(exp(-x))
```

$$\mathrm{frac}(x),\ \mathrm{frac}(x^2+\sin(1)),\ \mathrm{frac}\left(e^{-x}\right)$$

Example 2

Care should be taken when computing the fractional part of floating-point numbers of large absolute value:

```
10^13/3.0
```

$$3.333333333\ 10^{12}$$

Note that only the first 10 decimal digits are “significant”. Further digits are subject to round-off effects caused by the internal binary representation. These “insignificant” digits can enter the fractional part:

```
frac(10^13/3.0)
```

$$0.3333282471$$

The mantissa of the next floating-point number does not have enough digits to store “digits after the decimal point”:

```
floor(10^25/9.0), ceil(10^25/9.0), frac(10^25/9.0)
```

$$1111111111111111109246976,\ 1111111111111111109246976,\ 0.0$$

Example 3

Exact numerical expressions are converted to floating-point numbers. Consequently, the present setting of `DIGITS` affects the result:

```
x := 10^30 - exp(30)^ln(10) + 1/3
```

$$\frac{3000000000000000000000000000001}{3} - (e^{30})^{\ln(10)}$$

Note that the exact value of this number is $\frac{1}{3}$. Floating-point evaluation can be subject to severe cancellation:

```
DIGITS := 24: frac(x)
```

$$0.765625$$

The floating-point result is more accurate when a higher precision used:

```
DIGITS := 30: frac(x)
```

```
      0.33333364129066467285156250
```

```
delete x, DIGITS:
```

Parameters

x

An arithmetical expression

Return Values

Arithmetical expression.

Overloaded By

x

See Also

MuPAD Functions
`floor`

frandom

Generate random floating-point numbers

Syntax

```
frandom()
```

```
frandom(seed)
```

Description

`frandom()` returns a pseudo-random floating point number from the interval $[0.0, 1.0)$.

`frandom(seed)` returns a generator of pseudo-random floating-point numbers from the interval $[0.0, 1.0)$.

The calls `frandom()` produce uniformly distributed floating-point numbers from the interval $[0.0, 1.0)$.

`r := frandom(seed)` produces a random number generator `r`. Subsequent calls `r()` return uniformly distributed floating-point numbers from the interval $[0.0, 1.0)$.

Different generators created with the same integer seed generate the same sequences of numbers. See “Example 3” on page 1-813 and “Example 4” on page 1-813.

Generators created with `CurrentTime` use the time (in milliseconds) at their creation as their seed values. Generators created shortly after one another may thus return the same numbers.

Generators created in separate calls to `frandom` do not influence one another.

As for all functions returning floating point numbers, `frandom` reacts to `DIGITS` and returns numbers with the precision set by this variable.

Each time MuPAD is started or re-initialized with the `reset` function, random generators not using `CurrentTime` produce the same sequence of numbers.

`frandom` is the recommended function for generating uniform random floating-point numbers. It is much faster than the function `random` which produces uniform integer numbers.

Note: In contrast to `random`, `frandom` does not react to the environment variable `SEED`.

The function `stats::uniformRandom` allows to produce uniformly distributed floating-point numbers on arbitrary finite intervals. The stats library also provides random generators with various other distributions.

Environment Interactions

`frandom` and the procedures returned by `frandom` are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

`frandom` changes its internal state when generating a number and will thus produce a different number on the next call.

Examples

Example 1

The following call produces a sequence of pseudo-random numbers. Note that an index variable `i` must be used in the construction of the sequence. A call such as `frandom() $8` would produce 8 copies of the same random value:

```
frandom() $ i = 1..8
```

```
0.2703581656, 0.8310371787, 0.153156516, 0.9948127808, 0.2662729021, 0.1801642277,
  0.452083055, 0.6787819563
```

Example 2

`frandom` reacts to `DIGITS`, producing numbers which are equally random in the later digits as in the beginning ones:

```
DIGITS := 200: frandom(), frandom()
```

```
0.35498492614062364307860547970996890040547017454106048832367331188029835842270297\
  74118968367028586508146317918083320620569303174817390996691796086374198337243829\
  95763707222113304267125868781104786816 0.68185881324271781264136955162205580177776\
  50094725776810670342423655782682739382577269197633858281367320718386743709918815\
  44393470550284198189826692626896977688203057551877672629547636797382210153222 11
```

```
delete DIGITS:
```

Example 3

`frandom(seed)`, for some integer value of `seed`, returns a generator of floating-point numbers. For different generators created with the same seed, the sequences of numbers will be identical (apart from the digits cut off when producing numbers at lower settings of `DIGITS`):

```
r1 := frandom(42):
r2 := frandom(42):
r3 := frandom(42):
r1() $ i=1..4;
r2() $ i=1..4;
DIGITS := 20:
r3() $ i=1..4;
```

```
0.9239565296, 0.7847883691, 0.1939738073, 0.8908726445

0.9239565296, 0.7847883691, 0.1939738073, 0.8908726445

0.92395652959956197579, 0.78478836910657330549, 0.19397380730447780085,
  0.89087264450316274217
```

```
delete r1, r2, r3, DIGITS:
```

Example 4

Usually, `frandom` is used to generate experimental input or “random” examples. In these cases, reproducibility is a good thing. However, on occasion a “more random” sequence is

desirable. The usual way to get a random seed in a program is to use the current system time, which can be done by using `CurrentTime` as the value of seed:

```
r := frandom(CurrentTime):

r(), r(), r(), r()
```

```
0.794272125, 0.9179931363, 0.2210726413, 0.7790319119
```

Parameters

`seed`

An initialization value for the generator: an integer or the option `CurrentTime`

Return Values

`frandom()` returns a floating point number; `frandom(seed)` returns a procedure (a pseudo-random number generator).

Algorithms

`frandom` uses a linear congruence generator to directly manipulate the internal representation of a `DOM_FLOAT`.

See Also

MuPAD Functions

`random` | `stats::uniformRandom`

fread

Read and execute file

Syntax

```
fread(filename | n, <Quiet>, <Plain>, <Encoding = "encodingValue">)
```

Description

`fread(file)` reads and executes a MuPAD file.

`fread(filename)` reads the file and evaluates each MuPAD statement in the file. If the filename ends in "`.gz`" and the file is in `gzip`-compressed format, it will be transparently uncompressed upon reading. `fread` automatically opens the file, performs the read operation, and closes the file.

`fread(..., Encoding = "encodingValue")` uses the specified encoding. For supported encodings, see "Options" on page 1-819. You can use this option with the previously specified syntaxes.

`fread` is similar to `read`. The only difference is that `fread` does not search for files in the folders given by `READPATH` and by the library path; `fread` only searches for the file relative to the "working folder."

Note that the meaning of "working folder" depends on the operating system. On Windows systems and on Mac OS X systems, the "working folder" is the folder where MATLAB is installed. On UNIX systems, it is the current working folder in which MATLAB was started. When started from a menu or desktop item, this is typically the user's home folder.

Also absolute path names are processed by `fread`.

`fread` can read MuPAD binary files (created via `fprint` or `write`) as well as ASCII text files. `fread` recognizes the format of the file automatically.

Instead of a file name, a file descriptor of a file opened via `fopen` can also be used. See "Example 3" on page 1-818. When a file descriptor is used, `fread` does not automatically open and close the file. `fclose` must be used to close the file.

When a file is read with `fread`, the variable `FILEPATH` contains the path of the file.

Examples

Example 1

Create a new file in the system's temporary folder. The name of the temporary folder varies for different platforms. The `fopen` command with the `TempFile` option creates a file in any system's temporary folder (if such folder exists):

```
fid := fopen(TempFile, Write, Text):
fprint(Unquoted, fid, "a := 3; b := 5; a + b;"):
```

Use `fname` to return the name of the temporary file you created. Use `fclose` to close the file:

```
file := fname(fid):
fclose(fid)
```

When reading the file, MuPAD executes the statements. Each produces a print output. The second `8` below is the return value of `fread`:

```
delete a, b:
fread(file);
```

```
    3

    5

    8

    8
```

Now, the variables `a` and `b` have the values assigned inside the file:

```
a, b
```

```
    3, 5
```

With the option `Quiet`, only the return value of `fread` is printed:

```
delete a, b:
fread(file, Quiet)
```

```
    8
```

```
delete a, b:
```

Example 2

The next example demonstrates the option `Plain`. First, an appropriate input file is created:

```
fid := fopen(TempFile, Write, Text):
fprint(Unquoted, fid,
       "f := proc(x) begin x^2 end_proc:",
       "a := f(3): b := f(4):"):
file := fname(fid):
fclose(fid)
```

Define an `alias` for `f`:

```
alias(f = "some text"):
```

An error occurs if you try to read the file without the option `Plain`. In the parser context of the MuPAD session, the alias replaces `f` by the corresponding string in the assignment `f := ...`. However, strings cannot be assigned a value:

```
fread(file)
```

```
Error: Invalid left-hand side. [_assign]
  Reading File: /tmp/mupad.351omQ
```

With the option `Plain`, no such error arises: the alias for `f` is ignored by `fread`:

```
fread(file, Plain):
a, b
```

$9, 16$

```
unalias(f):
delete f, a, b:
```

Example 3

You use `write` to save the value of the identifier `a` in a temporary file:

```
a := PI + 1:
fid := fopen(TempFile, Write, Text):
file := fname(fid):
write(file, a):
delete a:
```

This file is opened for reading with `fopen`:

```
n := fopen(file):
```

The file descriptor returned by `fopen` can be passed to `fread`. Reading the file restores the value of `a`:

```
fread(n):
a
```

$\pi + 1$

```
fclose(n):
delete a:
```

Example 4

To specify the encoding for reading data, use Encoding. The Encoding option applies only to text files that are opened using a file name and not a file descriptor. Open a temporary file and write the statement `str := "abcäöü"` in the encoding “`UTF-8`”:

```
fprint(Unquoted, Text, Encoding="UTF-8",
                       "fread_test",
                       "str := \"abcäöü\"")
```

Specify the encoding to read the file. `fread` executes the statement:

```
fread("fread_test", Encoding="UTF-8"):

    "abcäöü"
```

If you do not specify an encoding, the default system encoding is used. Thus, your output might vary from that shown next. Characters unrecognized by the default system encoding are replaced by the default substitution character for that encoding:

```
fread("fread_test"):

    "abc??????"
```

Parameters

`filename`

The name of a file: a character string

`n`

A file descriptor provided by `fopen`: a positive integer

Options

`Plain`

Makes `fread` use its own parser context

With this option, the file is read in a new parser context. This means that the history, as returned by the command `history`, is not modified by the statements in the file. Further, abbreviations set outside the file via `alias` or user-defined operators are ignored during the execution of the file. This option is useful for reading initialization files in a clean environment.

`Quiet`

Suppresses output during execution of `fread`

With this option, output is suppressed while reading and executing the file. However, warnings and error messages as well as the output of `print` commands are still visible.

`Encoding`

This option lets you specify the character encoding to use. The allowed encodings are:

`"Big5"`	`"ISO-8859-1"`	`"windows-932"`
`"EUC-JP"`	`"ISO-8859-2"`	`"windows-936"`
`"GBK"`	`"ISO-8859-3"`	`"windows-949"`
`"KSC_5601"`	`"ISO-8859-4"`	`"windows-950"`
`"Macintosh"`	`"ISO-8859-9"`	`"windows-1250"`
`"Shift_JIS"`	`"ISO-8859-13"`	`"windows-1251"`
`"US-ASCII"`	`"ISO-8859-15"`	`"windows-1252"`
`"UTF-8"`		`"windows-1253"`
		`"windows-1254"`
		`"windows-1257"`

The default encoding is system dependent. If you specify the encoding incorrectly, characters might read incorrectly. Characters unrecognized by the encoding are replaced by the default substitution character for the specified encoding.

Encodings not listed here can be specified but might not produce correct results.

Return Values

Return value of the last statement of the file.

See Also

MuPAD Functions

`fclose` | `FILEPATH` | `finput` | `fname` | `fopen` | `fprint` | `ftextinput` | `import::readbitmap` | `import::readdata` | `input` | `pathname` | `print` | `protocol` | `read` | `READPATH` | `textinput` | `write` | `WRITEPATH`

freeIndets

Free indeterminates of an expression

Syntax

```
freeIndets(object, <All>)
```

Description

`freeIndets(object)` returns the free indeterminates of `object` as a set.

An identifier occurring in `object` is free if it cannot be replaced by another identifier without changing the mathematical meaning of `object`.

By default, `freeIndets` does not return free identifiers that occur only in the `0`th operand of subexpressions of `object`.

The special identifiers `PI`, `EULER`, `CATALAN` are not free indeterminates. See “Example 1” on page 1-821.

If `object` is a polynomial, a function environment, a procedure, or a built-in kernel function, then `freeIndets` returns the empty set. See “Example 3” on page 1-822.

Examples

Example 1

Find free identifiers in the following image set. In this set, `PI` is a mathematical constant; therefore, it is not a free identifier. The operand `f` is a `0`th operand. The variable `k` is not a free identifier because you can replace it by any other letter like `m` or `n` without changing the mathematical meaning. Therefore, only `u` is a free identifier:

```
e:= Dom::ImageSet(k*f(u)+PI, k, Z_)
```

$$\{\pi + k\, f(u) \mid k \in \mathbb{Z}\}$$

```
freeIndets(e)
```

$$\{u\}$$

To find all identifiers in the same image set, use `indets`:

```
indets(e)
```

$$\{\pi, k, u\}$$

Example 2

Use the `All` option to return free identifiers including the `0`th operands of subexpressions. For example, compare the sets of free identifiers returned by `freeIndets` with the `All` option and without this option:

```
e := Dom::ImageSet(k*f(u)+PI, k, Z_):
freeIndets(e, All);
freeIndets(e)
```

$$\{\_mult, \_plus, f, u\}$$

$$\{u\}$$

Example 3

`freeIndets` assumes that polynomials and functions do not have free indeterminates:

```
delete x, y:
freeIndets(poly(x*y, [x, y])),
freeIndets(sin),
freeIndets(x -> x^2+1)
```

$$\varnothing, \varnothing, \varnothing$$

Parameters

`object`

An arbitrary object

Options

`All`

Do not exclude free identifiers that occur in the `0`th operand of subexpressions of `object`.

With this option, `freeIndets` does not exclude the `0`th operand. If the `0`th operand of a subexpression is an indeterminate, such as `sin`, the `freeIndets` function includes this operand in the result. See “Example 2” on page 1-822.

Return Values

set of identifiers.

Overloaded By

object

Algorithms

If `object` is an element of a library domain `T` that has a slot `"freeIndets"`, then MuPAD calls the slot routine `T::freeIndets` with `object` as an argument. You can use this approach to extend the functionality of `freeIndets` to user-defined domains. If no such slot exists, then `freeIndets` regards all identifiers occurring in elements of that domain as free, with the exception of mathematical constants.

See Also

MuPAD Functions

`domtype` | `indets` | `op` | `type` | `Type::Indeterminate`

freeze

Create an inactive copy of a function

Syntax

```
freeze(f)
```

Description

`freeze(f)` creates an inactive copy of the function `f`.

`ff := freeze(f)` returns a function that is an "inactive" copy of the argument `f`. This means:

1. `ff` only evaluates its arguments, but does not compute anything else,
2. `ff` is printed in the same way as `f`,
3. symbolic `ff` calls have the same type as symbolic `f` calls,
4. if `f` is a function environment, then `ff` has all the slots of `f`.

Note that `ff` evaluates its incoming parameters even if the function `f` has the procedure option `hold`.

`freeze` can be used when many operations with `f` are to be performed that require `f` only in its symbolic form, but `f` need not be executed.

Neither `eval` nor `level` can enforce the evaluation of an inactive function. See "Example 2" on page 1-825.

Examples

Example 1

We create an inactive form of the function environment `int`:

```
_int := freeze(int): F := _int(x*exp(x^2), x = 0..1)
```

$$\int_0^1 x\, e^{x^2}\, dx$$

The inactive form of `int` keeps every information that is known about the function `int`, e.g., the output, the type, and the `"float"` slot for floating-point evaluation:

```
F, type(F), float(F)
```

$$\int_0^1 x\, e^{x^2}\, dx,\ \text{"int"},\ 0.8591409142$$

The original function environment `int` is not modified by `freeze`:

```
int(x*exp(x^2), x = 0..1)
```

$$\frac{e}{2} - \frac{1}{2}$$

Use `unfreeze` to reactivate the inactive function `_int` and evaluate the result:

```
unfreeze(F), unfreeze(F + 1/2)
```

$$\frac{e}{2} - \frac{1}{2},\ \frac{e}{2}$$

Example 2

We demonstrate the difference between `hold` and `freeze`. The result of the command `S := hold(sum)(...)` does not contain an inactive version of `sum`, but the unevaluated identifier`sum`:

```
S := hold(sum)(1/n^2, n = 1..infinity)
```

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

The next time S is evaluated, the identifier sum is replaced by its value, the function environmentsum, and the procedure computing the value of the infinite sum is invoked:

```
S
```

$$\frac{\pi^2}{6}$$

In contrast, evaluation of the result of freeze does not lead to an evaluation of the inactive function:

```
S := freeze(sum)(1/n^2, n = 1..infinity)
```

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

```
S
```

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

An inactive function does not even react to eval:

```
eval(S)
```

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

The only way to undo a freeze is to use unfreeze, which reactivates the inactive function in S and then evaluates the result:

```
unfreeze(S)
```

$$\frac{\pi^2}{6}$$

Example 3

Note that `freeze(f)` does not change the object `f` but returns a copy of `f` in an inactive form. This means that computations with the inactive version of `f` may contain the original function `f`.

For example, if we create an inactive version of the sine function:

```
Sin := freeze(sin):
```

and expand the term `Sin(x+y)`, then the result is expressed in terms of the (original) sine function `sin`:

```
expand(Sin(x + y))
```

$$\cos(x)\,\sin(y) + \cos(y)\,\sin(x)$$

Example 4

The function `unfreeze` uses `misc::maprec` to operate recursively along the structure of `object`. For example, if `object` is an array containing inactive functions, such as:

```
a := array(1..2,
  [freeze(int)(sin(x), x = 0..2*PI), freeze(sum)(k^2, k = 1..n)]
)
```

$$\left(\int_0^{2\,\pi} \sin(x)\,\mathrm{d}\,x \quad \sum_{k=1}^{n} k^2 \right)$$

then `unfreeze(a)` operates on the operands of `a`:

```
unfreeze(a)
```

$$\left(0 \quad \frac{n\,(2\,n+1)\,(n+1)}{6} \right)$$

This means that for library domains, the effect of `unfreeze` is specified by the method `"maprec"`. If the domain does not implement this method, then `unfreeze` does not operate on the objects of this domain. For example, we create a dummy domain and an object containing an inactive function as its operand:

```
dummy := newDomain("dummy"):
o := new(dummy, freeze(int)(sin(x), x = 0..2*PI))
```

$$\mathrm{new}\left(\mathrm{dummy}, \int_0^{2\pi} \sin(x)\,\mathrm{d}x\right)$$

The function `unfreeze` applied to the object `o` has no effect:

```
unfreeze(o)
```

$$\mathrm{new}\left(\mathrm{dummy}, \int_0^{2\pi} \sin(x)\,\mathrm{d}x\right)$$

If we overload the function `misc::maprec` in order to operate on the first operand of objects of the domain `dummy`, then `unfreeze` operates on `o` as desired:

```
dummy::maprec :=
  x -> extsubsop(x,
    1 = misc::maprec(extop(x,1), args(2..args(0)))
  ):
unfreeze(o)
```

$$\mathrm{new}(\mathrm{dummy}, 0)$$

Parameters

`f`

A procedure or a function environment

Return Values

`freeze` returns an object of the same type as `f`. `unfreeze` returns the evaluation of `object` after reactivating all inactive functions in it.

See Also

MuPAD Functions
`eval` | `hold` | `MAXDEPTH` | `unfreeze`

unfreeze

Create an active copy of a frozen function

Syntax

```
unfreeze(object)
```

Description

`unfreeze(object)` reactivates all inactive functions occurring in `object`, proceeding recursively along the structure of `object`, and then evaluates the result.

`unfreeze` uses `misc::maprec` to proceed recursively along the structure of `object`. This means that for basic domains such as arrays, tables, lists, or polynomials, the function `unfreeze` is applied to each operand of `object`.

Note that if `object` is an element of a library domain, then the behavior of `unfreeze` is specified by the method `"maprec"` which overloads the function `misc::maprec`. If this method does not exist, then `unfreeze` has no effect on `object`. See “Example 4” on page 1-833.

`unfreeze` does not operate on the body of procedures, therefore it is recommended not to embed inactive functions inside procedures.

Examples

Example 1

We create an inactive form of the function environment `int`:

```
_int := freeze(int): F := _int(x*exp(x^2), x = 0..1)
```

$$\int_0^1 x\, e^{x^2}\, dx$$

The inactive form of `int` keeps every information that is known about the function `int`, e.g., the output, the type, and the `"float"` slot for floating-point evaluation:

```
F, type(F), float(F)
```

$$\int_0^1 x\, e^{x^2}\, dx,\ \text{"int"},\ 0.8591409142$$

The original function environment `int` is not modified by `freeze`:

```
int(x*exp(x^2), x = 0..1)
```

$$\frac{e}{2} - \frac{1}{2}$$

Use `unfreeze` to reactivate the inactive function `_int` and evaluate the result:

```
unfreeze(F), unfreeze(F + 1/2)
```

$$\frac{e}{2} - \frac{1}{2},\ \frac{e}{2}$$

Example 2

We demonstrate the difference between `hold` and `freeze`. The result of the command `S := hold(sum)(...)` does not contain an inactive version of `sum`, but the unevaluated identifier`sum`:

```
S := hold(sum)(1/n^2, n = 1..infinity)
```

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

The next time `S` is evaluated, the identifier `sum` is replaced by its value, the function environment`sum`, and the procedure computing the value of the infinite sum is invoked:

```
S
```

$$\frac{\pi^2}{6}$$

In contrast, evaluation of the result of `freeze` does not lead to an evaluation of the inactive function:

```
S := freeze(sum)(1/n^2, n = 1..infinity)
```

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

```
S
```

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

An inactive function does not even react to `eval`:

```
eval(S)
```

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

The only way to undo a `freeze` is to use `unfreeze`, which reactivates the inactive function in `S` and then evaluates the result:

```
unfreeze(S)
```

$$\frac{\pi^2}{6}$$

Example 3

Note that `freeze(f)` does not change the object `f` but returns a copy of `f` in an inactive form. This means that computations with the inactive version of `f` may contain the original function `f`.

For example, if we create an inactive version of the sine function:

```
Sin := freeze(sin):
```

and expand the term `Sin(x+y)`, then the result is expressed in terms of the (original) sine function `sin`:

```
expand(Sin(x + y))
```

$$\cos(x)\,\sin(y) + \cos(y)\,\sin(x)$$

Example 4

The function `unfreeze` uses `misc::maprec` to operate recursively along the structure of `object`. For example, if `object` is an array containing inactive functions, such as:

```
a := array(1..2,
  [freeze(int)(sin(x), x = 0..2*PI), freeze(sum)(k^2, k = 1..n)]
)
```

$$\left(\int_0^{2\,\pi} \sin(x)\,\mathrm{d}\,x \quad \sum_{k=1}^{n} k^2 \right)$$

then `unfreeze(a)` operates on the operands of `a`:

```
unfreeze(a)
```

$$\left(0 \quad \frac{n\,(2\,n+1)\,(n+1)}{6} \right)$$

This means that for library domains, the effect of `unfreeze` is specified by the method `"maprec"`. If the domain does not implement this method, then `unfreeze` does not operate on the objects of this domain. For example, we create a dummy domain and an object containing an inactive function as its operand:

```
dummy := newDomain("dummy"):
o := new(dummy, freeze(int)(sin(x), x = 0..2*PI))
```

$$\mathrm{new}\left(\mathrm{dummy},\ \int_0^{2\,\pi} \sin(x)\,\mathrm{d}\,x \right)$$

The function `unfreeze` applied to the object `o` has no effect:

```
unfreeze(o)
```

$$\text{new}\left(\text{dummy}, \int_0^{2\pi} \sin(x)\, \mathrm{d}x\right)$$

If we overload the function `misc::maprec` in order to operate on the first operand of objects of the domain `dummy`, then `unfreeze` operates on `o` as desired:

```
dummy::maprec :=
  x -> extsubsop(x,
    1 = misc::maprec(extop(x,1), args(2..args(0)))
  ):
unfreeze(o)
```

$$\text{new}(\text{dummy}, 0)$$

Parameters

`object`

Any MuPAD object

Return Values

`freeze` returns an object of the same type as `f`. `unfreeze` returns the evaluation of `object` after reactivating all inactive functions in it.

See Also

MuPAD Functions

`eval` | `freeze` | `hold` | `MAXDEPTH`

fresnelC

The Fresnel cosine integral function

Syntax

```
fresnelC(z)
```

Description

`fresnelC(z)` = $\int_0^z \cos\left(\frac{\pi t^2}{2}\right) \mathrm{d}\, t$.

The function C = `fresnelC` is analytic throughout the complex plane. It satisfies `fresnelC(-z) = -fresnelC(z)`, `fresnelC(conjugate(z)) = conjugate(fresnelC(z))`, `fresnelC(I*z) = I*fresnelC(z)` for all complex values of z.

`fresnelC(z)` returns special values for $z = 0$, $z = \pm\infty$, and $z = \pm i\,\infty$. Symbolic function calls are returned for all other symbolic values of z. In the graphical user interface of MuPAD symbolic function calls are typeset as `fresnelC(z)` = $C(z)$.

When called with floating-point arguments, the function returns floating-point values.

With `simplify` and `Simplify`, the reflection rule `fresnelC(-z) = -fresnelC(z)` is used to create a "normal form" of symbolic function calls. Cf. "Example 3" on page 1-837.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We call the Fresnel functions with various arguments: Symbolic calls are typeset as `fresnelC(z)` = $C(z)$ and `fresnelS(z)` = $S(z)$, respectively:

```
fresnelC(0),
fresnelC(1),
fresnelC(PI + I),
fresnelC(z),
fresnelC(infinity)
```

$$0,\ C(1),\ C(\pi + i),\ C(z),\ \frac{1}{2}$$

```
fresnelS(0),
fresnelS(1),
fresnelS(PI + I),
fresnelS(z),
fresnelS(infinity)
```

$$0,\ S(1),\ S(\pi + i),\ S(z),\ \frac{1}{2}$$

Floating point values are returned for floating-point arguments:

```
fresnelC(1.0),
fresnelC(float(PI)),
fresnelS(-3.45 + 0.75*I)
```

$$0.7798934004,\ 0.5236985437,\ 101.6764728 - 115.6164932\,i$$

Example 2

The functions `diff`, `float`, `limit`, and `series` handle expressions involving the Fresnel functions:

```
diff(fresnelC(x), x),
diff(fresnelS(x), x)
```

$$\cos\left(\frac{\pi x^2}{2}\right), \sin\left(\frac{\pi x^2}{2}\right)$$

```
float(fresnelC(PI)),
float(fresnelS(-100))
```

$$0.5236985437, -0.4968169011$$

```
limit(fresnelC(x), x = infinity),
limit(fresnelS(x), x = -infinity)
```

$$\frac{1}{2}, -\frac{1}{2}$$

```
series(fresnelC(x), x = 0),
series(fresnelS(x), x = infinity, 4)
```

$$x - \frac{\pi^2 x^5}{40} + O(x^7), \frac{1}{2} - \frac{\cos\left(\frac{\pi x^2}{2}\right)}{\pi x} - \frac{\sin\left(\frac{\pi x^2}{2}\right)}{\pi^2 x^3} + O\left(\frac{1}{x^4}\right)$$

Example 3

With `simplify` and `Simplify`, the reflection rules `fresnelC(-z) = -fresnelC(z)` and `fresnelS(-z) = -fresnelS(z)` are used to create a “normal form” of symbolic function calls:

```
Simplify(fresnelC(1 - x)),
Simplify(fresnelC(x - 1))
```

$$-C(x-1), C(x-1)$$

```
3*fresnelS(z) + 2*fresnelS(-z)
```

$$2\,S(-z) + 3\,S(z)$$

```
Simplify(%)
```

$S(z)$

Parameters

z

An arithmetical expression

Return Values

Arithmetical expression.

Overloaded By

z

See Also

MuPAD Functions
`fresnelS`

fresnelS

The Fresnel sine integral function

Syntax

```
fresnelS(z)
```

Description

`fresnelS(z)` = $\int_0^z \sin\left(\frac{\pi t^2}{2}\right) dt$.

The function S = `fresnelS` is analytic throughout the complex plane. It satisfies `fresnelS(-z) = -fresnelS(z)`, `fresnelS(conjugate(z)) = conjugate(fresnelS(z))`, `fresnelS(I*z) =-I*fresnelS(z)` for all complex values of z.

`fresnelS(z)` returns special values for $z = 0$, $z = \pm\infty$, and $z = \pm i\,\infty$. Symbolic function calls are returned for all other symbolic values of z. In the graphical user interface of MuPAD symbolic function calls are typeset as `fresnelS(z)` = $S(z)$.

When called with floating-point arguments, the function returns floating-point values.

With `simplify` and `Simplify`, the reflection rule `fresnelS(-z) = -fresnelS(z)` is used to create a "normal form" of symbolic function calls. Cf. "Example 3" on page 1-841.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We call the Fresnel functions with various arguments: Symbolic calls are typeset as `fresnelC(z)` = $C(z)$ and `fresnelS(z)` = $S(z)$, respectively:

```
fresnelC(0),
fresnelC(1),
fresnelC(PI + I),
fresnelC(z),
fresnelC(infinity)
```

$$0,\ C(1),\ C(\pi + i),\ C(z),\ \frac{1}{2}$$

```
fresnelS(0),
fresnelS(1),
fresnelS(PI + I),
fresnelS(z),
fresnelS(infinity)
```

$$0,\ S(1),\ S(\pi + i),\ S(z),\ \frac{1}{2}$$

Floating point values are returned for floating-point arguments:

```
fresnelC(1.0),
fresnelC(float(PI)),
fresnelS(-3.45 + 0.75*I)
```

$$0.7798934004,\ 0.5236985437,\ 101.6764728 - 115.6164932\,i$$

Example 2

The functions `diff`, `float`, `limit`, and `series` handle expressions involving the Fresnel functions:

```
diff(fresnelC(x), x),
diff(fresnelS(x), x)
```

$$\cos\left(\frac{\pi x^2}{2}\right), \sin\left(\frac{\pi x^2}{2}\right)$$

```
float(fresnelC(PI)),
float(fresnelS(-100))
```

$$0.5236985437, -0.4968169011$$

```
limit(fresnelC(x), x = infinity),
limit(fresnelS(x), x = -infinity)
```

$$\frac{1}{2}, -\frac{1}{2}$$

```
series(fresnelC(x), x = 0),
series(fresnelS(x), x = infinity, 4)
```

$$x - \frac{\pi^2 x^5}{40} + O(x^7), \frac{1}{2} - \frac{\cos\left(\frac{\pi x^2}{2}\right)}{\pi x} - \frac{\sin\left(\frac{\pi x^2}{2}\right)}{\pi^2 x^3} + O\left(\frac{1}{x^4}\right)$$

Example 3

With `simplify` and `Simplify`, the reflection rules `fresnelC(-z) = -fresnelC(z)` and `fresnelS(-z) = -fresnelS(z)` are used to create a “normal form” of symbolic function calls:

```
Simplify(fresnelC(1 - x)),
Simplify(fresnelC(x - 1))
```

$$-C(x-1), C(x-1)$$

```
3*fresnelS(z) + 2*fresnelS(-z)
```

$$2\,S(-z) + 3\,S(z)$$

```
Simplify(%)
```

$S(z)$

Parameters

z

An arithmetical expression

Return Values

Arithmetical expression.

Overloaded By

z

See Also

MuPAD Functions
`fresnelC`

ftextinput

Read text file

Syntax

```
ftextinput(filename | n, <Encoding = "encodingValue">)
```

```
ftextinput(filename | n, x1, x2, …, <Encoding = "encodingValue">)
```

Description

`ftextinput(file, x)` reads a line from a text file, interprets the line as a string and assigns this string to the identifier `x`.

`ftextinput(filename)` reads the first line of the text file and returns it as a string to the MuPAD session. If the file is in `gzip`-compressed format and its name ends in "`.gz`", it will be transparently uncompressed upon reading.

`ftextinput(filename, x1, x2, ...)` reads the file line by line. The *i*-th line is converted to a character string and assigned to the identifier x_i. The identifiers are not evaluated while executing `ftextinput`; previously assigned values are overwritten.

`fread(..., Encoding = "encodingValue")` uses the specified encoding. For the supported encodings, see "Options" on page 1-847. You can use this option with any of the previously specified syntaxes.

Instead of a file name, also a file descriptor `n` of a file opened via `fopen` can be used. The functionality is as described above. However, there is one difference: With a file name, `ftextinput` automatically opens the file, performs the operation, and closes the file. A subsequent call to `ftextinput` starts at the beginning of the file. With a file descriptor, the file remains open (use `fclose` to close the file). The next time data are read from this file, the reading continues at the current position. Consequently, a file descriptor should be used, if the individual lines in the file are to be read via several subsequent calls of `ftextinput`. Cf. "Example 2" on page 1-845.

If the number of identifiers specified in the `ftextinput` call is larger than the number of lines in the file, the exceeding identifiers are not assigned any values. In such a case, `ftextinput` returns the void object of type `DOM_NULL`.

`ftextinput` interprets the file name as a pathname relative to the "working directory."

Note that the meaning of "working directory" depends on the operating system. On Windows systems and on Mac OS X systems, the "working directory" is the folder where MATLAB is installed. On UNIX systems, it is the current working directory in which MATLAB was started. When started from a menu or desktop item, this is typically the user's home directory.

Also absolute path names are processed by `ftextinput`.

Expression sequences are not flattened by `ftextinput` and cannot be used to pass several identifiers to `ftextinput`. Cf. "Example 3" on page 1-846.

Examples

Example 1

Use `fprint` to create a text file with three lines:

```
fid := fopen(TempFile, Write, Text):
fprint(Unquoted, fid, "x + 1\n2nd line\n3rd line"):
file := fname(fid):
```

Read the first two lines of the file and assign the corresponding strings to the identifiers `x1` and `x2`:

```
ftextinput(file, x1, x2):
x1, x2
```

```
"x + 1", "2nd line"
```

If you try to read beyond the last line of the file, `ftextinput` returns the void object of type `DOM_NULL`:

```
ftextinput(file, x1, x2, x3, x4); domtype(%)
```

```
DOM_NULL
```

```
x1, x2, x3, x4
```

```
     "x + 1", "2nd line", "3rd line", x4
```

```
delete x1, x2, x3:
```

Example 2

Read some lines from a file using several calls of `ftextinput`. You have to use a file descriptor for reading from the file. The file is opened for reading with `fopen`:

```
fid := fopen(TempFile, Write, Text):
fprint(Unquoted, fid,
       "x + 1\nx + 2\n3rd line\n4th line"):
file := fname(fid):

n := fopen(file):
```

The file descriptor returned by `fopen` can be passed to `ftextinput` for reading the data:

```
ftextinput(n, x1, x2):
x1, x2
```

```
     "x + 1", "x + 2"
```

```
ftextinput(n, x3, x4):
x3, x4
```

```
     "3rd line", "4th line"
```

Finally, close the file and delete the identifiers:

```
fclose(n):
delete n, x1, x2, x3, x4:
```

Alternatively, the contents of a file can be read into a MuPAD session in the following way:

```
n := fopen(file):
for i from 1 to 4 do
   x.i := ftextinput(n)
end_for:
x1, x2, x3, x4
```

```
     "x + 1", "x + 2", "3rd line", "4th line"

fclose(n):
delete n, i, x1, x2, x3, x4:
```

Example 3

Expression sequences are not flattened by `ftextinput` and cannot be used to pass identifiers to `ftextinput`:

```
fid := fopen(TempFile, Write, Text):
fprint(Unquoted, fid, "1st line\n2nd line\n3rd line"):
file := fname(fid):
ftextinput(file, (x1, x2), x3)


Error: The argument is invalid. [ftextinput]
```

The following call does not lead to an error because the identifier `x12` is not evaluated. Consequently, only one line is read from the file and assigned to `x12:`

```
x12 := x1, x2:
ftextinput(file, x12):
x1, x2, x12

     x1, x2, "1st line"

delete x12:
```

Example 4

To specify the encoding for reading data, use Encoding. The Encoding option applies only to text files that are opened using a file name and not a file descriptor. Open a temporary file and write the string `"abcäöü"` in the encoding “`UTF-8`”:

```
fprint(Unquoted, Text, Encoding = "UTF-8",
                       "ftextinput_test",
                       "abcäöü"):
```

Specify the encoding to read the file correctly:

```
ftextinput("ftextinput_test", Encoding="UTF-8")
```

```
"abcäöü"
```

If you do not specify an encoding, the default system encoding is used. Thus, your output might vary from that shown next. Characters unrecognized by the default system encoding are replaced by the default substitution character for that encoding:

```
ftextinput("ftextinput_test")
```

```
"abc??????"
```

Parameters

`filename`

The name of a file: a character string

`n`

A file descriptor provided by `fopen`: a positive integer

$x_1, x_2, \ldots$

identifiers

Options

`Encoding`

This option lets you specify the character encoding to use. The allowed encodings are:

`"Big5"`	`"ISO-8859-1"`	`"windows-932"`
`"EUC-JP"`	`"ISO-8859-2"`	`"windows-936"`
`"GBK"`	`"ISO-8859-3"`	`"windows-949"`
`"KSC_5601"`	`"ISO-8859-4"`	`"windows-950"`

"Macintosh"	"ISO-8859-9"	"windows-1250"
"Shift_JIS"	"ISO-8859-13"	"windows-1251"
"US-ASCII"	"ISO-8859-15"	"windows-1252"
"UTF-8"		"windows-1253"
		"windows-1254"
		"windows-1257"

The default encoding is system dependent. If you specify the encoding incorrectly, characters might read incorrectly. Characters unrecognized by the encoding are replaced by the default substitution character for the specified encoding.

Encodings not listed here can be specified but might not produce correct results.

Return Values

Last line that was read from the file: a character string or null().

See Also

MuPAD Functions

`fclose` | `finput` | `fname` | `fopen` | `fprint` | `fread` | `import::readbitmap` | `import::readdata` | `input` | `pathname` | `print` | `protocol` | `read` | `READPATH` | `textinput` | `write` | `WRITEPATH`

funcenv

Create a function environment

Syntax

```
funcenv(f1, <f2>, <slotTable>)
```

Description

`funcenv(f)` creates a function environment from `f`.

`funcenv` serves for generating a function environment of domain type `DOM_FUNC_ENV`.

From a user's point of view, function environments are similar to procedures and can be called like any MuPAD function.

However, in contrast to simple procedures, a function environment allows a tight integration into the MuPAD system. In particular, standard system functions such as `diff`, `expand`, `float` etc. can be told how to act on symbolic function calls to a function environment.

For this, a function environment stores special function attributes (slots) in an internal table. Whenever an overloadable system function such as `diff`, `expand`, `float` encounters an object of type `DOM_FUNC_ENV`, its searches the function environment for a corresponding slot. If found, it calls the corresponding slot and returns the value produced by the slot.

Slots can be incorporated into the function environment by creating a table `slotTable` and passing this to `funcenv`, when the function environment is created. Alternatively, the function `slot` can be used to add further slots to an existing function environment.

See “Example 1” on page 1-850 below for further information.

The first argument `f1` of `funcenv` determines the evaluation of function calls. With `f:= funcenv(f1)`, the call `f(x)` returns the result `f1(x)`. Note that calls of the form `f:= funcenv(f)` are possible (and, in fact, typical). This call embeds the procedure `f` into a function environment of the same name. The original procedure `f` is stored internally in

the function environment `f`. After this call, further function attributes can be attached to `f` via the `slot` function.

The second argument `f2` of `funcenv` determines the screen output of symbolic function calls. Consider `f:= funcenv(f1, f2)`. If the call `f(x)` returns a symbolic function call `f(x)` with 0-th operand `f`, then `f2` is called: the return value of `f2(f(x))` is used as the screen output of `f(x)`.

Note: Beware: `f2(f(x))` should not produce a result containing a further symbolic call of `f`, because this will lead to an infinite recursion, causing an error message.

The third argument `slotTable` of `funcenv` is a table containing function attributes (slots). The table has to use strings as indices to address system functions. E.g.,

```
   slotTable := table("diff" = mydiff, "float" = myfloat):      f
:= funcenv(f1, f2, slotTable):
```

attaches the slot functions `mydiff` and `myfloat` to `f`. They are called by the system functions `diff` and `float`, respectively, whenever they encounter a symbolic expression `f(x)` with 0-th operand `f`. The internal slot table can be changed or filled with additional function attributes via the function `slot`.

If the first argument `f1` of `funcenv` is itself a function environment, then the return value is a physical copy of `f1`.

The documentation of `float`, `print`, and `slot` provides further examples involving function environments.

Examples

Example 1

We want to introduce a function `f` that represents a solution of the differential equation $f'(x) = x + \sin(x)\, f(x)$. First, we define a function `f`, which returns any call `f(x)` symbolically:

```
f := proc(x) begin procname(args()) end_proc: f(x), f(3 + y)
```

$$f(x),\ f(y+3)$$

Because of the differential equation $f'(x) = x + \sin(x)\, f(x)$, derivatives of `f` can be rewritten in terms of `f`. How can we tell the MuPAD system to differentiate symbolic functions calls such as `f(x)` accordingly? For this, we first have to embed the procedure `f` into a function environment:

```
f := funcenv(f):
```

The function environment behaves like the original procedure:

```
f(x), f(3 + y)
```

$$f(x),\ f(y+3)$$

System functions such as `diff` still treat symbolic calls of `f` as calls to unknown functions:

```
diff(f(x + 3), x)
```

$$f'(x+3)$$

However, as a function environment, `f` can receive attributes that overload the system functions. The following `slot` call attaches a dummy `"diff"` attribute to `f`:

```
f::diff := mydiff:  diff(2*f(x^2) + x, x)
```

$$2\ \mathrm{mydiff}(f(x^2),\ x) + 1$$

We attach a more meaningful `"diff"` attribute to `f` that is based on $f'(x) = x + \sin(x)\, f(x)$. Note that arbitrary calls `diff(f(y), x1, x2, ..)` have to be handled by this slot:

```
fdiff := proc(fcall) local y; begin
    y:= op(fcall, 1);
    (y + sin(y)*f(y))*diff(y, args(2..args(0)))
end_proc:
f := slot(f, "diff", fdiff):
```

Now, as far as differentiation is concerned, the function `f` is fully integrated into MuPAD:

```
diff(f(x), x), diff(f(x), x, x)
```

$$x + f(x)\,\sin(x),\ \cos(x)\,f(x) + \sin(x)\,(x + f(x)\,\sin(x)) + 1$$

```
diff(sin(x)*f(x^2), x)
```

$$f(x^2)\,\cos(x) + 2\,x\,\sin(x)\,(x^2 + f(x^2)\,\sin(x^2))$$

Since Taylor expansion around finite points only needs to evaluate derivatives, also Taylor expansions of `f` can be computed:

```
taylor(f(x^2), x = 0, 9)
```

$$f(0) + x^4\left(\frac{f(0)}{2} + \frac{1}{2}\right) + x^8\left(\frac{f(0)}{12} + \frac{1}{8}\right) + O(x^9)$$

```
delete f, fdiff:
```

Example 2

Suppose that you have defined a function `f` that may return itself symbolically, and you want such symbolic expressions of the form `f(x,...)` to be printed in a special way. To this end, embed your procedure`f` in a function environment and supply an output procedure as second argument to the corresponding `funcenv` call. Whenever an expression of the form `f(x,...)` is to be printed, the output procedure will be called with the arguments `x,...` of the expression:

```
f := funcenv(f,
         proc(x) begin
            if nops(x) = 2 then
              "f does strange things with its arguments ".
              expr2text(op(x, 1))." and ".expr2text(op(x,2))
            else
              FAIL
            end
         end):

delete a, b:
print(f(a, b)/2):
print(f(a, b, c)/2):
```

$$\frac{\text{f does strange things with its arguments a and b}}{2}$$

$$\frac{f(a, b, c)}{2}$$

```
delete f:
```

Parameters

`f1`

An arbitrary MuPAD object. Typically, a procedure. It handles the evaluation of a function call to the function environment.

`f2`

A procedure handling the screen output of symbolic function calls

`slotTable`

A table of function attributes (slots)

Return Values

Function environment of type `DOM_FUNC_ENV`.

Algorithms

Mathematical functions such as `exp`, `ln` etc. or `abs`, `Re`, `Im` etc. are implemented as function environments.

See Also

MuPAD Functions
`slot`

More About

- “Integrate Custom Functions into MuPAD”

funm

General matrix function

Syntax

```
funm(A,f)
```

Description

`funm(A,f)` computes the function `f(A)` for the square matrix `A`. For details, see "Algorithms" on page 1-858.

Examples

Example 1

Find a matrix `B`, such that $B^3 = A$, where `A` is the 3-by-3 identity matrix.

To solve $B^3 = A$, compute the cube root of the matrix `A` using the `funm` function. Create the function that computes the cube root of its argument, and use it as the second argument for `funm`. The cube root of an identity matrix is the identity matrix itself.

```
A := matrix::identity(3):
f := x -> surd(x, 3)
```

$$x \to \sqrt[3]{x}$$

```
funm(A, f)
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Replace one of the `0` elements of matrix `A` with `1` and compute the matrix cube root again.

```
A[1, 2] := 1:
```

```
A
```

$$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

```
funm(A, f)
```

$$\begin{pmatrix} 1 & \frac{1}{3} & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Now, compute the cube root of the upper triangular matrix.

```
A[1..2, 3] := [1, 1]:
A
```

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

```
B := funm(A, f)
```

$$\begin{pmatrix} 1 & \frac{1}{3} & \frac{2}{9} \\ 0 & 1 & \frac{1}{3} \\ 0 & 0 & 1 \end{pmatrix}$$

Verify that $B^3 = A$.

```
B^3 = A
```

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Example 2

Find the matrix Lambert W function.

Create the 3-by-3 Pascal matrix `A`.

```
A := linalg::pascal(3)
```

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 3 & 6 \end{pmatrix}$$

To find the Lambert W function (W_0 branch) in a matrix sense, call `funm` using `lambertW` as its second argument. Approximate the result with floating-point numbers by using `float`.

```
W0 := funm(float(A), lambertW)
```

$$\begin{pmatrix} 0.4715911292 & 0.2869500057 & 0.09584568317 \\ 0.2869500057 & 0.5196607317 & 0.5264174526 \\ 0.09584568317 & 0.5264174526 & 1.285252109 \end{pmatrix}$$

Verify that this result is a solution of the matrix equation $A = W0 \cdot e^{W0}$ within the current floating-point accuracy.

```
A = W0*exp(W0)
```

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 3 & 6 \end{pmatrix} = \begin{pmatrix} 1.0 & 1.0 & 1.0 \\ 1.0 & 2.0 & 3.0 \\ 1.0 & 3.0 & 6.0 \end{pmatrix}$$

Now, find the W_{-1} branch of the Lambert W function for the matrix `A`. Create the function `f` representing the branch W_{-1} of the Lambert W function.

```
f := x -> lambertW(-1, x)
```

$$x \to W_{-1}(x)$$

Call `funm` using `f` as its second argument. Approximate the result with floating-point numbers by using `float`.

```
Wm1 := funm(float(A), f)
```

```
[[-2.116372833 - 4.266440523 i, 1.130732372 - 0.2199437576 i,
  -0.03418665371 - 0.002454497511 i],
 [1.130732372 - 0.2199437576 i, -2.441789243 - 4.214522706 i,
  1.353588821 - 0.2792250676 i],
 [-0.03418665371 - 0.002454497511 i, 1.353588821 - 0.2792250676 i,
  -0.248697806 - 4.659319216 i]]
```

Verify that this result is a solution of the matrix equation `A = Wm1·e^Wm1` within the current floating-point accuracy.

```
A = Wm1*exp(Wm1)
```

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 3 & 6 \end{pmatrix} = \begin{pmatrix} 1.0 + 7.047165043\ 10^{-15}\,i & 1.0 + 1.498020458\ 10^{-14}\,i & 1.0 + 8.012687736\ 10^{-15}\,i \\ 1.0 - 1.297139479\ 10^{-14}\,i & 2.0 - 1.910971381\ 10^{-14}\,i & 3.0 - 6.004224895\ 10^{-14}\,i \\ 1.0 + 1.15792792\ 10^{-14}\,i & 3.0 + 3.354608258\ 10^{-14}\,i & 6.0 + 1.192795862\ 10^{-14}\,i \end{pmatrix}$$

Parameters

A

A square array, hfarray, or matrix.

f

A function.

Return Values

An array, hfarray, or matrix.

Algorithms

Suppose `f(x)`, where `x` is a scalar, has a Taylor series expansion. Then the matrix function `f(A)`, where `A` is a matrix, is defined by the Taylor series of `f(A)`, with addition and multiplication performed in the matrix sense.

If `A` can be represented as `A = P·D·P^-1`, where `D` is a diagonal matrix, such that

$$D = \begin{pmatrix} d_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & d_n \end{pmatrix}$$

then the matrix function `f(A)` can be computed as follows:

$$f(A) = P \begin{pmatrix} f(d_1) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & f(d_n) \end{pmatrix} P^{-1}$$

Nondiagonalizable matrices can be represented as `A = P·J·P^-1`, where `J` is a Jordan form of the matrix `A`. (For details, see `linalg::jordanForm`.) Then, the matrix function `f(A)` can be computed by using the following definition on each Jordan block:

$$f\left(\begin{pmatrix} \lambda & 1 & 0 & \cdots & 0 \\ 0 & \lambda & 1 & & \vdots \\ 0 & 0 & \lambda & \ddots & 0 \\ \vdots & & \ddots & \ddots & 1 \\ 0 & \cdots & 0 & 0 & \lambda \end{pmatrix}\right) = \begin{pmatrix} \frac{f(\lambda)}{0!} & \frac{f'(\lambda)}{1!} & \frac{f''(\lambda)}{2!} & \cdots & \frac{f^{(n)}(\lambda)}{n!} \\ 0 & \frac{f(\lambda)}{0!} & \frac{f'(\lambda)}{1!} & \cdots & \frac{f^{(n-1)}(\lambda)}{(n-1)!} \\ 0 & 0 & \frac{f(\lambda)}{0!} & & \vdots \\ \vdots & & & \ddots & \vdots \\ 0 & \cdots & 0 & \frac{f(\lambda)}{0!} & \frac{f'(\lambda)}{1!} \\ 0 & \cdots & 0 & 0 & \frac{f(\lambda)}{0!} \end{pmatrix}$$

See Also

MuPAD Functions

`exp` | `linalg::jordanForm` | `linalg::sqrtMatrix` | `map` | `numeric::expMatrix` | `numeric::fMatrix`

gamma

Gamma function

Syntax

```
gamma(x)
```

```
gamma(iv)
```

Description

`gamma(x)` represents the gamma function $\Gamma(x) = \int_0^\infty e^{-t}\, t^{x-1}\, dt$.

The gamma function is defined for all complex arguments apart from the singular points 0, - 1, - 2, ….

The gamma function is related to the factorial function: `gamma(x)=fact(x-1)=(x-1)!` for all positive integers x.

If `x` is a floating-point value, then `gamma` returns a floating-point value. If `x` is a floating-point interval, a floating-point interval is returned. If `x` is a positive integer not larger than the value given by `Pref::autoExpansionLimit()`, then an integer is returned. (Use `expand(gamma(x))` to get an integer value for larger integers `x`.) If `x` is a rational number of domain type `DOM_RAT` not larger than the value given by `Pref::autoExpansionLimit()`, then the functional relation $\Gamma(x+1) = x\,\Gamma(x)$ is applied to "normalize" the result. (Again, use `expand(gamma(x))` to enforce this normalization for larger rational numbers `x`.) The functional relation

$$\Gamma(x)\,\Gamma(1-x) = \frac{\pi}{\sin(\pi\, x)}$$

is applied if $x < \frac{1}{2}$ is a rational number of domain type `DOM_RAT` that is an integer multiple of $\frac{1}{4}$ or $\frac{1}{6}$. The call `gamma(1/2)` yields `sqrt(PI)`; `gamma(infinity)` yields `infinity`.

For all other arguments, a symbolic function call is returned.

The float attribute of `gamma` is a kernel function, i.e., floating-point evaluation is fast.

The `expand` attribute uses the functional equation $\Gamma(x+1) = x\,\Gamma(x)$, the reflection formula

$$\Gamma(-x) = -\frac{\pi}{x \sin(\pi\, x\, \Gamma(x))},$$

and the Gauß multiplication formula for $\Gamma(k\,x)$ when k is a positive integer, to rewrite `gamma(x)`. Cf. “Example 3” on page 1-863. For numerical `x`, the functional equation is used to shift the argument to the range `0 < x < 1`.

The functional equations for *gamma* lead to various identities for *lngamma* which can be applied via `expand`. Cf. “Example 3” on page 1-863.

The logarithmic derivative of `gamma` is implemented by the digamma function `psi`.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
gamma(15),
gamma(3/2),
gamma(-3/2),
gamma(sqrt(2)),
gamma(x + 1)
```

$$87178291200,\ \frac{\sqrt{\pi}}{2},\ \frac{4\sqrt{\pi}}{3},\ \Gamma(\sqrt{2}),\ \Gamma(x+1)$$

```
lngamma(15),
lngamma(3/2),
lngamma(-3/2),
lngamma(sqrt(2)),
lngamma(x + 1)
```

$$\ln(87178291200),\ \ln\left(\frac{\sqrt{\pi}}{2}\right),\ \ln\Gamma\left(-\frac{3}{2}\right),\ \ln\Gamma(\sqrt{2}),\ \ln\Gamma(x+1)$$

Floating point values are computed for floating-point arguments:

```
gamma(11.5),
gamma(2.0 + 10.0*I)
```

$$11899423.08,\ -0.00001089258677 + 0.00000504737724\,i$$

```
lngamma(11.5),
lngamma(2.0 + 10.0*I)
```

$$16.29200048,\ -11.33017193 + 15.27404065\,i$$

Example 2

`gamma` and `lngamma` are singular for nonpositive integers:

```
gamma(0)
```

```
Error: Singularity. [gamma]
```

```
lngamma(-2)
```

```
Error: Singularity. [lngamma]
```

Example 3

The functions `diff`, `expand`, `float`, `limit`, and `series` handle expressions involving `gamma` and `lngamma`:

```
diff(gamma(x^2 + 1), x),
diff(lngamma(x^2 + 1), x)
```

$$2\,x\,\Gamma(x^2+1)\,\psi(x^2+1),\ 2\,x\,\psi(x^2+1)$$

```
float(ln(3 + gamma(sqrt(PI)))),
float(ln(3 + lngamma(sqrt(PI))))
```

$$1.367203476,\ 1.072044865$$

```
expand(gamma(x + 2)),
expand(lngamma(x + 2))
```

$$x\,\Gamma(x)\,(x+1),\ \ln(x+1)+\ln(x)+\ln\Gamma(x)$$

```
expand(gamma(2*x)),
expand(lngamma(2*x))
```

$$\frac{2^{2\,x}\,\Gamma\left(x+\frac{1}{2}\right)\Gamma(x)}{2\,\sqrt{\pi}},\ \ln\Gamma\left(x+\frac{1}{2}\right)-\ln(2)-\frac{\ln(\pi)}{2}+\ln\Gamma(x)+2\,x\,\ln(2)$$

```
expand(gamma(2*x - 1)),
expand(lngamma(2*x - 1))
```

$$\frac{2^{2\,x}\,\Gamma\left(x+\frac{1}{2}\right)\Gamma(x)}{2\,\sqrt{\pi}\,(2\,x-1)},\ \ln\Gamma\left(x+\frac{1}{2}\right)-\ln(2)-\ln(2\,x-1)-\frac{\ln(\pi)}{2}+\ln\Gamma(x)+2\,x\,\ln(2)$$

```
limit(1/gamma(x), x = infinity),
limit(1/lngamma(x), x = infinity)
```

$$0,\ 0$$

```
limit(gamma(x - 4)/gamma(x - 10), x = 0),
limit(lngamma(x - 4) - lngamma(x - 10), x = 0)
```

$$151200,\ \ln(151200) + 6\,\pi\,i$$

```
series(gamma(x), x = 0, 3),
series(lngamma(x), x = 0, 3)
```

$$\frac{1}{x} - \mathrm{EULER} + x\left(\frac{\mathrm{EULER}^2}{2} + \frac{\pi^2}{12}\right) + O(x^2),\ -\ln(x) - \mathrm{EULER}\,x + \frac{\pi^2\,x^2}{12} + O(x^3)$$

The Stirling formula is obtained as an asymptotic series:

```
series(gamma(x), x = infinity, 4),
series(lngamma(x), x = infinity, 4)
```

$$\frac{\sigma_1}{\sqrt{x}\left(\frac{1}{x}\right)^x} + \frac{\sigma_1}{12\,x^{3/2}\left(\frac{1}{x}\right)^x} + \frac{\sigma_1}{288\,x^{5/2}\left(\frac{1}{x}\right)^x} + O\left(\frac{e^{-x}}{x^{7/2}\left(\frac{1}{x}\right)^x}\right),\ x\,(\ln(x) - 1) + \frac{\ln(2)}{2} + \frac{\ln(\pi)}{2} - \frac{\ln(x)}{2}$$

$$+ \frac{1}{12\,x} + O\left(\frac{1}{x^3}\right)$$

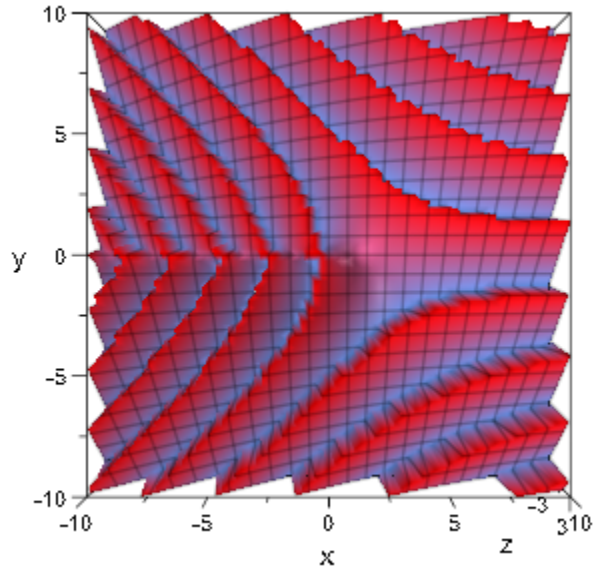
where

$$\sigma_1 = \sqrt{2}\,\sqrt{\pi}\,e^{-x}$$

Example 4

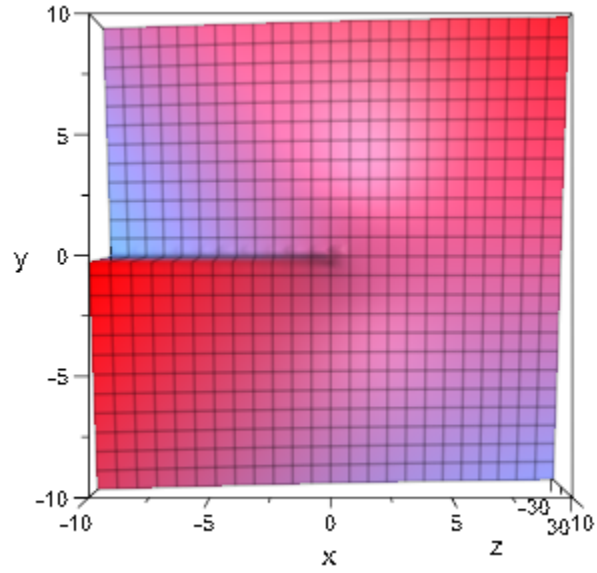
The logarithm function `ln` has a branch cut along the negative real semi axis, where the values jump by $2\,\pi\,i$ when crossing the cut. In the following plot of the imaginary part of the logarithm of the gamma function the lines in the complex z plane with $\Im(\Gamma(z)) = 0$ and $\Re(\Gamma(z)) \le 0$ are clearly visible as discontinuities:

```
plotfunc3d(Im(ln(gamma(x + I*y))), x = -10 .. 10, y = -10 .. 10,
           Submesh = [2, 2], CameraDirection = [0, -1, 1000]):
```



The function `lngamma(z)`, however, adds suitable integer multiples of $2\,\pi\,i$ to `ln(gamma(z))` making the function analytic throughout the complex plane with a branch cut along the negative real semi axis:

```
plotfunc3d(Im(lngamma(x + I*y)), x = -10 .. 10, y = -10 .. 10,
           Submesh = [2, 2], CameraDirection = [0, -1, 1000]):
```



Parameters

x

An arithmetical expression

iv

A floating-point interval

Return Values

Arithmetical expression or a floating-point interval.

Overloaded By

x

See Also

MuPAD Functions

`beta` | `binomial` | `fact` | `harmonic` | `igamma` | `lngamma` | `pochhammer` | `psi`

lngamma

Log-gamma function

Syntax

```
lngamma(x)
```

Description

`lngamma(x)` represents the logarithmic gamma function $\ln\Gamma(x) = \ln(\Gamma(x))$ for positive real `x`.

The logarithmic gamma function is defined for all complex arguments apart from the singular points 0, - 1, - 2, ….

Along the positive real semi axis, the logarithmic gamma function $\ln\Gamma(x)$ coincides with the logarithm $\ln(\Gamma(x))$ of the gamma function. For negative or general complex arguments *x*, however, one has $\ln\Gamma(x) = \ln(\Gamma(x)) + f(x)\ 2\ \pi\ i$ with some integer valued function *f*(*x*). The integer multiples of 2 π i are chosen such that *lngamma* is analytic throughout the complex plane with a branch cut along the negative real semi axes. Cf. “Example 4” on page 1-873. For negative real `x`, the value $\ln\Gamma(x)$ coincides with the limit “from above”.

If the argument `x` is a floating-point value, then `lngamma(x)` returns a floating-point value. For other values of `x` the call `lngamma(x)` returns `ln(gamma(x))` whenever `x` is a positive real number and `gamma(x)` is not returned as a symbolic call. For negative or non-real complex values `x` a symbolic call `lngamma(x)` is returned.

The functional equations for *gamma* lead to various identities for *lngamma* which can be applied via `expand`. Cf. “Example 3” on page 1-871.

The logarithmic derivative of `gamma` is implemented by the digamma function `psi`.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
gamma(15),
gamma(3/2),
gamma(-3/2),
gamma(sqrt(2)),
gamma(x + 1)
```

$$87178291200,\ \frac{\sqrt{\pi}}{2},\ \frac{4\sqrt{\pi}}{3},\ \Gamma(\sqrt{2}),\ \Gamma(x+1)$$

```
lngamma(15),
lngamma(3/2),
lngamma(-3/2),
lngamma(sqrt(2)),
lngamma(x + 1)
```

$$\ln(87178291200),\ \ln\left(\frac{\sqrt{\pi}}{2}\right),\ \ln\Gamma\left(-\frac{3}{2}\right),\ \ln\Gamma(\sqrt{2}),\ \ln\Gamma(x+1)$$

Floating point values are computed for floating-point arguments:

```
gamma(11.5),
gamma(2.0 + 10.0*I)
```

$$11899423.08,\ -0.00001089258677 + 0.00000504737724\,i$$

```
lngamma(11.5),
lngamma(2.0 + 10.0*I)
```

$$16.29200048,\ -11.33017193 + 15.27404065\,i$$

Example 2

`gamma` and `lngamma` are singular for nonpositive integers:

```
gamma(0)
```

```
Error: Singularity. [gamma]
```

```
lngamma(-2)
```

```
Error: Singularity. [lngamma]
```

Example 3

The functions `diff`, `expand`, `float`, `limit`, and `series` handle expressions involving `gamma` and `lngamma`:

```
diff(gamma(x^2 + 1), x),
diff(lngamma(x^2 + 1), x)
```

$$2\,x\,\Gamma(x^2+1)\,\psi(x^2+1),\ 2\,x\,\psi(x^2+1)$$

```
float(ln(3 + gamma(sqrt(PI)))),
float(ln(3 + lngamma(sqrt(PI))))
```

$$1.367203476,\ 1.072044865$$

```
expand(gamma(x + 2)),
expand(lngamma(x + 2))
```

$$x\,\Gamma(x)\,(x+1),\ \ln(x+1)+\ln(x)+\ln\Gamma(x)$$

```
expand(gamma(2*x)),
expand(lngamma(2*x))
```

$$\frac{2^{2\,x}\,\Gamma\left(x+\frac{1}{2}\right)\,\Gamma(x)}{2\,\sqrt{\pi}},\ \ln\Gamma\left(x+\frac{1}{2}\right)-\ln(2)-\frac{\ln(\pi)}{2}+\ln\Gamma(x)+2\,x\,\ln(2)$$

```
expand(gamma(2*x - 1)),
expand(lngamma(2*x - 1))
```

$$\frac{2^{2x}\,\Gamma\left(x+\frac{1}{2}\right)\Gamma(x)}{2\sqrt{\pi}\,(2x-1)},\ \ln\Gamma\left(x+\frac{1}{2}\right)-\ln(2)-\ln(2x-1)-\frac{\ln(\pi)}{2}+\ln\Gamma(x)+2x\ln(2)$$

```
limit(1/gamma(x), x = infinity),
limit(1/lngamma(x), x = infinity)
```

$$0,\ 0$$

```
limit(gamma(x - 4)/gamma(x - 10), x = 0),
limit(lngamma(x - 4) - lngamma(x - 10), x = 0)
```

$$151200,\ \ln(151200)+6\,\pi\,i$$

```
series(gamma(x), x = 0, 3),
series(lngamma(x), x = 0, 3)
```

$$\frac{1}{x}-\mathrm{EULER}+x\left(\frac{\mathrm{EULER}^2}{2}+\frac{\pi^2}{12}\right)+O(x^2),\ -\ln(x)-\mathrm{EULER}\,x+\frac{\pi^2\,x^2}{12}+O(x^3)$$

The Stirling formula is obtained as an asymptotic series:

```
series(gamma(x), x = infinity, 4),
series(lngamma(x), x = infinity, 4)
```

$$\frac{\sigma_1}{\sqrt{x}\left(\frac{1}{x}\right)^x}+\frac{\sigma_1}{12\,x^{3/2}\left(\frac{1}{x}\right)^x}+\frac{\sigma_1}{288\,x^{5/2}\left(\frac{1}{x}\right)^x}+O\left(\frac{e^{-x}}{x^{7/2}\left(\frac{1}{x}\right)^x}\right),\ x\,(\ln(x)-1)+\frac{\ln(2)}{2}+\frac{\ln(\pi)}{2}-\frac{\ln(x)}{2}$$

$$+\frac{1}{12\,x}+O\left(\frac{1}{x^3}\right)$$

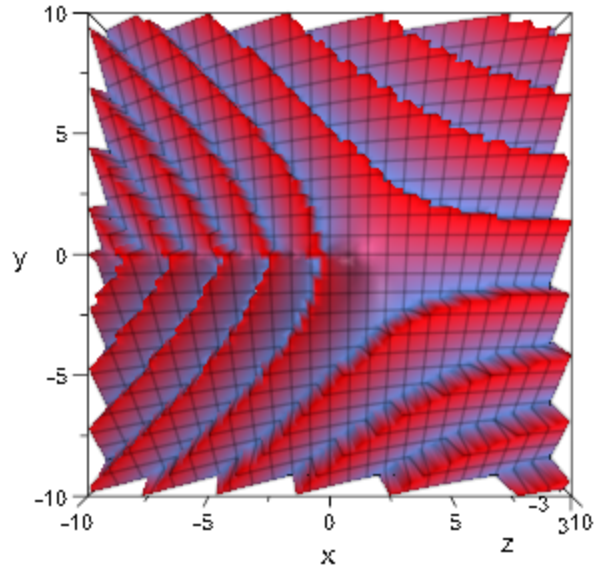
where

$$\sigma_1=\sqrt{2}\,\sqrt{\pi}\,e^{-x}$$

Example 4

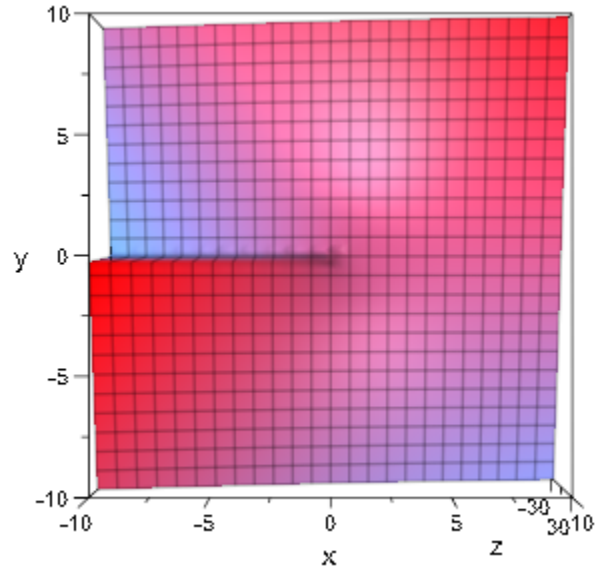
The logarithm function `ln` has a branch cut along the negative real semi axis, where the values jump by $2\,\pi\,i$ when crossing the cut. In the following plot of the imaginary part of the logarithm of the gamma function the lines in the complex z plane with $\Im(\Gamma(z)) = 0$ and $\Re(\Gamma(z)) \le 0$ are clearly visible as discontinuities:

```
plotfunc3d(Im(ln(gamma(x + I*y))), x = -10 .. 10, y = -10 .. 10,
           Submesh = [2, 2], CameraDirection = [0, -1, 1000]):
```



The function `lngamma(z)`, however, adds suitable integer multiples of $2\,\pi\,i$ to `ln(gamma(z))` making the function analytic throughout the complex plane with a branch cut along the negative real semi axis:

```
plotfunc3d(Im(lngamma(x + I*y)), x = -10 .. 10, y = -10 .. 10,
           Submesh = [2, 2], CameraDirection = [0, -1, 1000]):
```



Parameters

x

An arithmetical expression

Return Values

Arithmetical expression or a floating-point interval.

Overloaded By

x

See Also

MuPAD Functions

`beta` | `binomial` | `fact` | `gamma` | `harmonic` | `igamma` | `pochhammer` | `psi`

gcd

Greatest common divisor of polynomials

Syntax

```
gcd(p, q, …)
```

```
gcd(f, g, …)
```

Description

`gcd(p, q, ...)` returns the greatest common divisor of the polynomials *p*, *q*, … The coefficient ring of the polynomials may either be the integers or the rational numbers, `Expr`, a residue class ring `IntMod(n)` with a prime number `n`, or a domain.

All polynomials must have the same indeterminates and the same coefficient ring.

Polynomial expressions are converted to polynomials. See `poly` for details.

The return value is of the same type as the input polynomials, i.e., either a polynomial of type `DOM_POLY` or a polynomial expression.

`gcd` returns 0 if all arguments are 0, or if no argument is given. If at least one of the arguments is - 1 or 1, then `gcd` returns 1.

Use `igcd` if all arguments are known to be integers, since it is much faster than `gcd`.

Examples

Example 1

The greatest common divisor of two polynomial expressions can be computed as follows:

```
gcd(6*x^3 + 9*x^2*y^2, 2*x + 2*x*y + 3*y^2 + 3*y^3)
```

$$3\,y^2 + 2\,x$$

```
f := (x - sqrt(2))*(x^2 + sqrt(3)*x-1):
g := (x - sqrt(2))*(x - sqrt(3)):
gcd(f, g)
```

$$x - \sqrt{2}$$

One may also choose polynomials as arguments:

```
p := poly(2*x^2 - 4*x*y - 2*x + 4*y, [x, y], IntMod(17)):
q := poly(x^2*y - 2*x*y^2, [x, y], IntMod(17)):
gcd(p, q)
```

$$\mathrm{poly}(x - 2\,y, [x, y], \mathrm{IntMod}(17))$$

```
delete f, g, p, q:
```

Parameters

`p, q, …`

polynomials of type `DOM_POLY`

`f, g, …`

polynomial expressions

Return Values

Polynomial or a polynomial expression.

Overloaded By

f, g, p, q

Algorithms

If the arguments are polynomials with coefficients from a domain, then the domain must have the methods `"gcd"` and `"_divide"`. The method `"gcd"` must return the greatest common divisor of any number of domain elements. The method `"_divide"` must divide two domain elements. If domain elements cannot be divided, this method must return `FAIL`.

See Also

MuPAD Functions

`content` | `div` | `divide` | `factor` | `gcdex` | `icontent` | `ifactor` | `igcd` | `igcdex` | `ilcm` | `lcm` | `mod` | `poly`

gcdex

Extended Euclidean algorithm for polynomials

Syntax

```
gcdex(p, q, <x>)
```

```
gcdex(f, g, x)
```

Description

`gcdex(p, q, x)` regards `p` and `q` as univariate polynomials in `x` and returns their greatest common divisor as a linear combination of `p` and `q`.

`gcdex(p, q, x)` returns a sequence g, s, t with three elements, where the polynomial g is the greatest common divisor of p and q. The polynomials s and t satisfy $g = s\,p + t\,q$ and $deg(s) < deg(q)$, $deg(t) < deg(p)$. These data are computed by the extended Euclidean algorithm.

`gcdex` only processes univariate polynomials:

- If the indeterminate `x` is specified, the input polynomials are regarded as univariate polynomials in `x`.
- If no indeterminate is specified, the indeterminate of the polynomials is searched for internally. An error occurs if more than one indeterminate is found.

Note that `x` must be specified if polynomial expressions are used on input.

Polynomial expressions are converted to polynomials. See `poly` for details. `FAIL` is returned if an argument cannot be converted to a polynomial.

The returned polynomials are polynomial expressions if the input consists of polynomial expressions. Otherwise, polynomials of type `DOM_POLY` are returned.

The coefficient ring of the polynomials must provide the method `"_divide"`. This method must return `FAIL` if domain elements cannot be divided.

Note: If the coefficient domain of the polynomial is not a field, then it may not be possible to represent a greatest common divisor as a linear combination of the input polynomials. In such a case, an error is raised.

Examples

Example 1

The greatest common divisor of two univariate polynomials in extended form can be computed as follows:

```
gcdex(poly(x^3 + 1), poly(x^2 + 2*x + 1))
```

$$\mathrm{poly}(x+1,[x]),\ \mathrm{poly}\left(\frac{1}{3},[x]\right),\ \mathrm{poly}\left(-\frac{x}{3}+\frac{2}{3},[x]\right)$$

For multivariate polynomials, an indeterminate must be specified:

```
gcdex(poly(x^2*y), poly(x + y), x)
```

$$\mathrm{poly}(1,[x]),\ \mathrm{poly}\left(\frac{1}{y^3},[x]\right),\ \mathrm{poly}\left(\left(-\frac{1}{y^2}\right)x+\frac{1}{y},[x]\right)$$

```
gcdex(poly(x^2*y), poly(x + y), y)
```

$$\mathrm{poly}(1,[y]),\ \mathrm{poly}\left(-\frac{1}{x^3},[y]\right),\ \mathrm{poly}\left(\frac{1}{x},[y]\right)$$

```
gcdex(x^3 + a, x^2 + 1, x)
```

$$1,\ \frac{a}{a^2+1}+\frac{x}{a^2+1},\ \frac{1}{a^2+1}-\frac{x^2}{a^2+1}-\frac{a\,x}{a^2+1}$$

Parameters

p, q

polynomials of type DOM_POLY

f, g

polynomial expressions

x

An indeterminate: an identifier or an indexed identifier

Return Values

Sequence of three polynomials, or a sequence of three polynomial expressions, or FAIL.

Overloaded By

p, q

See Also

MuPAD Functions
div | divide | factor | gcd | ifactor | igcd | igcdex | ilcm | lcm | mod | poly

genident

Create an unused identifier

Syntax

```
genident()
```

```
genident(S)
```

Description

`genident()` creates an identifier not used before in the current session.

`genident()` creates an identifier with a name of the form `Xi`, where `i` is a positive integer. It is guaranteed that the returned identifier has not been used before in the current MuPAD session.

If a string`S` is given as argument, then `genident` returns an identifier with a name of the form `Si`, where `i` is a positive integer.

The returned identifier does not have a value.

Examples

Example 1

We create three new identifiers. The second identifier has a different prefix:

```
genident(), genident("Y"), genident()
```

```
X1, Y1, X2
```

In the next example, we assign a value to the identifier `X4`. Then the next two calls to `genident` skip the name `X4`:

```
X4 := 5:
genident(), genident()
```

```
X3, X5
```

Parameters

s

A character string

Return Values

identifier.

See Also

MuPAD Functions

`delete` | `hold`

genpoly

Create a polynomial using b-adic expansion

Syntax

```
genpoly(n, b, x)
```

Description

`genpoly(n, b, x)` creates a polynomial `p` in the variable `x` from the `b`-adic expansion of `n`, such that $p(b) = n$. The integer coefficients of the resulting polynomial are greater than $-\frac{b}{2}$ and less than or equal to $\frac{b}{2}$.

The b-adic expansion of an integer n is defined by $n = \sum_{i=0}^{m} c_i\, b^i$, such that the c_i are symmetric remainders modulo b, i.e., $-\frac{b}{2} < c_i \le \frac{b}{2}$ for all i (see `mods`). From this expansion the polynomial $p = \sum_{i=0}^{m} c_i\, x^i$ is created. The polynomial is defined over the coefficient ring `Expr`.

If the first argument of `genpoly` is a (multivariate) polynomial, then it must be defined over the coefficient ring `Expr` and must have only integer coefficients. The third argument `x` must not be a variable of the polynomial. In this case each integer coefficient is converted into a polynomial in `x` as described above. The result is a polynomial in the variable `x`, followed by the variables of the given polynomial. (`x` is the main variable of the returned polynomial.)

The first argument `n` may also be a polynomial expression. In this case, it is converted into a polynomial using `poly`, then `genpoly` is applied as described above, and the result is again converted into a polynomial expression.

If the first argument is an integer or a polynomial, then the result is a polynomial of domain type `DOM_POLY`; otherwise it is a polynomial expression.

Examples

Example 1

We create a polynomial `p` in the indeterminate `x` such that `p(7) = 15`. The coefficients of `p` are between `-3` and `3`:

```
p := genpoly(15, 7, x)
```

$$\mathrm{poly}(2x+1, [x])$$

```
p(7)
```

$$15$$

Here is an example with a polynomial expression as input:

```
p := genpoly(15*y^2 - 6*y + 3*z, 7, x)
```

$$y+3z-xy+2xy^2+y^2$$

The return value has the same type as the first argument:

```
p := genpoly(poly(15*y^2 + 8*z, [y, z]), 7, x)
```

$$\mathrm{poly}(2xy^2+xz+y^2+z, [x, y, z])$$

We check the result:

```
p(7, y, z)
```

$$15y^2+8z$$

Parameters

n

An integer, a polynomial of type `DOM_POLY`, or a polynomial expression

b

An integer greater than 1

x

The indeterminate: an identifier

Return Values

polynomial if the first argument is a polynomial or an integer. Otherwise, a polynomial expression.

See Also

MuPAD Functions

`genident` | `indets` | `int2text` | `interpolate` | `mods` | `numlib::g_adic` | `poly` | `text2int`

getlasterror

Retrieve the last error number and text

Syntax

```
getlasterror()
```

Description

`getlasterror()` returns the last error that occurred in the current MuPAD session, as a list of the error number and the error string.

After an error has occurred (whether visible or caught by `traperror`), `getlasterror` will return both the error number (as returned by `traperror`) and the error string.

In a MuPAD session where no errors occurred, `getlasterror` returns the list `[0, ""]`. This is also true after a call to `reset()`.

Note: Note that the MuPAD library uses `traperror` itself and that `getlasterror()` may return errors that have been caught and properly handled by the library already. You should not use `getlasterror` to detect errors, use the return value of `traperror` instead!

Examples

Example 1

In a fresh session, `getlasterror` returns a list indicating “no errors yet”:

```
getlasterror()
```

```
[0, ""]
```

After an error has been thrown, `getlasterror` returns the corresponding number and string:

```
ln(0)
```

```
Error: Singularity. [ln]
```

```
getlasterror()
```

```
[1028, "Error: Singularity. [ln]"]
```

This includes errors not displayed because of `traperror`:

```
traperror(solve(a, b, c))
```

```
1126
```

```
getlasterror()
```

```
[1126, "Error: The argument number 3 is invalid.  Evaluating: solvelib::getOptions"]
```

Return Values

List of an integer and a string

See Also

MuPAD Functions

`error` | `lasterror` | `traperror`

getpid

Process ID of the running kernel

Syntax

```
getpid()
```

Description

`getpid()` returns the process ID of the running MuPAD kernel.

The process ID may be useful for generating names for temporary files by appending it to a file basename.

Examples

Example 1

Querying the process ID of the running kernel may produce a result like this:

```
getpid()
```

```
      16184
```

Return Values

Nonnegative integer.

See Also

MuPAD Functions

`sysname | system`

getprop

Query properties of expressions

Syntax

```
getprop(f)
```

```
getprop()
```

Description

`getprop(f)` returns a set containing all possible values of the expression `f`.

The property mechanism helps to simplify expressions involving identifiers that carry "mathematical assumptions". The function `assume` allows to set basic assumptions such as 'x is a real number' or 'x is an odd integer', say. Arithmetical expressions involving `x` may inherit such properties. E.g., '`1 + x^2` is positive' if 'x is a real number'.

`getprop(f)` examines the assumptions of all identifiers in the expression `f` and derives a superset of all values of `f`.

Only basic mathematical properties can be represented with the available properties. Therefore, `getprop` performs certain simplifications during the derivation of a property for an expression. Thus it may happen that `getprop` derives a too large set.

`getprop` only shows a mathematical (super-)set of all possible values in respect to the assumptions. The command `property::showprops` displays a list of all valid assumptions for a special identifier.

Examples

Example 1

If `x` is a real number, then `x^2 + 1` must be positive:

```
assume(x, Type::Real):
```

```
getprop(x^2 + 1)
```

$$[1, \infty)$$

If `x` represents a number in the interval `[1, infinity[`, the expression `1 - x` has the following property:

```
assume(x, Type::Interval([1], infinity)):
getprop(1 - x)
```

$$(-\infty, 0]$$

```
unassume(x):
```

Example 2

An expression returns the superset `C_` or a set if it is constant, or if no properties are attached to the identifiers involved:

```
getprop(x), getprop(x + 2*y), getprop(sin(3))
```

$$\mathbb{C},\ \mathbb{C},\ \{\sin(3)\}$$

Example 3

The functions `abs`, `Re`, and `Im` have a “minimal property”: they produce real values. In fact, `abs` produces nonnegative real values:

```
delete x:
getprop(abs(x)), getprop(Re(x)), getprop(Im(x))
```

$$[0, \infty),\ \mathbb{R},\ \mathbb{R}$$

Parameters

`f`

An arithmetical expression

Return Values

`getprop(f)` returns a (super-)set containig all possible values of the expression `f`.

See Also

MuPAD Functions

`assume` | `is` | `property::hasprop` | `property::showprops` | `Type::Property` | `unassume`

gradient

Vector gradient

Syntax

```
gradient(f, x)
```

```
gradient(f, x, ogCoord, <c>)
```

Description

`gradient(f, x)` computes the vector gradient of the scalar function $f(\vec{x})$ with respect to $\vec{x}$ in Cartesian coordinates. This is the vector $\mathrm{grad}(f) = \left(\frac{\partial}{\partial x_1} f, \ldots, \frac{\partial}{\partial x_n} f\right)$.

`ogCoord` can be the name of a three-dimensional orthogonal coordinate system predefined in the table `linalg::ogCoordTab`. See “Example 2” on page 1-894.

Alternatively, `ogCoord` can be a list of vector of algebraic expressions representing the scale factors of the coordinate system. See example “Example 3” on page 1-894. For details, see the description of the `Scales` option on the `linalg::ogCoordTab` page.

Examples

Example 1

Compute the vector gradient of the scalar function $f(x, y) = x^2 + y$ in Cartesian coordinates:

```
delete x, y:
gradient(x^2 + y, [x, y])
```

$$\begin{pmatrix} 2x \\ 1 \end{pmatrix}$$

Example 2

Compute the gradient of the function $f(r, \phi, z) = r\,cos(\phi)\,z$ $(0 \le \phi < 2\pi)$ in cylindrical coordinates:

```
delete r, z, phi:
gradient(r*cos(phi)*z, [r, phi, z], Cylindrical)
```

$$\begin{pmatrix} z\cos(\mathrm{phi}) \\ -z\sin(\mathrm{phi}) \\ r\cos(\mathrm{phi}) \end{pmatrix}$$

Example 3

Compute the gradient of the function $f(r, \phi, \theta) = r\,sin(\phi)\,cos(\theta)$ in spherical coordinates given by

$$\vec{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} r\sin(\theta)\cos(\phi) \\ r\sin(\theta)\sin(\phi) \\ r\cos(\theta) \end{pmatrix}$$

with $0 \le \theta \le \pi$, $0 \le \phi < 2\,\pi$.

The vectors

$$\vec{e}_r = \frac{\frac{\partial}{\partial r}\vec{x}}{\left|\frac{\partial}{\partial r}\vec{x}\right|} = \begin{pmatrix} \sin(\theta)\cos(\phi) \\ \sin(\theta)\sin(\phi) \\ \cos(\theta) \end{pmatrix}, \quad \vec{e}_\theta = \frac{\frac{\partial}{\partial \theta}\vec{x}}{\left|\frac{\partial}{\partial \theta}\vec{x}\right|} = \begin{pmatrix} \cos(\theta)\cos(\phi) \\ \cos(\theta)\sin(\phi) \\ -\sin(\theta) \end{pmatrix},$$

$$\vec{e}_\phi = \frac{\frac{\partial}{\partial \phi}\vec{x}}{\left|\frac{\partial}{\partial \phi}\vec{x}\right|} = \begin{pmatrix} -\sin(\phi) \\ \cos(\phi) \\ 0 \end{pmatrix}$$

form an orthogonal system in spherical coordinates.

The scaling factors of the corresponding coordinate transformation (see `linalg::ogCoordTab`) are: $g_1 = \left|\vec{e}_r\right| = 1,\ g_2 = \left|\vec{e}_\theta\right| = r,\ g_3 = \left|\vec{e}_\phi\right| = r\sin(\theta)$, which we use in the following example to compute the gradient of the function *f* in spherical coordinates:

```
delete r, Theta, phi:
gradient(r*sin(phi)*cos(Theta), [r, Theta, phi],
                            [1, r, r*sin(Theta)])
```

$$\begin{pmatrix} \cos(\text{Theta})\ \sin(\text{phi}) \\ -\sin(\text{Theta})\ \sin(\text{phi}) \\ \frac{\cos(\text{Theta})\ \cos(\text{phi})}{\sin(\text{Theta})} \end{pmatrix}$$

The spherical coordinates are already defined in `linalg::ogCoordTab`. The last result can also be achieved with the input `gradient(r*sin(phi)*cos(Theta), [r, Theta, phi], Spherical[RightHanded])`:

```
gradient(r*sin(phi)*cos(Theta), [r, Theta, phi],
                          Spherical[RightHanded])
```

$$\begin{pmatrix} \cos(\text{Theta})\ \sin(\text{phi}) \\ -\sin(\text{Theta})\ \sin(\text{phi}) \\ \frac{\cos(\text{Theta})\ \cos(\text{phi})}{\sin(\text{Theta})} \end{pmatrix}$$

Parameters

`f`

An arithmetical expression in the variables given in `x`

`x`

A list of (indexed) identifiers

ogCoord

The name of a 3 dimensional orthogonal coordinate system predefined in the table `linalg::ogCoordTab`, or a list of algebraic expressions representing the scale factors of an orthogonal coordinate system.

c

The parameter of the coordinate systems EllipticCylindrical and Torus, respectively: an arithmetical expression. The default value is `c` = 1.

Return Values

Column vector of the domain `Dom::Matrix()`.

See Also

MuPAD Functions

`curl` | `divergence` | `laplacian` | `linalg::ogCoordTab` | `potential` | `vectorPotential`

ground

Ground term (constant coefficient) of a polynomial

Syntax

```
ground(p)
```

```
ground(f)
```

```
ground(f, vars)
```

Description

`ground(p)` returns the constant coefficient $p(0, 0, \ldots)$ of the polynomial p.

The first argument can either be a polynomial expression, or a polynomial generated by `poly`, or an element of some polynomial domain overloading `ground`.

If the first argument `f` is not element of a polynomial domain, then `ground` converts the expression to a polynomial via `poly(f)`. If a list of indeterminates is specified, then the polynomial `poly(f, vars)` is considered.

The constant coefficient is returned as an arithmetical expression.

The result of `ground` is not fully evaluated. Evaluation can be enforced by the function `eval`. Cf. “Example 2” on page 1-898.

`ground` returns `FAIL` if `f` cannot be converted to a polynomial in the specified indeterminates. Cf. “Example 3” on page 1-898.

Examples

Example 1

We demonstrate how the indeterminates influence the result:

```
f := 2*x^2 + 3*y + 1:
```

```
ground(f), ground(f, [x]), ground(f, [y]),
ground(poly(f)), ground(poly(f, [x])), ground(poly(f, [y]))
```

$$1,\ 3\,y+1,\ 2\,x^2+1,\ 1,\ 3\,y+1,\ 2\,x^2+1$$

The result is the evaluation at the origin:

```
subs(f, x = 0, y = 0), subs(f, x = 0), subs(f, y = 0)
```

$$1,\ 3\,y+1,\ 2\,x^2+1$$

Note the difference between `ground` and `tcoeff`:

```
g := 2*x^2 + 3*y:
ground(g), ground(g, [x]);
tcoeff(g), tcoeff(g, [x]);
```

$$0,\ 3\,y$$

$$3,\ 3\,y$$

```
delete f, g:
```

Example 2

The result of `ground` is not fully evaluated:

```
p := poly(27*x^2 + a, [x]): a := 5:
ground(p), eval(ground(p))
```

$$a,\ 5$$

```
delete p, a:
```

Example 3

The following expression is syntactically not a polynomial expression, and `ground` returns `FAIL`:

```
f := (x^2 - 1)/(x - 1): ground(f)
```

```
FAIL
```

After cancellation via `normal`, `ground` can compute the constant coefficient:

```
ground(normal(f))
```

```
1
```

```
delete f:
```

Parameters

`p`

A polynomial of type `DOM_POLY`

`f`

A polynomial expression

`vars`

A list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

Return Values

Element of the coefficient ring of `p`, an arithmetical expression, or `FAIL`.

Overloaded By

f, p

See Also

MuPAD Functions

`coeff` | `collect` | `degree` | `degreevec` | `lcoeff` | `ldegree` | `lmonomial` | `lterm` | `monomials` | `nterms` | `nthcoeff` | `nthmonomial` | `nthterm` | `poly` | `poly2list` | `tcoeff`

harmonic

Harmonic function

Syntax

```
harmonic(x)
```

Description

`harmonic(x)` = $\psi(x+1) + \mathrm{EULER}$ represents the harmonic function.

The harmonic function is defined for all complex arguments `x` apart from the singular points - 1, - 2, … (first order poles).

For positive integers `x` not larger than the value `Pref::autoExpansionLimit`(), the harmonic function procudes the harmonic number $\mathrm{harmonic}(x) = \sum_{k=1}^{x} \frac{1}{k}$. Use `expand(harmonic(x))` to compute an explicit result for integers `x` larger than `Pref::autoExpansionLimit`().

If `x` is a floating-point value, then a floating point value is returned.

Simplifcations are implemented for rational numbers x with $|x| \le$ *Pref*::*autoExpansionLimit*(). In particular, if x = `numer(x)/k` with denominators k = 1, 2, 3, 4, or 6, then an explicit result is computed and returned. For other rational numbers the functional equation $\mathrm{harmonic}(x+1) = \mathrm{harmonic}(x) + \frac{1}{x}$ is used to obtain a result with an argument `x` from the interval $(0, 1]$.

For rational numbers x with $|x| >$ *Pref*::*autoExpansionLimit*(), these simplifications can be enforced via `expand`.

Some explicit formulas are implemented including

$$\mathrm{harmonic}(0) = 0,\ \mathrm{harmonic}\left(\frac{1}{2}\right) = 2 - 2\ln(2),$$

$$\mathrm{harmonic}\left(\frac{1}{3}\right) = 3 - \frac{3\ln(3)}{2} - \frac{\sqrt{3}}{6}\pi,\ \mathrm{harmonic}\left(\frac{2}{3}\right) = \frac{3}{2} - \frac{3\ln(3)}{2} + \frac{\sqrt{3}}{6}\pi,$$

$$\text{harmonic}\left(\frac{1}{4}\right) = 4 - 3\ln(2) - \frac{\pi}{2},\ \text{harmonic}\left(\frac{3}{4}\right) = \frac{4}{3} - 3\ln(2) + \frac{\pi}{2},$$

$$\text{harmonic}\left(\frac{1}{6}\right) = 6 - 2\ln(2) - \frac{3\ln(3)}{2} - \frac{\sqrt{3}}{2}\pi,\ \text{harmonic}\left(\frac{5}{6}\right) = \frac{6}{5} - 2\ln(2) - \frac{3\ln(3)}{2} + \frac{\sqrt{3}}{2}\pi,$$

$$\text{harmonic}(1) = 1.$$

The special value $harmonic(\infty) = \infty$ is implemented.

For all other arguments, a symbolic function call of `harmonic` is returned.

The `expand` attribute uses the functional equation $\text{harmonic}(x+1) = \text{harmonic}(x) + \frac{1}{x}$, the reflection rule $\text{harmonic}(-x) = \text{harmonic}(x) - \frac{1}{x} + \pi\cot(\pi x)$ and the Gauß multiplication formula for $harmonic(k\,x)$ with some integer k to rewrite `harmonic(x)`. See "Example 3" on page 1-902 and "Example 4" on page 1-903.

Environment Interactions

When called with a floating-point value `x`, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
harmonic(3), harmonic(10), harmonic(3/2), harmonic(25/7)
```

$$\frac{11}{6},\ \frac{7381}{2520},\ \frac{8}{3} - 2\ln(2),\ \text{harmonic}\left(\frac{4}{7}\right) + \frac{6461}{4950}$$

```
harmonic(x + sqrt(2)), harmonic(infinity)
```

$$\text{harmonic}(x + \sqrt{2}),\ \infty$$

Floating point values are computed for floating-point arguments:

```
harmonic(-5.2), harmonic(27.0), harmonic(2.0 + 3.0*I)
```

$$6.450681124,\ 3.891456753,\ 1.939042529 + 0.8733604498\ i$$

Example 2

`harmonic` is singular for negative integers:

```
harmonic(-2)
```

```
Error: Singularity. [harmonic]
```

Example 3

For positive integers and rational numbers x with denominators 2, 3, 4 and 6, respectively, the result is expressed in terms of PI and `ln`, if $|x| < 500$:

```
harmonic(-5/2)
```

$$\frac{8}{3} - 2\ln(2)$$

```
harmonic(13/3)
```

$$\frac{8571}{1820} - \frac{\pi\sqrt{3}}{6} - \frac{3\ln(3)}{2}$$

```
harmonic(101/6)
```

$$\frac{\pi\sqrt{3}}{2} - \frac{3\ln(3)}{2} - 2\ln(2) + \frac{257690034865853321191998}{68836669705906834284553}$$

For larger arguments, the `expand` attribute can be used to obtain such expressions:

```
harmonic(1001)
```

```
harmonic(1001)
```

$$\text{harmonic}(1001)$$

```
expand(%)
```

```
5337003...5042517 / 7128865...3520000
```

Example 4

The functions `diff`, `expand`, `float`, `limit`, and `series` handle expressions involving `harmonic`:

```
diff(harmonic(x^2 + 1), x), float(ln(3 + harmonic(sqrt(PI))))
```

$$2\,x\,\psi'(x^2+2),\ 1.482943321$$

```
expand(harmonic(2*x + 3))
```

$$\frac{\text{harmonic}\left(x+\frac{1}{2}\right)}{2}+\ln(2)+\frac{\text{harmonic}(x)}{2}-\frac{1}{2\left(x+\frac{1}{2}\right)}+\frac{1}{2x+1}+\frac{1}{2x+2}+\frac{1}{2x+3}$$

```
limit((x + 1)*harmonic(x), x = -1), limit(harmonic(x), x = infinity)
```

$$-1,\ \infty$$

```
series(harmonic(x), x = 0)
```

$$\frac{\pi^2 x}{6}-x^2\,\zeta(3)+\frac{\pi^4 x^3}{90}-x^4\,\zeta(5)+\frac{\pi^6 x^5}{945}-x^6\,\zeta(7)+O(x^7)$$

```
series(harmonic(x), x = infinity, 3)
```

$$\text{EULER}+\ln(x)+\frac{1}{2x}-\frac{1}{12x^2}+O\left(\frac{1}{x^3}\right)$$

Parameters

x

An arithmetical expression

Return Values

Arithmetical expression.

Overloaded By

x

See Also

MuPAD Functions
`beta` | `binomial` | `fact` | `gamma` | `lngamma` | `zeta`

has

Check if an object occurs in another object

Syntax

```
has(object1, object2)
```

```
has(object1, l)
```

Description

`has(object1, object2)` checks, whether `object2` occurs syntactically in `object1`.

`has` is a fast test for the existence of sub-objects or subexpressions. It works syntactically, i.e., mathematically equivalent objects are considered to be equal only if they are syntactically identical. See “Example 2” on page 1-906.

If `object1` is an expression, then `has(object1, object2)` tests whether `object1` contains `object2` as a subexpression. Only complete subexpressions and objects occurring in the 0th operand of a subexpression are found (see “Example 1” on page 1-906).

If `object1` is a container, then `has` checks whether `object2` occurs in an entry of `object1`. See “Example 5” on page 1-908.

In this context, a floating-point interval is considered a container for (an infinite number of) complex numbers and `has` checks whether a given number is *inside* the interval. See “Example 4” on page 1-907.

If the second argument is a list or a set `l`, then `has` returns `TRUE` if at least one of the elements in `l` occurs in `object1` (see “Example 3” on page 1-907). In particular, if `l` is the empty list or the empty set, then the return value is `FALSE`.

If `object1` is an element of a domain with a `"has"` slot, then the slot routine is called with the same arguments, and its result is returned. If the domain does not have such a slot, then `FALSE` will be returned. See “Example 7” on page 1-909.

If has is called with a list or set as second argument, then the "has" slot of the domain of object1 is called for each object of the list or the set. When the first object is found that occurs in object1, the evaluation is terminated and TRUE is returned. If none of the objects occurs in object1, FALSE will be returned.

Examples

Example 1

The given expression has x as an operand:

```
has(x + y + z, x)
```

```
    TRUE
```

Note that x + y is not a complete subexpression. Only x, y, z and x + y + z are complete subexpressions:

```
has(x + y + z, x + y)
```

```
    FALSE
```

However, has also finds objects in the 0th operand of a subexpression:

```
has(x + sin(x), sin)
```

```
    TRUE
```

Every object occurs in itself:

```
has(x, x)
```

```
    TRUE
```

Example 2

has works in a purely syntactical fashion. Although the two expressions y*(x + 1) and y*x + y are mathematically equivalent, they differ syntactically:

```
has(sin(y*(x + 1)), y*x + y),
has(sin(y*(x + 1)), y*(x + 1))
```

```
FALSE, TRUE
```

Complex numbers are not regarded as atomic objects:

```
has(2 + 5*I, 2), has(2 + 5*I, 5), has(2 + 5*I, I)
```

```
TRUE, TRUE, TRUE
```

In contrast, rational numbers are considered to be atomic:

```
has(2/3*x, 2), has(2/3*x, 3), has(2/3*x, 2/3)
```

```
FALSE, FALSE, TRUE
```

Example 3

If the second argument is a list or a set, has checks whether one of the entries occurs in the first argument:

```
has((x + y)*z, [x, t])
```

```
TRUE
```

0th operands of subexpressions are checked as well:

```
has((a + b)*c, {_plus, _mult})
```

```
TRUE
```

Example 4

On floating-point intervals, has performs a containment check, not just testing the borders:

```
has(1...3, 1)
```

```
    TRUE
```

```
has(1...3, 2.7182), has(1...3, exp(1)), has(1...3, PI)
```

```
    TRUE, TRUE, FALSE
```

```
has(1...(3+I), [2, ln(3)])
```

```
    TRUE
```

Example 5

`has` works for lists, sets, tables, arrays, and hfarrays:

```
has([sin(f(a) + 2), cos(x), 3], {f, g})
```

```
    TRUE
```

```
has({a, b, c, d, e}, {a, z})
```

```
    TRUE
```

```
has(array(1..2, 1..2, [[1, 2], [3, 4]]), 2)
```

```
    TRUE
```

For an array `A`, the command `has(A,NIL)` checks whether the array has any uninitialized entries:

```
has(array(1..2, 1 = x), NIL),
has(array(1..2, [2, 3]), NIL)
```

```
    TRUE, FALSE
```

For tables, `has` checks indices, entries, as well as the internal operands of a table, given by equations of the form `index=entry`:

```
T := table(a = 1, b = 2, c = 3):
```

```
has(T, a), has(T, 2), has(T, b = 2)
```

TRUE, TRUE, TRUE

Example 6

has works syntactically. Although the variable x does not occur mathematically in the constant polynomial p in the following example, the identifier x occurs syntactically in p, namely, in the second operand:

```
delete x: p := poly(1, [x]):
has(p, x)
```

TRUE

Example 7

The second argument may be an arbitrary MuPAD object, even from a user-defined domain:

```
T := newDomain("T"):
e := new(T, 1, 2);
f := [e, 3];
```

new(*T*, 1, 2)

[new(*T*, 1, 2), 3]

```
has(f, e), has(f, new(T, 1))
```

TRUE, FALSE

If the first argument of has belongs to a domain without a "has" slot, then has always returns FALSE:

```
has(e, 1)
```

FALSE

Users can overload`has` for their own domains. For illustration, we supply the domain `T` with a `"has"` slot, which puts the internal operands of its first argument in a list and calls `has` for the list:

```
T::has := (object1, object2) -> has([extop(object1)], object2):
```

If we now call `has` with the object `e` of domain type `T`, the slot routine `T::has` is invoked:

```
has(e, 1), has(e, 3)
```

```
TRUE, FALSE
```

The slot routine is also called if an object of domain type `T` occurs syntactically in the first argument:

```
has(f, 1), has(f, 3)
```

```
TRUE, TRUE
```

Parameters

`object1, object2`

Arbitrary MuPAD objects

`l`

A list or a set

Return Values

Either `TRUE` or `FALSE`

Overloaded By

object1

See Also

MuPAD Functions

`_in` | `_index` | `contains` | `hastype` | `op` | `subs` | `subsex`

hastype

Test if an object of a specified type occurs in another object

Syntax

```
hastype(object, T, <inspect>)
```

Description

`hastype(object, T)` tests if an object of type `T` occurs syntactically in `object`.

`hastype(object, T)` tests if a sub-object `s` of type `T` occurs in `object`, i.e., such that `testtype(s, T)` returns `TRUE`.

The type specifier `T` may be either a domain type such as `DOM_INT`, `DOM_EXPR` etc., a string as returned by the function `type`, or a Type object. The latter are probably the most useful predefined values for the argument `T`.

If `T` is not a valid type specifier, then `hastype` returns `FALSE`.

See “Example 1” on page 1-913.

If `object` is an expression, then `hastype(object, T)` tests whether `object` contains a subexpression of type `T`; see “Example 1” on page 1-913.

If `object` is a container, then `hastype` checks whether a sub-object of type `T` occurs in an entry of `object`; see “Example 4” on page 1-915.

If the second argument is a list or a set, `hastype` checks whether a sub-object of one of the types in `T` occurs in `object`. See “Example 1” on page 1-913.

`hastype` works in a recursive fashion and descends into the following objects: expressions, arrays, hfarrays, lists, sets, and tables. See “Example 4” on page 1-915. `hastype` does not step into the other basic domains, such as rational numbers, complex numbers, polynomials, or procedures. See “Example 2” on page 1-914.

If the third argument `inspect` is present, then `hastype` also steps recursively into sub-objects of the domain types given in `inspect`. See “Example 2” on page 1-914.

Note: `hastype` looks only for sub-objects that are syntactically of type `T`. Properties of identifiers set via `assume` are not taken into account; see “Example 4” on page 1-915.

Examples

Example 1

In this example, we first test if a given expression has a subexpression of type `DOM_FLOAT`:

```
hastype(1.0 + x, DOM_FLOAT)
```

```
TRUE
```

```
hastype(1 + x, DOM_FLOAT)
```

```
FALSE
```

We may also test if an expressions contains a subexpression of one of the two types `DOM_FLOAT` or `DOM_INT`:

```
hastype(1.0 + x, {DOM_FLOAT, DOM_INT})
```

```
TRUE
```

While the first of following two tests returns `FALSE`, since `tan` is not a valid type specifier, the second test yields `TRUE`, since the given expression contains a subexpression of type `"tan"`:

```
hastype(sin(tan(x) + 1/exp(1 - x)), tan),
hastype(sin(tan(x) + 1/exp(1 - x)), "tan")
```

```
FALSE, TRUE
```

You can also use type specifiers from the Type library:

```
hastype([-1, 10, -5, 2*I], Type::PosInt)
```

```
TRUE
```

Example 2

We demonstrate the use of the optional third argument. We want to check if a procedure contains a subexpression of type `"float"`. By default, `hastype` does not descend recursively into a procedure:

```
f := x -> float(x) + 3.0:
hastype(f, "float")
```

```
FALSE
```

You can use the third argument to request the inspection of procedures explicitly:

```
hastype(f, "float", {DOM_PROC})
```

```
TRUE
```

Also, by default, `hastype` does not descend recursively into the basic domains `DOM_COMPLEX` and `DOM_RAT`:

```
hastype(1 + I, DOM_INT), hastype(2/3,  DOM_INT)
```

```
FALSE, FALSE
```

In order to inspect these data types, one has to use the third argument:

```
hastype(1 + I, DOM_INT, {DOM_COMPLEX}),
hastype(2/3,  DOM_INT, {DOM_RAT})
```

```
TRUE, TRUE
```

Example 3

Since matrices possess a slot `enableMaprec`, `hastype` automatically inspects their entries.

```
A := matrix([[1, 1], [1, 0]]):
```

```
hastype(A, DOM_INT)
```

```
TRUE
```

It is also possible to inspect elements of other domains using the third argument. As an example let us define a permutation and ask for a subexpression of type integer:

```
G:= Dom::SymmetricGroup(4):
perm:= G([2,4, 3, 1]):
hastype(perm, DOM_INT), hastype(perm, DOM_INT, {G})
```

```
FALSE, TRUE
```

Example 4

We demonstrate how `hastype` effects on container objects. Let us first stress tables:

```
hastype(table(1 = a), DOM_INT), hastype(table(a = 1), DOM_INT)
```

```
FALSE, TRUE
```

As shown, `hastype` does not inspect the indices of a table, but checks recursively whether a sub-object of a given type occurs in an entry. This is also true for arrays, hfarrays, lists and sets:

```
hastype(array(1..4, [1, 2, 3, 4]), DOM_INT),
hastype(hfarray(1..3, [1.0, 2.0, 3.0*I]), DOM_COMPLEX),
hastype([1, 2, 3, 4], DOM_INT),
hastype({1, 2, 3, 4}, DOM_INT),
hastype([[a, [1]], b, c], DOM_INT)
```

```
TRUE, TRUE, TRUE, TRUE, TRUE
```

`hastype` can only work syntactically, i.e. properties are not taken into account:

```
assume(a,Type::Integer):
hastype([a, b], Type::Integer), hastype([a, b], DOM_INT)
```

```
FALSE, FALSE
```

```
delete a:
```

Parameters

`object`

An arbitrary MuPAD object

`T`

A type specifier, or a set or a list of type specifiers

`inspect`

A set of domain types

Return Values

Either TRUE or FALSE.

Overloaded By

object

See Also

MuPAD Functions
`domtype` | `has` | `misc::maprec` | `testtype` | `type`

heaviside

The Heaviside step function

Syntax

```
heaviside(x)
```

Description

`heaviside(x)` represents the Heaviside step function.

If the argument represents a positive real number, then 1 is returned. If the argument represents a negative real number, then 0 is returned. If the argument is zero, $\frac{1}{2}$ is returned. If the argument is a complex number of domain type `DOM_COMPLEX`, then `undefined` is returned. For all other arguments, an unevaluated function call is returned.

The derivative of `heaviside` is the delta distribution `dirac`.

To change the value of `heaviside` at the origin, use `Pref::heavisideAtOrigin`. See “Example 4” on page 1-919. Common choices for this value are `0`, `1`, and `1/2`.

Examples

Example 1

`heaviside` returns 1 or 0 for arguments representing positive or negative real numbers, respectively:

```
heaviside(-3), heaviside(-sqrt(3)), heaviside(-2.1),
heaviside(PI - exp(1)), heaviside(sqrt(3))
```

```
0, 0, 0.0, 1, 1
```

`heaviside` returns $\frac{1}{2}$ if the argument is zero:

```
heaviside(0), heaviside(0.0)
```

$$\frac{1}{2}, 0.5$$

Arguments of domain type `DOM_COMPLEX` yield `undefined`:

```
heaviside(1 + I), heaviside(2/3 + 7*I), heaviside(0.1*I)
```

$$\text{undefined}, \text{undefined}, \text{undefined}$$

An unevaluated call is returned for other arguments:

```
heaviside(x), heaviside(ln(-5)), heaviside(x + I)
```

$$\text{heaviside}(x), \text{heaviside}(\ln(5) + \pi\, i), \text{heaviside}(x + i)$$

Example 2

`heaviside` reacts to assumptions set by `assume`:

```
assume(x > 0): heaviside(x)
```

$$1$$

```
unassume(x):
```

Example 3

The derivative of `heaviside` is the delta distribution `dirac`:

```
diff(heaviside(x - 4), x)
```

$$\delta(x-4)$$

The integrator `int` handles `heaviside`:

```
int(exp(-x)*heaviside(x), x = -infinity..infinity)
```

$$1$$

We do not recommend to use `heaviside` in numerical integration. It is much more efficient to split the quadrature into pieces, each of which having a smooth integrand:

```
DIGITS := 3: numeric::int(exp(-x)*heaviside(x^2 - 2), x=-3..10)
```

$$16.2$$

```
numeric::int(exp(-x), x = -3..-2^(1/2)) +
numeric::int(exp(-x), x = 2^(1/2)..10)
```

$$16.2$$

```
delete DIGITS:
```

Example 4

`heaviside` assumes that the value of the Heaviside function at the origin is `1/2`.

```
heaviside(0)
```

$$\frac{1}{2}$$

Other common values for the Heaviside function at the origin are `0` and `1`. To change the value of `heaviside` at the origin, use `Pref::heavisideAtOrigin`. Store the previous value, so that you can restore it later.

```
oldval := Pref::heavisideAtOrigin(1):
```

Check the new value of `heaviside` at `0`.

```
heaviside(0)
```

$$1$$

Restore the previous value of `heavisideAtOrigin` using `oldval`.

```
Pref::heavisideAtOrigin(oldval):
```

Also, you can restore the default value of `heavisideAtOrigin` by specifying the input as `NIL`.

```
Pref::heavisideAtOrigin(NIL):
```

Parameters

x

An arithmetical expression

Return Values

Arithmetical expression.

Overloaded By

x

See Also

MuPAD Functions

`dirac` | `Pref::heavisideAtOrigin`

?, help

Display a help page

Syntax

```
?word
```

```
help("word")
```

Description

`help("word")` or `?word` displays a help page with information about the keyword `word`.

When you use `help("word")` interactively, you can use `?word` as a shortcut. The `?` command is not a MuPAD function. You cannot use `?` in expressions or in files. Do not enclose `word` in quotation marks, and do not terminate it with a semicolon.

Parameters

`word`

Any keyword

Return Values

Void object `null()` of type `DOM_NULL`.

See Also

MuPAD Functions
`info`

hessian

Hessian matrix of a scalar function

Syntax

```
hessian(f, x)
```

Description

`hessian(f, x)` computes the Hesse matrix (the Hessian) of the scalar function $f(\vec{x})$ in Cartesian coordinates, i.e., the square matrix of second partial derivatives of $f(\vec{x})$.

Examples

Example 1

The Hessian of the function $f(x, y, z) = x\,y + 2\,x\,z$ is the following matrix:

```
delete x, y, z:
hessian(x*y + 2*z*x, [x, y, z])
```

$$\begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 0 \\ 2 & 0 & 0 \end{pmatrix}$$

Parameters

f

An arithmetical expression (the scalar function)

x

A list of (indexed) identifiers

Return Values

Matrix of the domain `Dom::Matrix()`.

Algorithms

For a function $f: X \to \mathbb{R}$, X a subset of $\mathbb{R}^p$, the $p \times p$ matrix

$$H_f(\vec{x}) = \begin{pmatrix} \frac{\partial^2}{\partial x_1^2} f & \frac{\partial}{\partial x_1}\frac{\partial}{\partial x_2} f & \cdots & \frac{\partial}{\partial x_1}\frac{\partial}{\partial x_p} f \\ \frac{\partial}{\partial x_2}\frac{\partial}{\partial x_1} f & \frac{\partial^2}{\partial x_2^2} f & \cdots & \frac{\partial}{\partial x_2}\frac{\partial}{\partial x_p} f \\ \vdots & \vdots & & \vdots \\ \frac{\partial}{\partial x_p}\frac{\partial}{\partial x_1} f & \frac{\partial}{\partial x_p}\frac{\partial}{\partial x_2} f & \cdots & \frac{\partial^2}{\partial x_p^2} f \end{pmatrix}$$

is called the *Hesse matrix* of f.

See Also

MuPAD Functions

`diff` | `gradient` | `jacobian`

HISTORY

Maximal number of elements in the history table

Description

The environment variable HISTORY determines the maximal number of entries of the history table at interactive level.

Possible values: Nonnegative integer smaller than 2^{31}.

The commands that are entered interactively in a MuPAD session, executed in a procedure, or read from a file, as well as the resulting MuPAD outputs are stored in an internal data structure, the history table. Only the most recent entries are kept in memory.

Entries of the history table can be accessed via `history` or `last`.

The default value of HISTORY is 20; HISTORY has this value after starting or resetting the system via `reset`. Also the command `delete HISTORY` restores the default value.

Within a procedure, the maximal number of entries in the local history table of the procedure is always 3, independent of the value of HISTORY.

Examples

Example 1

In the following example, we set the value of HISTORY to 2. Afterwards, only the two most recent inputs and outputs are stored in the history table at interactive level:

```
HISTORY := 2:
a := 1: b := 2: max(a, b):
history(history() - 1), history(history())
```

$$[b := 2,\ 2],\ [\max(a,\ b),\ 2]$$

The attempt to access the third last entry in the history table leads to an error:

```
history(history() - 2)
```

```
Error: The argument is invalid. [history]
```

We use `delete` to restore the default value of `HISTORY`:

```
delete HISTORY: HISTORY
```

```
20
```

See Also

MuPAD Functions

`history` | `last`

history

Access an entry of the history table

Syntax

```
history(n)
```

```
history()
```

Description

`history(n)` returns the `n`th entry of the history table.

`history()` returns the index of the most recent entry in the history table.

The commands that are entered interactively in a MuPAD session, executed in a procedure, or read from a file, as well as the resulting MuPAD outputs are stored in an internal data structure, the history table. `history()` returns the index of the most recent entry in the history table. At interactive level, this is the number of commands that have been entered since the start of the session or the last restart.

`history(n)` returns the `n`th entry in the history table in form of a list with two elements. The first element of this list is a MuPAD command, and the second element is the result of this command returned by MuPAD. The order of the entries in the history table is such that larger indices correspond to more recent entries.

The command `last` accesses the result entries from the history table. The call `last(n)` is equivalent to `history ( history() - n + 1) [2]` at interactive level.

The environment variable `HISTORY` determines the maximal number of history entries that are stored at interactive level. The default value is `20`. Only the most recent entries are kept in memory. Thus valid arguments for `history` are all integers between `history() - HISTORY + 1` and `history()`. All other integers lead to an error message.

The result returned by `history` is not evaluated again (see example history-eval). Use the function `eval` to force a subsequent evaluation.

Commands and their results are stored in the history table even if the output is suppressed by a colon. See “Example 1” on page 1-927.

Compound statements, such as `for`, `repeat`, and `while` loops, `if` and `case` branching instructions, and procedure definitions via `proc` are stored in the history table as a whole at interactive level. See the help page of `last` for examples.

Commands appearing on the same input line lead to separate entries in the history table if they are separated by a colon or a semicolon. In contrast, a statement sequence is regarded as a single command (see “Example 3” on page 1-929).

Commands that are read from a file via `fread` or `read` are stored in the history table, and at last the `fread` or `read` command is stored in the history table (because the `fread` or `read` command is finished foremost after reading the file). However, if the option `Plain` is used, then a separate history table is in effect within the file, and the commands from the file do not appear in the history table of the enclosing context.

Note that every call of `history` modifies the history table and possibly erases the earliest history entry.

Every procedure has its own local history table. However, the entries of this table cannot be accessed via `history` (see `last`). The command `history` always refers to the history table at interactive level.

Examples

Example 1

The index of the most recent entry in the history table increases by one for each entered command, also by `history()`. Note that every command is stored in the history table, whether its output is suppressed by a colon or not:

```
history(); sqrt(1764); history(): history()
```

```
 3 42 6
```

`history ( history() )` returns a list with two elements. The first element is the last command, and the second element is the result returned by MuPAD, which is equal to `last(1)` or `%`:

```
int(2*x*exp(x^2), x);
```

```
history(history()), last(1)
```

$$e^{x^2}$$

$$\left[\int 2\,x\,e^{x^2}\,dx,\ e^{x^2}\right],\ e^{x^2}$$

The following command returns the next to last command and its result:

```
history(history() - 1)
```

$$\left[\int 2\,x\,e^{x^2}\,dx,\ e^{x^2}\right]$$

A restart cleans up the history table:

```
reset():
history()
```

```
4
```

The output of the command `history()` above depends on the number of commands in your MuPAD startup file `userinit.mu`.

Example 2

First `a` should be `0`:

```
a := 0:
a
```

$$0$$

Now `1` is assigned to `a`:

```
a := 1:
a
```

$$1$$

The command `history ( history() -2)` refers to the command `a` after assigning `0` to `a`, the return value of `history` is not the new value of `a`, because the result returned by `history` is not evaluated again:

```
history(history() - 2)
```

$$[a, 0]$$

Example 3

The following commands create two entries in the history table. The command `history ( history() -1)` returns only the last command `b:=a`, not both commands:

```
a := 0: b := a:
history(history() - 1)
```

$$[a := 0, 0]$$

If the commands are entered as a statement sequence (enclosed in `( )`), they create one entry. `history ( history() )` picks out the last command, that is, the statement sequence:

```
(a := 0; b := a;):
history(history())
```

```
[(a := 0;
b := a), 0]
```

The last input

```
type(op(%, 1))
```

"_stmtseq"

Parameters

n

A positive integer

Return Values

`history(n)` returns a list with two elements, and `history()` returns a nonnegative integer.

See Also

MuPAD Functions
`fread` | `HISTORY` | `last` | `read`

hold

Delay evaluation

Syntax

```
hold(object)
```

Description

`hold(object)` prevents the evaluation of `object`.

When a MuPAD object is entered interactively, then the system evaluates it and returns the evaluated result. When a MuPAD object is passed as an argument to a procedure, then the procedure usually evaluates the argument before processing it. *Evaluation* means that identifiers are replaced by their values and function calls are executed. `hold` is intended to prevent such an evaluation when it is undesirable.

A typical application of `hold` is when a function that can only process numerical arguments, but not symbolical ones, is to be used as an expression. See “Example 6” on page 1-936.

Another possible reason for using `hold` is efficiency. For example, if a function call `f(x, y)` with symbolic arguments is passed as argument to another function, but is known to return itself symbolically, then the possibly costly evaluation of the “inner” function call can be avoided by passing the expression `hold(f) (x, y)` as argument to the “outer” function instead. Then the arguments `x, y` are evaluated, but the call to `f` is not executed. See examples “Example 1” on page 1-932 and “Example 7” on page 1-936.

Since using `hold` may lead to strange effects, it is recommended to use it only when absolutely necessary.

`hold` only delays the evaluation of an object, but cannot completely prevent it on the long run; see “Example 5” on page 1-935.

You can use `freeze` to completely prevent the evaluation of a procedure or a function environment.

A MuPAD procedure can be declared with the option hold. This has the effect that arguments are passed to the procedure unevaluatedly. See the help page of `proc` for details.

The functions `eval` or `level` can be used to force a subsequent evaluation of an unevaluated object (see example “Example 2” on page 1-933). In procedures with `option hold`, use `context` instead.

Examples

Example 1

In the following two examples, the evaluation of a MuPAD expression is prevented using `hold`:

```
x := 2:
hold(3*0 - 1 + 2^2 + x)
```

$$3\,0 - 1 + 2^2 + x$$

```
hold(error("not really an error"))
```

error("not really an error")

Without `hold`, the results would be as follows:

```
x := 2:
3*0 - 1 + 2^2 + x
```

$$5$$

```
error("not really an error")
```

```
Error: not really an error
```

The following command prevents the evaluation of the operation `_plus`, but not the evaluation of the operands:

```
hold(_plus)(3*0, -1, 2^2, x)
```

$$0 - 1 + 4 + 2$$

Note that in the preceding example, the arguments of the function call are evaluated, because `hold` is applied only to the function `_plus`. In the following example, the argument of the function call is evaluated, despite that fact that `f` has the option hold:

```
f := proc(a)
       option hold;
     begin
       return(a + 1)
     end_proc:
x := 2:
hold(f)(x)
```

$$f(2)$$

This happens for the following reason. When `f` is evaluated, the option `hold` prevents the evaluation of the argument `x` of `f` (see the next example). However, if the evaluation of `f` is prevented by `hold`, then the option `hold` has no effect, and MuPAD evaluates the operands, but not the function call.

The following example shows the expected behavior:

```
f(x), hold(f(x))
```

$$x + 1,\ f(x)$$

The function `eval` undoes the effect of `hold`. Note that it yields quite different results, depending on how it is applied:

```
eval(f(x)), eval(hold(f)(x)), eval(hold(f(x))), eval(hold(f))(x)
```

$$3,\ 3,\ x + 1,\ x + 1$$

Example 2

Several `hold` calls can be nested to prevent subsequent evaluations:

```
x := 2:
hold(x), hold(hold(x))
```

$$x, \text{hold}(x)$$

The result of `hold ( hold(x) )` is the unevaluated operand of the outer call of `hold`, that is, `hold(x)`. Applying `eval` evaluates the result `hold(x)` and yields the unevaluated identifier `x`:

```
eval(%)
```

$$2, x$$

Another application of `eval` yields the value of `x`:

```
eval(%)
```

$$2, 2$$

```
delete x, f:
```

Example 3

The following command prevents the evaluation of the operation `_plus`, replaces it by the operation `_mult`, and then evaluates the result:

```
eval(subsop(hold(_plus)(2, 3), 0 = _mult))
```

$$6$$

Example 4

The function `domtype` evaluates its arguments:

```
x := 0:
domtype(x), domtype(sin), domtype(x + 2)
```

$$\text{DOM\_INT}, \text{DOM\_FUNC\_ENV}, \text{DOM\_INT}$$

Using `hold`, the domain type of the unevaluated objects can be determined: `x` and `sin` are identifiers, and `x + 2` is an expression:

```
domtype(hold(x)), domtype(hold(sin)), domtype(hold(x + 2))
```

DOM_IDENT, DOM_IDENT, DOM_EXPR

Example 5

`hold` prevents only one evaluation of an object, but it does not prevent evaluation at a later time. Thus using `hold` to obtain a a symbol without a value is usually not a good idea:

```
x := 2:
y := hold(x);
y
```

x

2

In this example, deleting the value of the identifier `x` makes it a symbol, and using `hold` is not necessary:

```
delete x:
y := x;
y
```

x

x

However, the best way to obtain a new symbol without a value is to use `genident`:

```
y := genident("z");
y
```

z1

```
    z1
```

```
delete y:
```

Example 6

Consider the piecewise defined function $f(x)$ that is identically zero on the negative real axis and equal to e^{-x} on the positive real axis:

```
f := x -> if x < 0 then 0 else exp(-x) end_if:
```

This function cannot be called with a symbolic argument, because the condition $x < 0$ cannot be decided:

```
f(x)
```

```
Error: Cannot evaluate to Boolean. [_less]
  Evaluating: f
```

We wish to integrate `f` numerically. However, the numerical integrator expects the function as an expression:

```
numeric::int(f(x), x = -2..2)
```

```
Error: Cannot evaluate to Boolean. [_less]
  Evaluating: f
```

The solution is to suppress premature evaluation of `f` when passing the function with a symbolic argument. Inside the numerical integrator, numerical values are substituted for x before the function is called and evaluated:

```
numeric::int(hold(f)(x), x = -2..2)
```

```
    0.8646647168
```

Example 7

The function `int` is unable to compute a closed form of the following integral and returns a symbolic `int` call:

```
int(sqrt(x)*sqrt(sqrt(x) + 1), x)
```

$$\int \sqrt{x}\,\sqrt{\sqrt{x}+1}\,dx$$

After the change of variables `sqrt(x)=t`, a closed form can be computed:

```
t := time():
f := intlib::changevar(int(sqrt(x)*sqrt(sqrt(x) + 1), x), sqrt(x) = y);
time() - t;
eval(f)
```

$$\int \frac{y\sqrt{y+1}}{\sqrt{-y^2+1}}\,dy$$

9210

$$-\frac{2\,(y+2)\sqrt{-y^2+1}}{3\sqrt{y+1}}$$

$$\int 2y^2\sqrt{y+1}\,dy$$

$$-\frac{4\,(y+1)^{3/2}\,(42y-15\,(y+1)^2+7)}{105}$$

Measuring computing times with `time` shows: Most of the time in the call to `intlib::changevar` is spent in re-evaluating the argument. This can be prevented by using `hold`:

```
t := time():
f := intlib::changevar(hold(int)(sqrt(x)*sqrt(sqrt(x) + 1), x),
                       sqrt(x) = y);
time() - t;
```

$$\int \frac{y\sqrt{y+1}}{\sqrt{-y^2+1}}\,dy$$

$$20$$

Parameters

`object`

Any MuPAD object

Return Values

Unevaluated object.

See Also

MuPAD Functions

`context` | `delete` | `eval` | `freeze` | `genident` | `indexval` | `level` | `proc` | `val`

More About

- "Prevent Evaluation"

..., hull

Convert to a floating-point interval

Syntax

```
l ... r
```

```
hull(object)
```

Description

`hull`(object) returns a floating-point interval enclosing object.

`l ... r` is equivalent to `hull(l, r)`.

`hull` converts numerical and interval expressions to numerical intervals of type `DOM_INTERVAL`. It accepts lists and sets of numerical expressions or intervals as well as numerical expressions, intervals, and set-theoretic functions of intervals and sets.

Infinities are displayed using `RD_INF` for `infinity` and `RD_NINF` for `-infinity`.

`hull` is mapped recursively to the operands of any expression given—but for subexpressions, lists and sets are not accepted. Identifiers are replaced by intervals, respecting a certain subset of properties. Cf. “Example 3” on page 1-941. Likewise, function calls and domain elements not overloading `hull` are converted to the interval representing the complex plane.

The output of floating-point intervals is influenced by the same parameters as the output of floating-point numbers:

`DIGITS`, `Pref::floatFormat`, and `Pref::trailingZeroes`.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Each sub-object of object can be evaluated multiple times and must not have any side-effects.

Examples

Example 1

`hull` returns an interval enclosing its arguments. You can also use the operator `...` instead of the function call:

```
hull(0, PI) = 0 ... PI
```

$$0.0 \ldots 3.141592654 = 0.0 \ldots 3.141592654$$

Infinities are displayed using `RD_NINF` for $-\infty$ and `RD_INF` for *infinity*:

```
hull(-infinity, 9/7), hull({1/4, 9/7, infinity})
```

$$\mathrm{RD\_NINF} \ldots 1.285714286,\ 0.25 \ldots \mathrm{RD\_INF}$$

Please note that any number whose absolute value is larger than MuPAD can store in a float is considered infinite:

```
hull(0, 1e100000000)^4
```

$$0.0 \ldots \mathrm{RD\_INF}$$

Example 2

Inversion of intervals may lead to unions of intervals. If these are not required, you may use `hull` to unify them:

```
1/(-1 ... 1); hull(%)
```

$$\mathrm{RD\_NINF} \ldots -1.0 \cup 1.0 \ldots \mathrm{RD\_INF}$$

$$\mathrm{RD\_NINF} \ldots \mathrm{RD\_INF}$$

Example 3

The application of `hull` to an identifier without a value returns an interval representing the complex plane:

```
delete x:  hull(x)
```

$$\text{RD\_NINF} \ldots \text{RD\_INF} + \text{RD\_NINF} \ldots \text{RD\_INF}\, i$$

Certain properties are respected during this conversion:

```
assume(x > 0): hull(x);
delete x:
```

$$0.0 \ldots \text{RD\_INF}$$

This way, you can enclose the values of an expression:

```
hull(sin(abs(x)))
```

$$-1.0 \ldots 1.0$$

Calls to “unknown” functions are regarded as potentially returning the complex plane:

```
hull(f(x))
```

$$\text{RD\_NINF} \ldots \text{RD\_INF} + \text{RD\_NINF} \ldots \text{RD\_INF}\, i$$

Parameters

`l`, `r`, `object`

Arbitrary MuPAD objects

Return Values

floating-point interval, the empty set, or `FAIL`.

Overloaded By

object

See Also

MuPAD Domains
`Dom::FloatIV`

MuPAD Functions
`DIGITS` | `float` | `interval` | `Pref::floatFormat` | `Pref::trailingZeroes`

hypergeom

Hypergeometric functions

Syntax

```
hypergeom([a1, a2, …], [b1, b2, …], z)
```

Description

`hypergeom([a1, a2, ...], [b1, b2, ...], z)` represents the hypergeometric function.

The hypergeometric function is defined for complex arguments a_i, b_j, and z.

With $a = [a_1, a_2, \ldots, a_p]$ and $b = [b_1, b_2, \ldots, b_q]$, the hypergeometric function of order p, q is defined as

$$ {}_pF_q(a; b; z) = \sum_{k=0}^{\infty} \frac{(a_1)_k (a_2)_k \cdots (a_p)_k}{(b_1)_k (b_2)_k \cdots (b_q)_k} \frac{z^k}{k!} , $$

where $(c)_k = c\,(c + 1) \ldots (c + k - 1)$, $(c)_0 = 1$ is the usual Pochhammer symbol. The quantities a and b are called 'the lists for the upper and lower parameters,' respectively.

A floating-point value is returned if at least one of the arguments is a floating-point number and all other arguments can be converted to floating-point numbers.

For most exact arguments, the hypergeometric function returns a symbolic function call. If an upper parameter coincides with a lower parameter, these values cancel and are removed from the parameter lists.

The following special values are implemented:

- ${}_pF_p(a; a; z) = {}_0F_0(; ; z) = e^z$
- ${}_pF_q(a; b; z) = 1$ if the list of upper parameters `a` contains more zeroes than the list of lower parameters `b`.

- ${}_pF_q(a; b; 0) = 1$.

If, after cancellation of identical parameters, the upper parameters contain a negative integer larger than the largest negative integer in the lower parameters, then ${}_pF_q(a; b; z)$ is a polynomial in `z`. If all upper and lower parameters as well as the argument `z` do not contain any symbolic identifiers, a corresponding explicit result is returned. If the parameters or `z` contain symbols, expansion to the polynomial representation is available via `simplify`. Cf. “Example 2” on page 1-945.

Also empty lists a = [] or b = [] may be passed to `hypergeom`. The corresponding functions are:

$${}_0F_q(; b; z) = \sum_{k=0}^{\infty} \frac{1}{(b_1)_k (b_2)_k \cdots (b_q)_k} \frac{z^k}{k!},$$

$${}_pF_0(a; ; z) = \sum_{k=0}^{\infty} (a_1)_k (a_2)_k \cdots (a_p)_k \frac{z^k}{k!},$$

$${}_0F_0(; ; z) = \sum_{k=0}^{\infty} \frac{z^k}{k!} = e^z.$$

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Symbolic calls are returned for exact or symbolic arguments:

```
hypergeom([], [2], x),
```

```
hypergeom([1], [2, 3], PI),
hypergeom([1, 1/2], [1/3], x + 3*I)
```

$$ {}_0F_1(\text{Null};\ 2;\ x),\ {}_1F_2(1;\ 2,\ 3;\ \pi),\ {}_2F_1\left(\frac{1}{2},\ 1;\ \frac{1}{3};\ x+3\,i\right) $$

Floating point values are returned for floating-point arguments:

```
hypergeom([], [2], 3.0),
hypergeom([1], [2.0], PI),
hypergeom([PI], [2, 3], 4.0),
hypergeom([1, 2], [3, 4, 5, 6], 1.0*I),
hypergeom([1 + I], [1/(2 + I)], 1.0*I)
```

$$ 3.468649619,\ 7.047601352,\ 5.152314068,\ 0.9999801588 + 0.005555508314\,i,\ -0.7438410785 - 0.5956994573\,i $$

Example 2

${}_0F_0(;\ ;\ z)$ is equal to e^z:

```
hypergeom([], [], z)
```

$$ e^z $$

Because identical values in a and b cancel, the same is true for ${}_pF_p(a;\ a;\ z)$:

```
hypergeom([a, b], [a, b], z)
```

$$ e^z $$

Any hypergeometric function, evaluated at 0, has the value 1:

```
hypergeom([a, b], [c, d, e], 0)
```

$$ 1 $$

If, after cancelling identical parameters, the list of upper parameters contains a zero, the resulting hypergeometric function is constant with the value 1:

```
hypergeom([0, 0, 2, 3], [a, 0, 4], z)
```

$$1$$

If, after cancelling identical parameters, the upper parameters contain a negative integer larger than the largest negative integer in the lower parameters, the hypergeometric function is a polynomial. If all parameters as well as the argument `z` are numerical, a corresponding explicit value is returned:

```
hypergeom([-4, -2 , 3], [-3, 1, 4], PI*sqrt(2))
```

$$\frac{6\,\pi^2}{5} - 2\,\pi\,\sqrt{2} + 1$$

For symbolic parameters or symbolic `z`, the polynomial representation may be obtained via `simplify` or `Simplify`:

```
hypergeom([-40, -5], [1, 4], z) =
simplify(hypergeom([-40, -5], [1, 4], z))
```

$${}_2F_2(-40, -5; 1, 4; z) = \frac{27417\,z^5}{280} + \frac{45695\,z^4}{84} + \frac{2470\,z^3}{3} + 390\,z^2 + 50\,z + 1$$

```
hypergeom([-3, a], [b], z) =
Simplify(hypergeom([-3, a], [b], z))
```

$${}_2F_1(-3, a; b; z) = \frac{3\,a\,z^2\,(2\,a+2)}{2\,b\,(b+1)} - \frac{3\,a\,z}{b} - \frac{a\,z^3\,(2\,a+2)\,(a+2)}{2\,b\,(b+1)\,(b+2)} + 1$$

If the largest negative integer in the list of lower parameters is larger than the largest negative integer in the list of upper parameters, the corresponding hypergeometric function is not defined (because its definition involves a division by zero):

```
hypergeom([-40, -5, 3], [-3, 1, 4], z)
```

```
Error: Invalid arguments. [hypergeom]
```

Example 3

The functions `float`, `diff`, and `series` handle expressions involving the hypergeometric functions:

```
float(ln(3 + hypergeom([17], [exp(1), ln(5)], sqrt(PI))))
```

$$3.488880173$$

```
diff(hypergeom([a, b], [c, d], x), x)
```

$$\frac{a\,b\,{}_2F_2(a+1, b+1; c+1, d+1; x)}{c\,d}$$

Note that differentiation of a hypergeometric function w.r.t. one of its uppper or lower parameters does not, in general, lead to hypergeometric functions. Certain peculiar cases are an exception:

```
diff(hypergeom([a + 1, b], [a + 2], x), a)
```

$$\frac{b\,x\,{}_3F_2(a+2, a+2, b+1; a+3, a+3; x)}{(a+2)^2}$$

```
series(hypergeom([1, 2], [3], x), x)
```

$$1+\frac{2\,x}{3}+\frac{x^2}{2}+\frac{2\,x^3}{5}+\frac{x^4}{3}+\frac{2\,x^5}{7}+O(x^6)$$

Expansions about *infinity* are possible:

```
series(hypergeom([1/2], [1/3], x), x = infinity, 3)
```

$$\frac{2\sqrt{3}\,\sqrt{\pi}\,x^{1/6}\,e^x}{3\,\Gamma\left(\frac{2}{3}\right)}-\frac{\sqrt{3}\,\sqrt{\pi}\,e^x}{18\,x^{5/6}\,\Gamma\left(\frac{2}{3}\right)}-\frac{5\sqrt{3}\,\sqrt{\pi}\,e^x}{144\,x^{11/6}\,\Gamma\left(\frac{2}{3}\right)}+O\left(\frac{e^x}{x^{17/6}}\right)$$

However, there are very few (if any) complete expansions for hypergeometric functions about any of its upper or lower parameters.

Example 4

Often, at particular choices of parameters, the hypergeometric function reduces to simpler special functions. For example, in the case of ${}_1F_1$, also known as the standard confluent hypergeometric function, the hypergeometric function can be reduced to a Bessel function if its (single) lower parameter is exactly twice its (single) upper parameter. This is verified numerically below:

```
v:= 1.0 + I: z:= float(PI):
hypergeom([v + 1/2], [2*v + 1], 2*I*z) =
(gamma(1 + v)*exp(I*z)*((z/2)^(-v))*besselJ(v, z))
```

$$-0.2766083174 - 0.2537119431\,i = -0.2766083174 - 0.2537119431\,i$$

```
delete v, z:
```

In the following example, ${}_2F_1$, which is known as the Gauss hypergeometric function, can be reduced into a simple elementary function involving logarithms when the parameters are `[1, 1], [2]`, as verified numerically below:

```
eq := hypergeom([1, 1], [2], z) = -ln(1 - z)/z:
float(subs(eq, z = 1/3)), float(subs(eq, z = 1/2))
```

$$1.216395324 = 1.216395324,\ 1.386294361 = 1.386294361$$

```
delete eq:
```

Example 5

The interval $[1, \infty)$ is a branch cut for the hypergeometric function; the sign of the imaginary part changes when crossing the cut. The branch cut belongs to the lower branch:

```
eq := hypergeom([1, 1], [2], z) = -ln(1 - z)/z:
float(subs(eq, z = 2 + I*10^(-DIGITS)))
```

$$7.853981634\,10^{-11} + 1.570796327\,i = 7.853981634\,10^{-11} + 1.570796327\,i$$

```
float(subs(eq, z = 2 - I*10^(-DIGITS)))
```

$$7.853981634\,10^{-11} - 1.570796327\,i = 7.853981634\,10^{-11} - 1.570796327\,i$$

```
float(subs(eq, z = 2))
```

$$-1.570796327\,i = -1.570796327\,i$$

Parameters

a_1, a_2, …

The 'upper parameters': arithmetical expressions

b_1, b_2, …

The 'lower parameters': arithmetical expressions

z

The 'argument': an arithmetical expression

Return Values

Arithmetical expression.

Overloaded By

z

Algorithms

When no b_j in the list b lies in the set {0, - 1, - 2, …}, the series

$$ {}_pF_q(a;\, b;\, z) = \sum_{k=0}^{\infty} \frac{(a_1)_k\,(a_2)_k \cdots (a_p)_k}{(b_1)_k\,(b_2)_k \cdots (b_q)_k}\,\frac{z^k}{k!} $$

converges if one of the following conditions hold:

1. $p \le q,\ |z| < \infty$;
2. $p = q + 1,\ |z| < 1$;
3. $p = q + 1,\ |z| = 1,\ \Re(\psi_q) > 0$;
4. $p = q + 1,\ |z| = 1,\ z \neq 1,\ -1 < \Re\left(\psi_q\right) \le 0$;
5. a contains a zero or a negative integer;

where $\psi_q = \left(\sum_{k=1}^{q} b_k\right) - \left(\sum_{j=1}^{q+1} a_j\right)$. The series diverges in the remaining cases. If one of the parameters in a isequal to zero or a negative integer, then the series terminates, turning into what is called a hypergeometric polynomial.

The generalized hypergeometric function of order (p, q) is given by the series definition in the region of convergence, while for $p = q + 1,\ |z| \ge 1$, it is defined as an analytic continuation of this series.

The function ${}_pF_q(a;\, b;\, z)$ is symmetric w.r.t. the parameters, i.e., it does not depend on the order of the arrangement $a_1, a_2, \ldots$ in a or $b_1, b_2, \ldots$ in b.

As mentioned above, if some upper parameter is equal to $n = 0, -1, -2, \ldots$, the function turns into a polynomial of degree n. If we relax the condition stated above for the lower parameters b and there is some lower parameter equal to $m = 0, -1, -2, \ldots$, the function ${}_pF_q(a;\, b;\, z)$ also reduces to a polynomial in z provided $n > m$. It is undefined if $m > n$ or if no upper parameter is a nonpositive integer (resulting in division by zero in one of the series coefficients). The case $m = n$ is handled by the following rule.

If for r values of the upper parameters, there are r values of the lower parameters equal to them (i.e., $a = [a_1, \ldots, a_{p-r}, c_1, \ldots, c_r]$, $b = [b_1, \ldots, b_{q-r}, c_1, \ldots, c_r]$), then the order (p, q) of the function ${}_pF_q(a;\, b;\, z)$ is reduced to $(p - r, q - r)$:

$$ {}_pF_q\left(a_1, \ldots, a_{p-r}, c_1, \ldots, c_r;\, b_1, \ldots, b_{q-r}, c_1, \ldots, c_r;\, z\right) = {}_{p-r}F_{q-r}\left(a_1, \ldots, a_{p-r};\, b_1, \ldots, b_{q-r};\, z\right). $$

The above rule applies even if any of the c_i happens to be zero or a negative integer (for details, see Luke in the list of references, p. 42).

$U(z) = {}_pF_q(a; b; z)$ satisfies a differential equation in z:

$$[\delta\,(\delta + b - 1) - z\,(\delta + a)]\,U(z) = 0,\ \delta = z\,\frac{\partial}{\partial z}$$

,

where $(\delta + a)$ and $(\delta + b)$ stand for $\prod_{i=1}^{p}\left(\delta + a_i\right)$ and $\prod_{j=1}^{q}\left(\delta + b_j\right)$, respectively.

Thus, the order of this differential equation is $max(p, q + 1)$ and the hypergeometric function is only one of its solutions. If $p < q + 1$, this differential equation has a regular singularity at $z = 0$ and an irregular singularity at $z = \infty$. If $p = q + 1$, the points $z = 0$, $z = 1$, and $z = \infty$ are regular singularities, thus explaining the convergence properties of the hypergeometric series.

The analytic continuation for $p = q + 1$, $|z| \geq 1$, is defined by selecting the principal branch of this continuation (also denoted as ${}_pF_q(a; b; z)$) satisfying the condition $|arg(1 - z)| < \pi$, the cut along $[1, \infty)$ is drawn in the complex z-plane. In particular, the analytic continuation can be obtained by means of an integral representation (for details, see Prudnikov *et al.* in the references) or by the Meijer G function.

References

[1] Luke, Y.L. “The Special Functions and Their Approximations”, Vol. 1, Academic Press, New York, 1969.

[2] Prudnikov, A.P., Yu.A. Brychkov, and O.I. Marichev, “Integrals and Series”, Vol. 3: More Special Functions, Gordon and Breach, 1990.

[3] Abramowitz, M. and I.A. Stegun, “Handbook of Mathematical Functions”, Dover Publications, New York, 9th printing, 1970.

icontent

Content of a polynomial with rational coefficients

Syntax

```
icontent(p)
```

Description

`icontent(p)` calculates the content of a polynomial or polynomial expression with integer or rational coefficients, i.e., the greatest common divisor of the coefficients, such that `p/ icontent(p)` has integral coefficients whose greatest common divisor is `1`. In particular, if `p` is itself an integer or a rational number, then `icontent` returns `abs(p)` (see “Example 1” on page 1-952).

If `p` is a polynomial or polynomial expression with integer coefficients, then the content is the greatest common divisor of the coefficients. If `p` is a polynomial or polynomial expression with rational coefficients, then the content is the greatest common divisor of the numerators of the coefficients divided by the least common multiple of the denominators (see “Example 2” on page 1-953).

If `p` is a polynomial expression, then it is first converted into a polynomial of domain type `DOM_POLY` using `poly`. If this conversion is not possible, then `icontent` returns `FAIL`.

`icontent` returns an error message if not all coefficients of `p` are integers or rational numbers.

Examples

Example 1

The first argument can be a polynomial or a polynomial expression. The following two calls of `icontent` are equivalent:

```
p := 6*x*y - 9*y^2 + 21:
```

```
icontent(poly(p)), icontent(p)
```

$$3,\ 3$$

The result of `icontent` is always nonnegative:

```
icontent(2*x - 4), icontent(-2*x + 4)
```

$$2,\ 2$$

The content of a constant polynomial is its absolute value:

```
icontent(0), icontent(-2), icontent(poly(-2, [x]))
```

$$0,\ 2,\ 2$$

Example 2

The content of a polynomial with rational coefficients is a rational number in general:

```
q := 6/7*x*y - 9/4*y + 12:
icontent(poly(q)), icontent(q)
```

$$\frac{3}{28},\ \frac{3}{28}$$

The polynomial divided by its content has integral coefficients whose greatest common divisor is 1:

```
q/icontent(q)
```

$$8\,x\,y - 21\,y + 112$$

```
icontent(%)
```

$$1$$

Parameters

p

A polynomial or polynomial expression with integer or rational number coefficients

Return Values

Nonnegative integer or rational number, or `FAIL`

See Also

MuPAD Functions

`coeff` | `content` | `factor` | `gcd` | `ifactor` | `igcd` | `ilcm` | `lcm` | `poly` | `polylib::primpart`

id

Identity map

Syntax

```
id(x)
```

```
id(x1, x2, …)
```

Description

`id(x)` evaluates and returns `x`; `id(x1, x2, ...)` returns the evaluated arguments as an expression sequence; `id()` returns the void object `null()`.

Examples

Example 1

`id` returns the evaluated arguments:

```
a := 2: id(a + 2)
```

$$4$$

```
id(a, b, 4 + 2)
```

$$2, b, 6$$

`id()` returns `null()`:

```
domtype(id())
```

$$\text{DOM\_NULL}$$

```
delete a:
```

Example 2

`id` is useful when working with functional expressions:

```
f := 3*id + sin + 5*id^2 + exp@(-id^2):
f(x)
```

$$3\,x + e^{-x^2} + \sin(x) + 5\,x^2$$

```
f'(x)
```

$$10\,x + \cos(x) - 2\,x\,e^{-x^2} + 3$$

```
delete f:
```

Parameters

`x, x1, x2, …`

Arbitrary MuPAD objects

Return Values

Sequence of the input parameters.

if, then, elif, else, end_if, _if

If-statement (conditional branch in a program)

Syntax

```
if condition1
then casetrue1
   elif condition2 then casetrue2
   elif condition3 then casetrue3
   ...
   else casefalse
 end_if

_if(condition1, casetrue1, casefalse)
```

Description

`if-then-else-end_if` allows conditional branching in a program.

If the Boolean expression `condition1` can be evaluated to `TRUE`, the branch `casetrue1` is executed and its result is returned. Otherwise, if `condition2` evaluates to `TRUE`, the branch `casetrue2` is executed and its result is returned etc. If all of the conditions evaluate to `FALSE`, the branch `casefalse` is executed and its result is returned.

All conditions that are evaluated during the execution of the `if` statement must be reducible to either `TRUE` or `FALSE`. Conditions may be given by equations or inequalities, combined with the logical operators `and`, `or`, `not`. There is no need to enforce Boolean evaluation of equations and inequalities via `bool`. Implicitly, the `if` statement enforces "lazy" Boolean evaluation via the functions `_lazy_and` or `_lazy_or`, respectively. A condition leads to a runtime error if it cannot be evaluated to `TRUE` or `FALSE` by these functions. Cf. "Example 3" on page 1-959.

The keyword `end_if` may be replaced by the keyword `end`.

The statement `if condition then casetrue else casefalse end_if` is equivalent to the function call `_if(condition, casetrue, casefalse)`.

Examples

Example 1

The `if` statement operates as demonstrated below:

```
if TRUE then YES else NO end_if,
if FALSE then YES else NO end_if
```

```
YES, NO
```

The `else` branch is optional:

```
if FALSE then YES end_if
```

```
if FALSE
  then if TRUE
         then NO_YES
         else NO_NO
       end_if
  else if FALSE
         then YES_NO
         else YES_YES
       end_if
end_if
```

```
YES_YES
```

Typically, the Boolean conditions are given by equations, inequalities or Boolean constants produced by system functions such as `isprime`:

```
for i from 100 to 600 do
  if 105 < i and i^2 <= 17000 and isprime(i) then
     print(expr2text(i)." is a prime")
  end_if;
  if i < 128 then
     if isprime(2^i - 1) then
        print("2^".expr2text(i)." - 1 is a prime")
     end_if
  end_if
end_for:
```

"107 is a prime"

"2^107 - 1 is a prime"

"109 is a prime"

"113 is a prime"

"127 is a prime"

"2^127 - 1 is a prime"

Example 2

Instead of using nested `if-then-else` statements, the `elif` statement can make the source code more readable. However, internally the parser converts such statements into equivalent `if-then-else` statements:

```
hold(if FALSE then NO elif TRUE then YES_YES else YES_NO end_if)
```

```
if FALSE then
  NO
else
  if TRUE then
    YES_YES
  else
    YES_NO
  end_if
end_if
```

Example 3

If the condition cannot be evaluated to either TRUE or FALSE, then a runtime error is raised. In the following call, `is(x > 0)` produces UNKNOWN if no corresponding properties was attached to `x` via `assume`:

```
if is(x > 0) then
  1
else
  2
end_if
```

```
Error: Cannot evaluate to Boolean. [if]
```

Note that Boolean conditions using <, <=, >, >= may fail if they involve symbolic expressions:

```
if 1 < sqrt(2) then print("1 < sqrt(2)"); end_if
```

"1 < sqrt(2)"

```
if 10812186006/7645370045 < sqrt(2)
  then print("10812186006/7645370045 < sqrt(2)");
end_if
```

"10812186006/7645370045 < sqrt(2)"

```
if is(10812186006/7645370045 < sqrt(2)) = TRUE
  then print("10812186006/7645370045 < sqrt(2)");
end_if
```

"10812186006/7645370045 < sqrt(2)"

Example 4

This example demonstrates the correspondence between the functional and the imperative use of the `if` statement:

```
condition := 1 > 0: _if(condition, casetrue, casefalse)
```

casetrue

```
condition := 1 > 2: _if(condition, casetrue, casefalse)
```

casefalse

```
delete condition:
```

Parameters

$condition_1$, $condition_2$, …

Boolean expressions

$casetrue_1$, $casetrue_2$, casefalse, …

Arbitrary sequences of statements

Return Values

Result of the last command executed inside the `if` statement. The empty sequence, `null()` is returned if no command was executed.

See Also

MuPAD Functions

`case` | `piecewise`

More About

- "Conditional Control"

ifactor

Factor an integer into primes

Syntax

```
ifactor(n, <UsePrimeTab>)
```

```
ifactor(<PrimeLimit>)
```

Description

`ifactor(n)` computes the prime factorization $n = s\,p_1^{e_1} \ldots p_r^{e_r}$ of the integer n, where s is the sign of n, p_1, …, p_r are the distinct positive prime divisors of n, and e_1, …, e_r are positive integers.

The result of `ifactor` is an object of domain type `Factored`. Let `f:= ifactor(n)` be such an object. Internally, it is represented by the list `[s, p1, e1, ..., pr, er]` of odd length $2r + 1$, where r is the number of distinct prime divisors of n. The p_i are not necessarily sorted by magnitude.

You may extract the sign s and the terms $p_i^{e_i}$ by means of the index operator `[ ]`, i.e., `f[1] = p1^e1, f[2] = p2^e2, ...` for positive n and `f[1] = s, f[2] = p1^e1, f[3] = p2^e2, ...` for negative n.

The call `Factored::factors(f)` yields a list of the factors `[p1, p2, ...]`, while `Factored::exponents(f)` returns a list of the exponents `[e1, e2, ...]` with $1 \le i \le r$.

The factorization of 0, 1, and - 1 yields the single factor 0, 1, and - 1, respectively. In these cases, the internal representation is the list `[0]`, `[1]`, and `[-1]`, respectively.

The call `coerce(f,DOM_LIST)` returns the internal representation of a factored object, i.e., the list `[s, p1, e1, p2, e2, ...]`.

Note that the result of `ifactor` is printed as an expression, and it is implicitly converted into an expression whenever it is processed further by other MuPAD functions. For

example, the result of `ifactor(12)` is printed as `2^2*3`, which is an expression of type `"_mult"`.

See “Example 1” on page 1-963 for illustrations, and the help page of `Factored` for more details.

If you do not need the prime factorization of `n`, but only want to know whether it is composite or prime, use `isprime` instead, which is much faster.

`ifactor` returns an error when the argument is a number but not an integer. A symbolic `ifactor` call is returned if the argument is not a number.

Examples

Example 1

To get the prime factorization of 120, enter:

```
f := ifactor(120)
```

$$2^3\ 3\ 5$$

You can access the terms of this factorization using the index operator:

```
f[1], f[2], f[3]
```

$$2^3,\ 3,\ 5$$

The internal representation of `f`, namely the list as described above, is returned by the following command:

```
coerce(f, DOM_LIST)
```

$$[1,\ 2,\ 3,\ 3,\ 1,\ 5,\ 1]$$

The result of `ifactor` is an object of domain type `Factored`:

```
domtype(f)
```

Factored

This domain implements some features for handling such objects. Some of them are described below.

You may extract the factors and exponents of the factorization also in the following way:

```
Factored::factors(f), Factored::exponents(f)
```

[2, 3, 5], [3, 1, 1]

You can ask for the type of the factorization:

```
Factored::getType(f)
```

"irreducible"

This output means that all factors p_i are prime. Other possible types are `"squarefree"` (see `polylib::sqrfree`) or `"unknown"`.

Multiplying factored objects preserves the factored form:

```
f2 := ifactor(12)
```

$2^2\ 3$

```
f*f2
```

$2^5\ 3^2\ 5$

It is important to note that you can apply nearly any function operating on arithmetical expressions to an object of domain type `Factored`. The result is usually not of this domain type:

```
expand(f);
```

```
domtype(%)
```

$$120$$

$$\mathrm{DOM\_INT}$$

For a detailed description of these objects, please refer to the help page of the domain `Factored`.

Example 2

The factorizations of `0`, `1`, and `-1` each have exactly one factor:

```
ifactor(0), ifactor(1), ifactor(-1)
```

$$0,\ 1,\ -1$$

```
map(%, coerce, DOM_LIST)
```

$$[0],\ [1],\ [-1]$$

The internal representation of the factorization of a prime number `p` is the list `[1, p, 1]`:

```
coerce(ifactor(5), DOM_LIST)
```

$$[1,\ 5,\ 1]$$

Example 3

The bound on the prime number table is:

```
ifactor(PrimeLimit)
```

$$1000000$$

We assign a large prime number to p:

```
p := nextprime(10^10);
q := nextprime(10^12)
```

```
10000000019
```

```
1000000000039
```

Completely factoring the 36 digit number 6*p^3 takes some time; the second output line shows the time in seconds:

```
t := time():
f := ifactor(p^3*q^4);
(time() - t)/1000.0
```

```
10000000019^3*1000000000039^4
```

```
2.5
```

```
Factored::getType(f)
```

```
"irreducible"
```

```
delete f
```

Extracting only the prime factors in the prime table is much faster, but it does not yield the complete factorization; the factor p^3 remains undecomposed:

```
t := time():
f := ifactor(p^3*q^4, UsePrimeTab);
(time() - t)/1000.0
```

```
10000000058560000117283260086007354771701933667061781196953525306500458678918 19
```

```
0.015625
```

```
Factored::getType(f)
```

```
"unknown"
```

```
delete f
```

Parameters

`n`

An arithmetical expression representing an integer

Options

`UsePrimeTab`

Internally, MuPAD has stored a pre-computed table of all prime numbers up to a certain bound. `ifactor(n, UsePrimeTab)` looks only for prime factors that are stored in this internal prime number table, extracts them from `n`, and returns the undecomposed product of all other prime factors as a single factor. This is usually much faster than without the option `UsePrimeTab`, but it does not necessarily yield the complete prime factorization of `n`. See “Example 2” on page 1-965.

`PrimeLimit`

`ifactor(PrimeLimit)` returns an integer, namely a bound on the size of prime numbers in the internal prime number table. The table contains all primes below this bound. The default values are: 1000000 on UNIX systems and 300000 on Mac OS platforms and Windows platforms.

The size of this table can be changed via the MuPAD command line flag `-L`.

Return Values

Object of domain type `Factored`, or a symbolic `ifactor` call.

Algorithms

`ifactor` uses the elliptic curve method.

`ifactor` is an interface to the kernel function `stdlib::ifactor`. It calls `stdlib::ifactor` with the given arguments and convert its result, which is the list

`[s, p1, e1, ..., pr, er]` as described above, into an object of the domain type `Factored`.

You may directly call the kernel function `stdlib::ifactor` inside your routines, in order to avoid this conversion and to decrease the running time.

See Also

MuPAD Functions

`content` | `factor` | `Factored` | `icontent` | `igcd` | `ilcm` | `isprime` | `ithprime` | `nextprime` | `numlib::divisors` | `numlib::primedivisors` | `prevprime`

ifourier

Inverse Fourier transform

Syntax

```
ifourier(F, w, t)
```

Description

`ifourier(F, w, t)` computes the inverse Fourier transform of the expression `F = F(w)` with respect to the variable `w` at the point `t`.

The inverse Fourier transform of the expression `F = F(w)` with respect to the variable `w` at the point `t` is defined as follows:

$$f(t) = \frac{|s|}{2\,\pi\,c} \int_{-\infty}^{\infty} F(w)\, e^{-i\,s\,w\,t}\, dw$$

.

`c` and `s` are parameters of the Fourier transform. By default, `c = 1` and `s = -1`.

To change the parameters `c` and `s` of the Fourier transform, use `Pref::fourierParameters`. See “Example 3” on page 1-971. Common choices for the parameter `c` are 1, $\frac{1}{2\,\pi}$, or $\frac{1}{\sqrt{2\,\pi}}$. Common choices for the parameter `s` are -1, 1, - 2 π, or 2 π.

If `F` is a matrix, `ifourier` applies the inverse Fourier transform to all components of the matrix.

MuPAD computes `ifourier(F, w, t)` as

$$\frac{|s|}{2\,\pi\,c^2}\ \mathrm{fourier}(F(w),\, w,\, -t)$$

.

If `ifourier` cannot find an explicit representation of the inverse Fourier transform, it returns results in terms of the direct Fourier transform. See “Example 4” on page 1-972.

To compute the direct Fourier transform, use `fourier`.

To compute the inverse discrete Fourier transform, use `numeric::invfft`.

Environment Interactions

Results returned by `ifourier` depend on the current `Pref::fourierParameters` settings.

Examples

Example 1

Compute the inverse Fourier transform of this expression with respect to the variable `w`:

```
ifourier(sqrt(PI)*exp(-w^2/4), w, t)
```

$$e^{-t^2}$$

Example 2

Compute the inverse Fourier transform of this expression with respect to the variable `w` for positive values of the parameter t_0:

```
assume(t_0 > 0):
f := ifourier(-(PI^(1/2)*w*exp(-w^2*t_0^2/4)*I)*t_0^3/2, w, t)
```

$$t\,e^{-\frac{t^2}{t_0^2}}$$

Evaluate the inverse Fourier transform of the expression at the points $t = -2\,t_0$ and $t = 1$. You can evaluate the resulting expression `f` using `|` (or its functional form `evalAt`):

```
f | t = -2*t_0
```

$$-2\,t_0\,e^{-4}$$

Also, you can evaluate the inverse Fourier transform at a particular point directly:

```
ifourier(-(PI^(1/2)*w*exp(-w^2*t_0^2/4)*I)*t_0^3/2, w, 1)
```

$$e^{-\frac{1}{t_0^2}}$$

Example 3

The default parameters of the Fourier and inverse Fourier transforms are `c = 1` and `s = -1`:

```
ifourier(-(sqrt(PI)*w*exp(-w^2/4)*I)/2, w, t)
```

$$t\,e^{-t^2}$$

To change these parameters, use `Pref::fourierParameters` before calling `ifourier`:

```
Pref::fourierParameters(1, 1):
```

Evaluate the transform of the same expression with the new parameters:

```
ifourier(-(sqrt(PI)*w*exp(-w^2/4)*I)/2, w, t)
```

$$-t\,e^{-t^2}$$

For further computations, restore the default values of the Fourier transform parameters:

```
Pref::fourierParameters(NIL):
```

Example 4

If `ifourier` cannot find an explicit representation of the transform, it returns results in terms of the direct Fourier transform:

```
ifourier(exp(-w^4), w, t)
```

$$\frac{\text{fourier}\left(e^{-w^4}, w, -t\right)}{2\,\pi}$$

Example 5

Compute the following inverse Fourier transforms that involve the Dirac and the Heaviside functions:

```
ifourier(dirac(w), w, t)
```

$$\frac{1}{2\,\pi}$$

```
ifourier(heaviside(w + 5), w, t)
```

$$\frac{e^{-5\,t\,i}\left(\pi\,\delta(t) + \frac{i}{t}\right)}{2\,\pi}$$

Parameters

F

Arithmetical expression or matrix of such expressions

w

Identifier or indexed identifier representing the transformation variable

t

Arithmetical expression representing the evaluation point

Return Values

Arithmetical expression or an expression containing an unevaluated function call of type `fourier`. If the first argument is a matrix, then the result is returned as a matrix.

Overloaded By

F

References

F. Oberhettinger, “Tables of Fourier Transforms and Fourier Transforms of Distributions”, Springer, 1990.

See Also

MuPAD Functions

`fourier` | `fourier::addpattern` | `ifourier::addpattern` | `numeric::fft` | `numeric::invfft` | `Pref::fourierParameters`

ifourier::addpattern

Add patterns for the inverse Fourier transform

Syntax

```
ifourier::addpattern(pat, w, t, res, <vars, <conds>>)
```

Description

`ifourier::addpattern(pat, w, t, res)` teaches `ifourier` to return `res` for the expression `pat`.

The `ifourier` function uses a set of patterns for computing inverse Fourier transforms. You can extend the set by adding your own patterns. To add a new pattern to the pattern matcher, use `ifourier::addpattern`. MuPAD does not save custom patterns permanently. The new patterns are available in the *current* MuPAD session only.

After the call `ifourier::addpattern(pat, w, t, res)`, the `ifourier` function returns `res` for the expression `pat`. Note that the inverse Fourier transform is defined as $\frac{|s|}{2\pi c}\int_{-\infty}^{\infty} F\, \mathrm{e}^{-\mathrm{i}\,s\,w\,t}\,\mathrm{d}w$, where `c` and `s` are the parameters specified by `Pref::fourierParameters`. If you add a new pattern, and then change the Fourier transform parameters, the result returned by `ifourier(pat, w, t)` will also change. See “Example 2” on page 1-976.

Variable names that you use when calling `ifourier::addpattern` can differ from the names that you use when calling `ifourier`. See “Example 3” on page 1-976.

You can include a list of free parameters and a list of conditions on these parameters. These conditions and the result are protected from premature evaluation. That means you can use `not iszero(a^2-b)` instead of `hold( _not @ iszero )(a^2-b)`.

The following conditions treat assumptions on identifiers differently:

- `a^2-b <> 0` takes into account assumptions on identifiers.
- `not iszero(a^2-b)` disregards assumptions on identifiers.

See “Example 4” on page 1-976 and “Example 5” on page 1-977.

Environment Interactions

The Fourier pair `(pat, res)` holds only for the current values of the Fourier transform parameters specified by `Pref::fourierParameters`.

Calling `ifourier::addpattern` can change the expressions returned by future calls to `fourier` and `ifourier` in the current MuPAD session.

Examples

Example 1

Compute the inverse Fourier transform of the function `bar`. By default, MuPAD does not have a pattern for this function:

```
ifourier(bar(w), w, t)
```

$$\frac{\mathrm{fourier}(\mathrm{bar}(w), w, -t)}{2\,\pi}$$

Add a pattern for the inverse Fourier transform of `bar` using `ifourier::addpattern`:

```
ifourier::addpattern(bar(w), w, t, foo(t)):
```

Now `ifourier` returns the Fourier transform of `bar`:

```
ifourier(bar(w), w, t)
```

$$\mathrm{foo}(t)$$

After you add a new transform pattern, MuPAD can use that pattern indirectly:

```
ifourier(exp(-a*I*s)*bar(2*s + 10), s, t)
```

$$\frac{\mathrm{foo}\left(\frac{t}{2}-\frac{a}{2}\right)\,\mathrm{e}^{5\,a\,i-5\,t\,i}}{2}$$

Example 2

Add this pattern for the inverse Fourier transform of the function `bar`:

```
ifourier::addpattern(bar(w), w, t, foo(t)):
ifourier(bar(w), w, t)
```

$$\mathrm{foo}(t)$$

Now change the Fourier transform parameters using `Pref::fourierParameters`:

```
Pref::fourierParameters(a, b):
```

Evaluate the transform with the new parameters:

```
ifourier(bar(w), w, t)
```

$$\frac{\mathrm{foo}(-b\,t)\,|b|}{a}$$

For further computations, restore the default values of the Fourier transform parameters:

```
Pref::fourierParameters(NIL):
```

Example 3

Define the inverse Fourier transform of `bar(y)` using variables `y` and `x` as parameters:

```
ifourier::addpattern(bar(y), y, x, foo(x)):
```

The `ifourier` function recognizes the added pattern even if you use other variables as parameters:

```
ifourier(bar(w), w, t)
```

$$\mathrm{foo}(t)$$

Example 4

Use assumptions when adding the following pattern for the inverse Fourier transform:

```
ifourier::addpattern(bar(x, w), w, t, foo(x, t), [x], [abs(x) < 1]):
ifourier(bar(x, w), w, t) assuming abs(x) < 1/2
```

$$\mathrm{foo}(x, t)$$

If $|x| \geq 1$, you cannot apply these patterns:

```
ifourier(bar(x, w), w, t) assuming x < -1
```

$$\frac{\mathrm{fourier}(\mathrm{bar}(x, w), w, -t)}{2\,\pi}$$

If MuPAD cannot determine whether the conditions are satisfied, it returns a `piecewise` object:

```
ifourier(bar(x, w), w, t)
```

$$\begin{cases} \mathrm{foo}(x, t) & \text{if } |x| < 1 \end{cases}$$

Example 5

Add this pattern for the inverse Fourier transform of `g`:

```
ifourier::addpattern(g(a, w), w, t, f(a, t)/a):
ifourier(g(a, W), W, T)
```

$$\frac{f(a, T)}{a}$$

This pattern holds only when the first argument of `g` is the symbolic parameter `a`. If you use any other value of this parameter, `ifourier` ignores the pattern:

```
ifourier(g(b, W), W, T);
ifourier(g(2, W), W, T)
```

$$\frac{\mathrm{fourier}(g(b, W), W, -T)}{2\,\pi}$$

$$\frac{\text{fourier}(g(2, W), W, -T)}{2\,\pi}$$

To use the pattern for arbitrary values of the parameter, declare the parameter `a` as an additional pattern variable:

```
ifourier::addpattern(g(a, w), w, t, f(a, t)/a, [a]):
```

Now `ifourier` applies the specified pattern for an arbitrary value of `a`:

```
ifourier(g(2, W), W, T)
```

$$\frac{f(2, T)}{2}$$

```
ifourier(g(a^2 + 1, W), W, T)
```

$$\frac{f(a^2+1, T)}{a^2+1}$$

Note that the resulting expression `f(a, t)/a` defining the Fourier transform of `g(a, w)` implicitly assumes that the value of `a` is not zero. A strict definition of the pattern is:

```
ifourier::addpattern(g(a, w), w, t, f(a, t)/a, [a], [a <> 0]):
```

For this particular pattern, you can omit specifying the assumption `a <> 0` explicitly. If `a = 0`, MuPAD throws an internal "Division by zero." error and ignores the pattern:

```
ifourier(f(0, W), W, T)
```

$$\frac{\text{fourier}(f(0, W), W, -T)}{2\,\pi}$$

Parameters

`pat`

Arithmetical expression in the variable `w` representing the pattern to match

`w`

Identifier or indexed identifier used as a variable in the pattern

`t`

Identifier or indexed identifier used as a variable in the result

`res`

Arithmetical expression in the variable `t` representing a pattern for the result

`vars`

List of identifiers or indexed identifiers used as "pattern variables" (placeholders in `pat` and `res`). You can use pattern variables as placeholders for almost arbitrary MuPAD expressions not containing `w` or `t`. You can restrict them by conditions given in the optional parameter `conds`.

`conds`

List of conditions on the pattern variables

Return Values

Object of type `DOM_NULL`

See Also

MuPAD Functions

`fourier` | `fourier::addpattern` | `ifourier`

igamma

Incomplete gamma function

Syntax

```
igamma(a, x)
```

Description

`igamma(a, x)` returns the upper incomplete gamma function $\int_x^\infty e^{-t}\, t^{a-1}\, dt$.

Note: The MATLAB `gammainc` function returns the regularized lower incomplete gamma function: `igamma(a, x) = gamma(a)(1 - gammainc(x, a))`. See the `gamma` and `gammainc` function reference pages in the MATLAB documentation.

To find the lower incomplete gamma function for arguments a and x, subtract `igamma(a, x)` from `gamma(a)`.

A floating-point value is returned if at least one of the arguments is a floating-point value and both values are numerical. Otherwise, symbolic calls of `igamma` and/or other special functions may be returned.

The following simplifications and rewriting rules are implemented: $\Gamma(a, 0) = \Gamma(a)$, $\Gamma(0, x) = \mathrm{Ei}(x)$, $\Gamma\left(\frac{1}{2}, x\right) = \sqrt{\pi}\, \mathrm{erfc}(\sqrt{x})$, $\Gamma(1, x) = e^{-x}$.

For real numerical values of a of `Type::Real` satisfying $|a| \le$ *Pref*::*autoExpansionLimit*(), the functional relation

$$\Gamma(a, x) = x^{(a-1)}\, e^{-x} + (a-1)\, \Gamma(a-1, x)$$

is used recursively to shift the first argument to the interval $0 \le a \le 1$. Thus rewriting in terms of `Ei`, `erfc`, and `exp` occurs if a is an integer multiple of $\frac{1}{2}$. Cf. "Example

1" on page 1-981. Use `expand` if these transformations are also desired for $|a| >$ *Pref*::*autoExpansionLimit*().

The special value `igamma(a, infinity) = 0` for $a \neq \infty$ is implemented.

Environment Interactions

When called with a floating-point argument, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
igamma(2, 3), igamma(1/7, x), igamma(sqrt(2), 3)
```

$$4\,e^{-3},\ \Gamma\left(\frac{1}{7}, x\right),\ \Gamma(\sqrt{2}, 3)$$

```
igamma(a, 4), igamma(1 + I, x^2 + 1), igamma(a, infinity)
```

$$\Gamma(a, 4),\ \Gamma(1+i, x^2+1),\ 0$$

If the first argument a is a real numerical value with $|a| \leq$ *Pref*::*autoExpansionLimit*(), the functional relations are used recursively until `igamma` is called with a first argument from the the interval $0 \leq a \leq 1$:

```
igamma(-1/10, 1), igamma(7/4, 1)
```

$$10\,e^{-1} - 10\,\Gamma\left(\frac{9}{10}, 1\right),\ e^{-1} + \frac{3\,\Gamma\left(\frac{3}{4}, 1\right)}{4}$$

If the first argument is an integer multiple of $\frac{1}{2}$, then complete rewriting in terms of `Ei`, `erfc`, and `exp` occurs:

```
igamma(-3, x), igamma(-5/2, x), igamma(8, x), igamma(13/2, 4)
```

$$e^{-x}\left(\frac{1}{6x}-\frac{1}{6x^2}+\frac{1}{3x^3}\right)-\frac{Ei(1,x)}{6},\ e^{-x}\left(\frac{8}{15\sqrt{x}}-\frac{4}{15x^{3/2}}+\frac{2}{5x^{5/2}}\right)-\frac{8\sqrt{\pi}\,\mathrm{erfc}(\sqrt{x})}{15},$$

$$e^{-x}(x^7+7x^6+42x^5+210x^4+840x^3+2520x^2+5040x+5040),$$

$$\frac{210979\,e^{-4}}{16}+\frac{10395\sqrt{\pi}\,\mathrm{erfc}(2)}{64}$$

Floating point values are computed for floating-point arguments:

```
igamma(0.1, 4.0), igamma(7, 0.5), igamma(100, 100.0)
```

$$0.004420083058,\ 719.9992783,\ 4.542198121\ 10^{155}$$

Example 2

The functional relation between `igamma` with different first arguments is used to "normalize" the returned expressions:

```
igamma(-8, x), igamma(7/3, x)
```

$$\frac{Ei(1,x)}{40320}-e^{-x}\left(\frac{1}{40320x}-\frac{1}{40320x^2}+\frac{1}{20160x^3}-\frac{1}{6720x^4}+\frac{1}{1680x^5}-\frac{1}{336x^6}+\frac{1}{56x^7}-\frac{1}{8x^8}\right),\ e^{-x}\left(\frac{4x^{1/3}}{3}+x^{4/3}\right)+\frac{4\,\Gamma\left(\frac{1}{3},x\right)}{9}$$

Parameters

a, x

arithmetical expressions

Return Values

Arithmetical expression.

Overloaded By

a, x

See Also

MuPAD Functions
`Ei` | `erfc` | `exp` | `fact` | `gamma` | `int`

igcd

Greatest common divisor of integers and complex numbers with integer real and imaginary parts

Syntax

```
igcd(i1, i2, …)
```

Description

`igcd(i1, i2, ...)` computes the greatest common divisor of the integers i_1, i_2, …

`igcd` computes the greatest common nonnegative divisor of a sequence of integers. If an argument of `igcd` is a single integer number, the function returns the absolute value of that argument.

`igcd` also computes the greatest common divisor of a sequence of complex numbers of the domain `DOM_COMPLEX`. Both the real and the imaginary parts of all complex numbers in a sequence must be integers. The greatest common divisor is a complex number with a positive real part and a nonnegative imaginary part.

If all arguments are `0`, `igcd` returns `0`.

If there are no arguments, `igcd` also returns `0`.

If one argument is a number, but is neither an integer nor a complex number with integer real and imaginary parts, then `igcd` returns an error message.

If at least one of the arguments is `1` or `-1`, `igcd` returns `1`. Otherwise, if one argument is not a number, the `igcd` function returns a symbolic `igcd` call.

Examples

Example 1

Compute the greatest common divisor of the following integers:

```
igcd(-10, 6), igcd(6, 10, 15)
```

$$2, 1$$

```
a := 4420, 128, 8984, 488:
igcd(a), igcd(a, 64)
```

$$4, 4$$

Example 2

Compute the greatest common divisor of the following complex numbers:

```
igcd(-10*I, 6), igcd(10 - 5*I, 20 - 10*I, 30 - 15*I)
```

$$2, 5 + 10\,i$$

Example 3

The following example shows some special cases:

```
igcd(), igcd(0), igcd(1), igcd(-1), igcd(2)
```

$$0, 0, 1, 1, 2$$

Example 4

If one argument is not a number, then the result is a symbolic `igcd` call. However, if at least one of the arguments is `1` or `-1`, the greatest common divisor is always `1`:

```
delete x:
igcd(a, x), igcd(1, x), igcd(-1, x)
```

$$\mathrm{igcd}(4420, 128, 8984, 488, x), 1, 1$$

```
type(igcd(a, x))
```

"igcd"

Parameters

i1, i2, …

arithmetical expressions representing integers or arithmetical expressions representing complex numbers of the domain `DOM_COMPLEX`, of which both the real part and the imaginary part are integers.

Return Values

Nonnegative integer, a complex number both the real and imaginary parts of which are integers, or a symbolic `igcd` call.

See Also

MuPAD Functions

`content` | `div` | `divide` | `factor` | `gcd` | `gcdex` | `icontent` | `ifactor` | `igcdex` | `ilcm` | `lcm` | `mod`

igcdex

Extended Euclidean algorithm for two integers

Syntax

```
igcdex(x, y)
```

Description

`igcdex(x, y)` computes the nonnegative greatest common divisor `g` of the integers `x` and `y` and integers `s` and `t` such that $g = sx + ty$.

`igcdex(x, y)` returns an expression sequence `g, s, t` with three elements, where `g` is the nonnegative greatest common divisor of `x` and `y` and `s`, `t` are integers such that $g = sx + ty$. These data are computed by the extended Euclidean algorithm for integers.

`igcdex(0, 0)` returns the sequence `0, 1, 0`. If `x` is non-zero, then `igcdex(0, x)` and `igcdex(x, 0)` return `abs(x), 0, sign(x)` and `abs(x), sign(x), 0`, respectively.

If both `x` and `y` are non-zero integers, then the numbers `s,t` satisfy the inequalities $|s| < \left|\frac{y}{g}\right|$ and $|t| < \left|\frac{x}{g}\right|$.

If one of the arguments is a number but not an integer, then `igcdex` returns an error message. If some argument is not a number, then `igcdex` returns a symbolic `igcdex` call.

The function `numlib::igcdmult` is an extension of `igcdex` for more than two arguments.

Examples

Example 1

We compute the greatest common divisor of some integers:

```
igcdex(-10, 6)
```

$$2, 1, 2$$

```
igcdex(3839882200, 654365735423132432848652680)
```

$$109710920, -681651885490791809, 4$$

The returned numbers satisfy the described equation:

```
[g, s, t] := [igcdex(9, 15)];
g = s*9 + t*15
```

$$[3, 2, -1]$$

$$3 = 3$$

If one argument is not a number, the result is the a symbolic `igcdex` call:

```
delete x:
igcdex(4, x)
```

$$\text{igcdex}(4, x)$$

Parameters

x, y

arithmetical expressions representing integers

Return Values

Sequence of three integers, or a symbolic `igcdex` call.

See Also

MuPAD Functions

`div` | `divide` | `factor` | `gcd` | `gcdex` | `ifactor` | `igcd` | `ilcm` | `lcm` | `mod` | `numlib::igcdmult`

ilaplace

Inverse Laplace transform

Syntax

```
ilaplace(F, s, t)
```

Description

`ilaplace(F, s, t)` computes the inverse Laplace transform of the expression `F = F(s)` with respect to the variable `s` at the point `t`.

The inverse Laplace transform can be defined by a contour integral in the complex plane:

$$f(t) = \frac{1}{2\pi i}\int_{c-i\infty}^{c+i\infty} F(s)\, e^{s t}\, ds$$

,

where c is a suitable complex number.

If `ilaplace` cannot find an explicit representation of the transform, it returns an unevaluated function call. See “Example 3” on page 1-992.

If `F` is a matrix, `ilaplace` applies the inverse Laplace transform to all components of the matrix.

To compute the direct Laplace transform, use `laplace`.

Examples

Example 1

Compute the inverse Laplace transforms of these expressions with respect to the variable `s`:

```
ilaplace(1/(a + s), s, t)
```

$$e^{-a\,t}$$

```
ilaplace(1/(s^3 + s^5), s, t)
```

$$\cos(t) + \frac{t^2}{2} - 1$$

```
ilaplace(exp(-2*s)/(s^2 + 1) + s/(s^3 + 1), s, t)
```

$$\text{heaviside}(t-2)\,\sin(t-2) - \frac{e^{-t}}{3} + \frac{e^{\frac{t}{2}}\left(\cos\left(\frac{\sqrt{3}\,t}{2}\right) + \sqrt{3}\,\sin\left(\frac{\sqrt{3}\,t}{2}\right)\right)}{3}$$

Example 2

Compute the inverse Laplace transform of this expression with respect to the variable `s`:

```
f := ilaplace(1/(1 + s)^2, s, t)
```

$$t\,e^{-t}$$

Evaluate the inverse Laplace transform of the expression at the points $t = -2\,t_0$ and $t = 1$. You can evaluate the resulting expression `f` using `|` (or its functional form `evalAt`):

```
f | t = -2*t_0
```

$$-2\,t_0\,e^{2\,t_0}$$

Also, you can evaluate the inverse Laplace transform at a particular point directly:

```
ilaplace(1/(1 + s)^2, s, 1)
```

$$e^{-1}$$

Example 3

If `laplace` cannot find an explicit representation of the transform, it returns an unevaluated call:

```
ilaplace(1/(1 + sqrt(t)), t, s)
```

$$\text{ilaplace}\left(\frac{1}{\sqrt{t}+1}, t, s\right)$$

`laplace` returns the original expression:

```
laplace(%, s, t)
```

$$\frac{1}{\sqrt{t}+1}$$

Example 4

Compute this inverse Laplace transform. The result is the Dirac function:

```
ilaplace(1, s, t)
```

$$\delta(t)$$

Parameters

F

Arithmetical expression or matrix of such expressions

s

Identifier or indexed identifier representing the transformation variable

t

Arithmetical expression representing the evaluation point

Return Values

Arithmetical expression or unevaluated function call of type `ilaplace`. If the first argument is a matrix, then the result is returned as a matrix.

Overloaded By

F

See Also

MuPAD Functions

`ilaplace::addpattern` | `laplace` | `laplace::addpattern`

ilaplace::addpattern

Add patterns for the inverse Laplace transform

Syntax

```
ilaplace::addpattern(pat, s, t, res, <vars, <conds>>)
```

Description

`ilaplace::addpattern(pat, s, t, res)` teaches `ilaplace` to return *ilaplace*(*pat*, *s*, *t*) = *res*.

The `ilaplace` function uses a set of patterns for computing inverse Laplace transforms. You can extend the set by adding your own patterns. To add a new pattern to the pattern matcher, use `ilaplace::addpattern`. MuPAD does not save custom patterns permanently. The new patterns are available in the *current* MuPAD session only.

Variable names that you use when calling `ilaplace::addpattern` can differ from the names that you use when calling `ilaplace`. See “Example 2” on page 1-995.

You can include a list of free parameters and a list of conditions on these parameters. These conditions and the result are protected from premature evaluation. This means that you can use `not iszero(a^2 - b)` instead of `hold( _not @ iszero )(a^2 - b)`.

The following conditions treat assumptions on identifiers differently:

- `a^2 - b <> 0` takes into account assumptions on identifiers.
- `not iszero(a^2 - b)` disregards assumptions on identifiers.

See “Example 4” on page 1-997.

Environment Interactions

Calling `ilaplace::addpattern` changes the expressions returned by future calls to `ilaplace`.

Examples

Example 1

Compute the inverse Laplace transform of the function `bar`. By default, MuPAD does not have a pattern for this function:

```
ilaplace(bar(s), s, t)
```

$$\text{ilaplace}(\text{bar}(s), s, t)$$

Add a pattern for the inverse Laplace transform of `bar` using `ilaplace::addpattern`:

```
ilaplace::addpattern(bar(s), s, t, foo(t)):
```

Now `ilaplace` returns the inverse Laplace transform of `bar`:

```
ilaplace(bar(s), s, t)
```

$$\text{foo}(t)$$

After you add a new transform pattern, MuPAD can use that pattern indirectly:

```
ilaplace(exp(-s)*bar(s), s, t)
```

$$\text{foo}(t-1)\ \text{heaviside}(t-1)$$

Example 2

Define the inverse Laplace transform of `bar(y)` using the variables `x` and `y` as parameters:

```
ilaplace::addpattern(bar(y), y, x, foo(x)):
```

The `ilaplace` function recognizes the added pattern even if you use other variables as parameters:

```
ilaplace(bar(s), s, t)
```

$$\mathrm{foo}(t)$$

Example 3

Add this pattern for the inverse Laplace transform of F:

```
ilaplace::addpattern(F(c, S)*G(c, S), S, T, T/(T^4 + 4*c^4)):
ilaplace(F(c, s)*G(c, s), s, t)
```

$$\frac{t}{4\,c^4 + t^4}$$

This pattern holds only when the first argument of F is the symbolic parameter c. If you use any other value of this parameter, `ilaplace` ignores the pattern:

```
ilaplace(F(A, s)*G(A, s), s, t)
```

$$\mathrm{ilaplace}(F(A, s)\,G(A, s), s, t)$$

To use the pattern for arbitrary values of the parameter, declare the parameter c as an additional pattern variable:

```
ilaplace::addpattern(F(c, S)*G(c, S), S, T, T/(T^4 + 4*c^4), [c]):
```

Now `ilaplace` applies the specified pattern for an arbitrary value of c:

```
ilaplace(F(C, s)*G(C, s), s, t)
```

$$\frac{t}{4\,C^4 + t^4}$$

You also can declare several parameters as pattern variables. For example, this pattern has two pattern variables, a and b:

```
ilaplace::addpattern(f(a*y + b), y, x, g(x/a - b), [a, b]):
ilaplace(f(2*s + B), s, t)
```

$$g\left(\frac{t}{2} - B\right)$$

Example 4

Use assumptions when adding this pattern for the inverse Laplace transform:

```
ilaplace::addpattern(BAR(x*s), s, t, sin(1/(x - 1/2))*FOO(t),
                                        [x], [abs(x) < 1]):
ilaplace(BAR(x*s), s, t) assuming -1 < x < 1
```

$$\sin\left(\frac{1}{x-\frac{1}{2}}\right)\,\mathrm{FOO}(t)$$

If $|x| \geq 1$, you cannot apply this pattern:

```
ilaplace(BAR(x*s), s, t) assuming x >= 1
```

$$\mathrm{ilaplace}(\mathrm{BAR}(s\,x),\,s,\,t)$$

If MuPAD cannot determine whether the conditions are satisfied, it returns a `piecewise` object:

```
ilaplace(BAR(x*s), s, t)
```

$$\begin{cases} \sin\left(\frac{1}{x-\frac{1}{2}}\right)\,\mathrm{FOO}(t) & \text{if } |x| < 1 \end{cases}$$

Note that the resulting expression defining the inverse Laplace transform of `BAR(x*s)` implicitly assumes that the value of `x` is not 1/2. A strict definition of the pattern is:

```
ilaplace::addpattern(BAR(x*t), s, t, sin(1/(x - 1/2))*FOO(t),
                     [x], [abs(x) < 1, x <> 1/2]):
```

If either the conditions are not satisfied or substituting the values into the result gives an error, `ilaplace` ignores the pattern. For this particular pattern, you can omit specifying the assumption `x <> 1/2`. If `x = 1/2`, MuPAD throws an internal "Division by zero." error and ignores the pattern:

```
ilaplace(BAR(s/2), s, t)
```

$$\mathrm{ilaplace}\left(\mathrm{BAR}\left(\frac{s}{2}\right),\,s,\,t\right)$$

Parameters

pat

Arithmetical expression in the variable `s` representing the pattern to match.

s

Identifier or indexed identifier used as a variable in the pattern

t

Identifier or indexed identifier used as a variable in the result

res

Arithmetical expression in the variable `t` representing a pattern for the result

vars

List of identifiers or indexed identifiers used as "pattern variables" (placeholders in `pat` and `res`). You can use pattern variables as placeholders for almost arbitrary MuPAD expressions not containing `s` or `t`. You can restrict them by conditions given in the optional parameter `conds`.

conds

List of conditions on the pattern variables

Return Values

Object of type `DOM_NULL`

See Also

MuPAD Functions

`ilaplace` | `laplace` | `laplace::addpattern`

ilcm

Least common multiple of integers

Syntax

```
ilcm(i1, i2, …)
```

Description

`ilcm(i1, i2, ...)` computes the least common multiple of the integers i_1, i_2, …

`ilcm` computes the least common nonnegative multiple of a sequence of integers. `ilcm` with a single numeric argument returns its absolute value. `ilcm` returns `1` when all arguments are `1` or `-1` or no argument is given.

`ilcm` returns an error message when one of the arguments is a number but not an integer. If at least one of the arguments is `0`, then `ilcm` returns `0`. Otherwise, if one argument is not a number, then a symbolic `ilcm` call is returned.

Examples

Example 1

We compute the least common multiple of some integers:

```
ilcm(-10, 6), ilcm(6, 10, 15)
```

```
30, 30
```

```
a := 4420, 128, 8984, 488:
ilcm(a), ilcm(a, 64)
```

```
9689064320, 9689064320
```

The next example shows some special cases:

```
ilcm(), ilcm(0), ilcm(1), ilcm(-1), ilcm(2)
```

$$1, 0, 1, 1, 2$$

If one argument is not a number, then the result is a symbolic `ilcm` call, except in some special cases:

```
delete x:
ilcm(a, x), ilcm(0, x)
```

$$\mathrm{ilcm}(4420, 128, 8984, 488, x), 0$$

```
type(ilcm(a, x))
```

$$\text{"ilcm"}$$

Parameters

`i1i2, …`

arithmetical expressions representing integers

Return Values

Nonnegative integer, or a symbolic `ilcm` call.

See Also

MuPAD Functions

`content` | `factor` | `gcd` | `gcdex` | `icontent` | `ifactor` | `igcd` | `igcdex` | `lcm`

in, _in

Membership

Syntax

```
x in set
```

```
_in(x, set)
```

```
for y in object do body end_for
```

```
f(y) $ y in object
```

Description

`x in set` is the MuPAD notation for the statement "*x* is a member of `set`."

In conjunction with one of the keywords `for` or `$`, the meaning changes to "iterate over all operands of the object". See `for` and `$` for details. Cf. "Example 6" on page 1-1004.

Apart from the usage with `for` and `$`, the statement `x in object` is equivalent to the function call `_in(x, object)`.

`x in set` is just evaluated to itself. `expand(x in set)` tries to return an equivalent expression without using the operator `in`, as described in the following paragraphs.

For sets of type `DOM_SET`, set unions, differences and intersections, `x in set` is expanded to an equivalent Boolean expression of equations and expressions involving `in`. Cf. "Example 1" on page 1-1002.

If `set` is a solution set of a single equation in one unknown, given by a symbolic call to `solve`, expanding `in` returns a Boolean condition that is equivalent to `x` being a solution. Cf. "Example 2" on page 1-1002.

If `set` is a `RootOf` expression, expanding `in` returns a Boolean condition that is equivalent to `x` being a root of the corresponding equation. Cf. "Example 3" on page 1-1003.

The function `bool` and every function that uses boolean evaluation can also handle many logical expressions involving `in`. Cf. “Example 4” on page 1-1003.

The function `is` handles various logical statements involving `in`, including a variety of types for the parameter `set` which are not handled by `in` itself. Cf. “Example 5” on page 1-1004 for a few typical cases.

Apart from the usual overloading mechanism by the first argument of an `in` call, `in` can be overloaded by its second argument, too. This argument must define the slot`"set2expr"` for this purpose. The slot will be called with the arguments `set, x`.

Examples

Example 1

`x in {1, 2, 3}` is expanded into an equivalent statement involving = and `or`:

```
expand(x in {1, 2, 3})
```

$$x = 1 \vee x = 2 \vee x = 3$$

The same happens if you replace `x` by a number, because Boolean expressions are only evaluated inside certain functions such as `bool` or `is`:

```
expand(1 in {1, 2, 3}), bool(1 in {1, 2, 3}), is(1 in {1, 2, 3})
```

$$1 = 1 \vee 1 = 2 \vee 1 = 3,\ \text{TRUE},\ \text{TRUE}$$

If only some part of the expression can be simplified this way, the returned expression can contain unevaluated calls to `in`:

```
expand(x in {1, 2, 3} union A)
```

$$x \in A \vee x = 1 \vee x = 2 \vee x = 3$$

Example 2

For symbolic calls to `solve` representing the solution set of a single equation in one unknown, `in` can be used to check whether a particular value lies in the solution set:

```
solve(cos(x) + x = cos(2) + 2, x);
expand(2 in %), bool(2 in %)
```

$$\text{solve}(x + \cos(x) = \cos(2) + 2, x)$$

$$\cos(2) + 2 = \cos(2) + 2, \text{TRUE}$$

Example 3

`in` can be used to check whether a value is a member of the solution set represented by a `RootOf` expression:

```
r := RootOf(x^2 - 1, x);
expand(1 in r), bool(1 in r), expand(2 in r), bool(2 in r)
```

$$\text{RootOf}(x^2 - 1, x)$$

$$0 = 0, \text{TRUE}, 3 = 0, \text{FALSE}$$

```
expand((y - 1) in RootOf(x^2 - 1 - y^2 + 2*y, x))
```

$$2y + (y - 1)^2 - y^2 - 1 = 0$$

```
expand(%)
```

$$0 = 0$$

```
delete r:
```

Example 4

Expressions with operator `in` are boolean expressions: they can be used like equations or inequalities.

```
if 2 in {2, 3, 5} then "ok" end
```

"ok"

Example 5

The MuPAD function `is` can investigate membership of objects in infinite sets. It respects properties of identifiers:

```
is(123 in Q_), is(2/3 in Q_)
```

$$\text{TRUE, TRUE}$$

Example 6

In conjunction with `for` and `$`, `y in object` iterates y over all operands of the object:

```
for y in [1, 2] do
   print(y)
end_for:
```

$$1$$

$$2$$

```
y^2 + 1 $ y in a + b*c + d^2
```

$$a^2+1,\ b^2\,c^2+1,\ d^4+1$$

```
delete y:
```

Parameters

`x`

An arbitrary MuPAD object

`set`

A set or an object of set-like type

y

An identifier or a local variable (DOM_VAR) of a procedure

object, f(y)

Arbitrary MuPAD objects

Return Values

`x in set` just returns the input.

Overloaded By

set, x

See Also

MuPAD Functions

`_seqin` | `bool` | `contains` | `for` | `has` | `is`

indets

Indeterminates of an expression

Syntax

```
indets(object)
indets(object, <All>)
indets(object, <PolyExpr>)
indets(object, <RatExpr>)
```

Description

`indets(object)` returns the indeterminates contained in `object`.

`indets(object)` returns the indeterminates of `object` as a set, i.e., the identifiers without a value that occur in `object`, with the exception of those identifiers occurring in the `0`th operand of a subexpression of `object` (see “Example 1” on page 1-1006).

`indets` regards the special identifiers `PI`, `EULER`, `CATALAN` as indeterminates, although they represent constant real numbers. If you want to exclude these special identifiers, use `indets(object) minus Type::ConstantIdents` (see example “Example 1” on page 1-1006).

If `object` is a polynomial, a function environment, a procedure, or a built-in kernelfunction, then `indets` returns the empty set. See “Example 2” on page 1-1008.

Examples

Example 1

Consider the following expression:

```
delete g, h, u, v, x, y, z:
```

```
e := 1/(x[u] + g^h) - f(1/3) + (sin(y) + 1)^2*PI^3 + z^(-3)*v^(1/2)
```

$$\frac{\sqrt{v}}{z^3} - f\left(\frac{1}{3}\right) + \frac{1}{g^h + x_u} + \pi^3\,(\sin(y) + 1)^2$$

```
indets(e)
```

$$\{\pi, g, h, u, v, x, y, z\}$$

Note that the returned set contains `x` and `u` and not, as one might expect, `x[u]`, since internally `x[u]` is converted into the functional form `_index(x, u)`. Moreover, the identifier `f` is not considered an indeterminate, since it is the `0`th operand of the subexpression `f(1/3)`.

Although `PI` mathematically represents a constant, it is considered an indeterminate by `indets`. Use `Type::ConstantIdents` to circumvent this:

```
indets(e) minus Type::ConstantIdents
```

$$\{g, h, u, v, x, y, z\}$$

The result of `indets` is substantially different if one of the two options `RatExpr` or `PolyExpr` is specified:

```
indets(e, RatExpr)
```

$$\left\{\pi, z, \sin(y), g^h, x_u, \sqrt{v}\right\}$$

Indeed, `e` is a rational expression in the "indeterminates" `z, PI, sin(y), g^h, x[u], v^(1/2)`: `e` is built from these atoms and the constant expression `f(1/3)` by using only the rational operations `+`, `-`, `*`, `/`, and `^` with integer exponents. Similarly, `e` is built from `PI,sin(y),z^(-3),1/(g^h+x[u]),v^(1/2)` and the constant expression `f(1/3)` using only the polynomial operations `+`, `-`, `*`, and `^` with nonnegative integer exponents:

```
indets(e, PolyExpr)
```

$$\left\{\pi,\ \sin(y),\ \frac{1}{g^h + x_u},\ \sqrt{v},\ \frac{1}{z^3}\right\}$$

Example 2

`indets` also works for various other data types. Polynomials and functions are considered to have no indeterminates:

```
delete x, y:
indets(poly(x*y, [x, y])), indets(sin), indets(x -> x^2+1)
```

$$\varnothing,\ \varnothing,\ \varnothing$$

For container objects, `indets` returns the union of the indeterminates of all entries:

```
indets([x, exp(y)]), indets([x, exp(y)], PolyExpr)
```

$$\{x, y\},\ \{x, e^y\}$$

For tables, only the indeterminates of the entries are returned; indeterminates in the indices are ignored:

```
indets(table(x = 1 + sin(y), 2 = PI))
```

$$\{\pi, y\}$$

Example 3

In the previous examples we saw that the `0`th operand of a subexpression is not used for finding indeterminates. With the option `All` this is changed:

```
delete x: e := sin(x):
indets(e, All)
```

$$\{\sin, x\}$$

A more complex example:

```
delete g, h, u, v, y, z:
e := 1/(x[u] + g^h) - f(1/3) + (sin(y) + 1)^2*PI^3 + z^(-3)*v^(1/2)
```

$$\frac{\sqrt{v}}{z^3} - f\left(\frac{1}{3}\right) + \frac{1}{g^h + x_u} + \pi^3\,(\sin(y)+1)^2$$

```
indets(e,All)
```

$$\{\pi,\ \_index,\ \_mult,\ \_plus,\ \_power,\ f,\ g,\ h,\ \sin,\ u,\ v,\ x,\ y,\ z\}$$

```
delete e:
```

Parameters

`object`

An arbitrary object

Options

`All`

Identifiers occurring in the `0`th operand of a subexpression of `object` are also included in the result.

With this option, the `0`th operand of a subexpression is not excluded from the search for indeterminates of `object`. So if the `0`th operand of a subexpression is a indeterminate e.g. like `sin` it is included in the result, Cf. “Example 3” on page 1-1008.

`PolyExpr`

Return a set of arithmetical expressions such that `object` is a polynomial expression in the returned expressions

With this option, `object` is considered as a polynomial expression. Non-polynomial subexpressions, such as `sin(x)`, `x^(1/3)`, `1/(x+1)`, or `f(a, b)`, are considered as indeterminates and are included in the returned set. However, subexpressions such

as `f(2, 3)` are considered as constants even when the identifier `f` has no value. The philosophy behind this is that the expression is constant because the operands are constant (see “Example 1” on page 1-1006).

If `object` is an array, a list, a set, or a table, then `indets` returns a set of arithmetical expressions such that each entry of `object` is a polynomial expression in these expressions. See “Example 2” on page 1-1008.

`RatExpr`

Return a set of arithmetical expressions such that `object` is a rational expression in the returned expressions

With this option, `object` is considered as a rational expression. Similar to `PolyExpr`, non-rational subexpressions are considered as indeterminates (see “Example 1” on page 1-1006).

Return Values

set of arithmetical expressions.

Overloaded By

object

Algorithms

If `object` is an element of a library domain`T` that has a slot `"indets"`, then the slot routine `T::indets` is called with `object` as argument. This can be used to extend the functionality of `indets` to user-defined domains. If no such slot exists, then `indets` returns the empty set.

See Also

MuPAD Functions

`collect` | `domtype` | `op` | `poly` | `rationalize` | `type` | `Type::Indeterminate` | `Type::PolyExpr` | `Type::RatExpr`

[], _index

Indexed access

Syntax

```
x[ i ]

x[ i1,i2,... ]

x[ i1..i2 ]

x[ [i1,i2,...] ]

x[ [i1,i2,...], [k1,k2,...] ]

_index( x, i )

_index( x, i1,i2,... )

_index( x, i1..i2 )

_index( x, [i1,i2,...] )

_index( x, [i1,i2,...], [k1,k2,...] )
```

Description

`_index` is a functional form of the `[]` operator. The calls `x[...]` and `_index(x,...)` are equivalent.

`x[i]` returns the entry of `x` corresponding to the index `i`. Any MuPAD object `x` allows an indexed call of the form `x[i]`. If `x` is not a container object (such as sets, lists, vectors, arrays, hfarrays, tables, matrices), then `x[i]` returns a symbolic indexed object. In particular, if `x` is an identifier, then `x[i]` returns an "indexed identifier". In this case, indices can be arbitrary MuPAD objects. See "Example 1" on page 1-1013.

`x[i1,i2,...]` returns the entries of `x` corresponding to the indices `i1,i2,...`, specified as a sequence. For example, if `x` is a matrix, then `x[2, 3]` returns the third element of the second row of `x`.

`x[i1..i2]` returns the entries of `x` corresponding to integer indices in the range `[i1..i2]`, including `x[i1]` and `x[i2]`. In particular, this applies to lists, sets, expression sequences, and strings. See "Example 7" on page 1-1016.

`x[[i1,i2,...]]` returns the entries of `x` corresponding to the specified indices, given as a list of integers. Here, `x` must be a list, matrix, or vector.

- If `x` is a list, then `x[[i1,i2,...]]` returns the list `[x[i] $ i in [i1,i2,...]]`. See "Example 8" on page 1-1018.
- If `x` is a row vector, then `x[[i1,i2,...]]` returns the row vector `matrix(1, nops([i1,i2,...]), [x[i] $ i in [i1,i2,...]])`.
- If `x` is a column vector, then `x[[i1,i2,...]]` returns the column vector `matrix(nops([i1,i2,...]), 1, [x[i] $ i in [i1,i2,...]])`.

`x[[i1,i2,...],[k1,k2,...]]` returns the matrix `matrix([[x[i,k] $ k in [k1,k2,...]] $ i in [i1,i2,...]])`. Here, `x` must be a matrix. See "Example 9" on page 1-1018.

Depending on the type of `x`, these restrictions apply to the indices:

- For lists, finite sets, or expression sequences, the index `i` can only be an integer from `1` to `nops(x)`, or `-nops(x)` to `-1`, or a range of these numbers.
- For arrays and hfarrays, use appropriate indices `i` or multiple indices `i1,i2,...` from the index range defined by `array` or `hfarray`. Integers outside this range cause an error. If any specified index is not an integer (for example, a symbol `i`), then `x[i]` or `x[i1,i2,...]` is returned symbolically.
- For matrices, use appropriate indices `i` or double indices `i1,i2` from the index range defined by `matrix`. Indices outside this range or symbolic indices cause an error.
- For tables, you can use any object as an index. If there is no corresponding entry in the table, then `x[i]` or `x[i1,i2,...]` is returned symbolically.
- For character strings, the index `i` must be an integer from `1` to `length(x)`.

`_index` uses the order in which the entries appear on the screen, and `op` uses the internal order of the entries. For some container objects, these orders differ. In particular:

- For lists and sequences, `x[i] = op(x,i)` for positive indices. For negative indices, `x[i] = op(x, nops(x) + 1 + i)`.

- For finite sets, `x[i]` returns the `i`th element as printed on the screen. Before screen output and indexed access, the elements of sets are sorted via the slot `DOM_SET::sort`. In general, `x[i] <> op(x, i)` for finite sets `x`.
- For one-dimensional arrays `x := array(1..n, [...])` or `x := hfarray(1..n, [...])`, the entries correspond to the operands, `x[i] = op(x,i)`.
- For a one-dimensional matrix representing a column vector, `x[i] = x[i, 1] = op(x, i)`. For a one-dimensional matrix representing a row vector, `x[i] = x[1, i] = op(x, i)`.

The entry returned by an indexed call is fully evaluated. For lists, matrices, arrays, and tables, you can suppress evaluation in indexed calls by using `indexval`. See "Example 10" on page 1-1018.

Indexed access to expressions and numbers is implemented via library callbacks. Do not use `_index` in program files to access the operands of expressions and numbers. Use `op` instead for more efficiency.

If `x` is not a container object (such as sets, lists, vectors, arrays, hfarrays, tables, matrices), then indexed assignments (such as `x[i] := value`) implicitly convert `x` into a table with a single entry.

Examples

Example 1

Solve these equations specifying variables as indexed identifiers:

```
n := 4:
equations := {x[i-1] - 2*x[i] + x[i+1] = 1 $ i = 1..n}:
unknowns := {x[i] $ i = 1..n}:
linsolve(equations, unknowns)
```

$$\left[x_1 = \frac{4\,x_0}{5} + \frac{x_5}{5} - 2,\ x_2 = \frac{3\,x_0}{5} + \frac{2\,x_5}{5} - 3,\ x_3 = \frac{2\,x_0}{5} + \frac{3\,x_5}{5} - 3,\ x_4 = \frac{x_0}{5} + \frac{4\,x_5}{5} - 2\right]$$

Symbolic indexed objects are of type `"_index"`:

```
type(x[i])
```

```
"_index"
```

```
delete n, equations, unknowns:
```

Example 2

Use indices to access the entries of typical container objects, such as lists, arrays, hardware floating-point arrays, and tables:

```
L := [1, 2, [3, 4]]:
A := array(1..2, 2..3, [[a12, a13], [a22, a23]]):
B := hfarray(1..2, 2..3, [[12.0, 13.0], [22.0, 23.0]]):
T := table(1 = T1, x = Tx, (1, 2) = T12):
```

```
L[1], L[3][2], A[2, 3], B[2, 3], T[1], T[x], T[1, 2]
```

$$1,\ 4,\ a23,\ 23.0,\ T1,\ Tx,\ T12$$

Use indexed assignments to change the entries:

```
L[2]:= 22: L[3][2]:= 32: A[2, 3]:= 23: B[2, 3]:= 0: T[x]:= T12:
L, A, B, T
```

$$[1,\ 22,\ [3,\ 32]],\ \begin{pmatrix} a12 & a13 \\ a22 & 23 \end{pmatrix},\ \begin{pmatrix} 12.0 & 13.0 \\ 22.0 & 0.0 \end{pmatrix},\ \begin{array}{r|l} 1 & T1 \\ x & T12 \\ 1,\ 2 & T12 \end{array}$$

```
delete L, A, B, T:
```

Example 3

For finite sets, an indexed call `x[i]` returns the `i`th element as printed on the screen. This element does not necessarily coincide with the `i`th (internal) operand returned by `op`:

```
S := {3, 2, 1}
```

$$\{1,\ 2,\ 3\}$$

```
S[i] $ i = 1..3
```

$$1, 2, 3$$

```
op(S, i) $ i = 1..3
```

$$3, 2, 1$$

```
delete S:
```

Example 4

The index operator also operates on character strings. The characters are enumerated starting from 1:

```
"ABCDEF"[1], "ABCDEF"[6]
```

$$\text{"A"}, \text{"F"}$$

Example 5

The index operator also operates on mathematical expressions containing operators, such as +, -, *, and so on:

```
X := a - b + c - 2;
X[2], X[3..-1];
delete X:
```

$$a - b + c - 2$$

$$-b, c - 2$$

For expressions with `_plus`- and `_mult` operators, the output of `_index` corresponds to the output order of the operands. If an expression with `_mult` is printed as a fraction, you can access the nominator and the denominator via indices 1 and 2:

```
X := ((a/2 + b) * c * 2)/(e-f)/x^2;
```

```
X[1], X[2], X[1][3];
delete X:
```

$$\frac{2\,c\left(\frac{a}{2}+b\right)}{x^2\,(e-f)}$$

$$2\,c\left(\frac{a}{2}+b\right),\ x^2\,(e-f),\ \frac{a}{2}+b$$

Example 6

The index operator also operates on rational and complex numbers. For rational numbers, index 1 refers to the numerator, and index 2 refers to the denominator:

```
(2/3)[1], (2/3)[2]
```

$$2,\ 3$$

For complex numbers, indices 1 and 2 refer to the real and imaginary parts, respectively:

```
(3*I)[1], (1-I)[2]
```

$$0,\ -1$$

Example 7

You can use a range as an index. For lists, sets, expression sequences, and strings, this operation returns a “subexpression” consisting of the entries within the range, according to `_index`:

```
L := [1, 2, 3, 4, 5]:
S := {1, 2, 3, 4, 5}:
Str := "abcde":

L[3..4];
S[3..4];
Str[3..4]
```

```
[3, 4]
```

```
{3, 4}
```

```
"cd"
```

This includes ranges with negative numbers. When you use negative indices `i` for container type objects `x`, such as lists and sets, the call `x[i]` returns `x[nops(x) + 1 + i]`. Thus, you access elements counting indices from the end of `x`: the index `-1` refers to the last element of `x`, the index `-2` refers to the second element from the end, and so on.

```
L[3..-1];
S[1..-2]
```

```
[3, 4, 5]
```

```
{1, 2, 3, 4}
```

When you use negative indices `i` for strings, `_index` internally replaces `Str[i]` with indices `op(Str, i + 1 + length(Str))`:

```
Str[3..-1]
```

```
"cde"
```

You also can use this form of indexing to assign values to elements of lists and strings:

```
L[2..4] := [234]: L;
Str[3..-1] := " ??": Str;
```

```
[1, 234, 5]
```

```
"ab ??"
```

As seen above, this operation can change the number of elements in a list or the length of a string. If necessary, new places are filled with `NIL` or spaces, respectively:

```
L[42..42] := [42]: L;
Str[10..11] := "the end.": Str
```

[1, 234, 5, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, 42]

"ab ?? the end."

Example 8

Use the following indexed call to return a permutation of the list `L`. Here, the index list `perm` specifies the permutation.

```
L := [a, b, c, d, e]:
perm := [5, 3, 1, 2, 4]:
L[perm]
```

$[e, c, a, b, d]$

Example 9

Use two lists of indices to pick and return a particular submatrix of a matrix:

```
A := matrix([[a11, a12, a13], [a21, a22, a23], [a31,a32, a33]]):
l1 := [1,2]: l2 := [2,3]:
A[l1, l2]
```

$$\begin{pmatrix} a12 & a13 \\ a22 & a23 \end{pmatrix}$$

Example 10

Indexed calls evaluate the returned entry. Use `indexval` to suppress full evaluation:

```
delete a:
x := [a, b]: a := c:
```

```
x[1] = eval(x[1]), x[1] <> indexval(x, 1)
```

$$c = c,\ c \neq a$$

```
delete a:
x := table(1 = a, 2 = b): a := c:
x[1] = eval(x[1]), x[1] <> indexval(x, 1)
```

$$c = c,\ c \neq a$$

```
delete a:
x := array(1..2, [a, b]): a := c:
x[1] = eval(x[1]), x[1] <> indexval(x, 1)
```

$$c = c,\ c \neq a$$

```
delete a: x := matrix([a, b]): a := c:
x[1] = eval(x[1]), x[1] <> indexval(x, 1)
```

$$c = c,\ c \neq a$$

```
delete x, a:
```

Example 11

Indexed access is not implemented for some kernel domains:

```
12343[3]
```

```
Error: The operand is invalid. [_index]
```

Define a method implementing the indexed access to integer numbers:

```
unprotect(DOM_INT):
DOM_INT::_index := (n, i) -> text2expr(expr2text(n)[i]):
12343[3];
delete DOM_INT::_index: protect(DOM_INT):
```

```
3
```

Parameters

x

An arbitrary MuPAD object. In particular, a container object: a list, a finite set, an array, an hfarray, a matrix, a table, an expression sequence, an expression in operator notation, a rational number, a complex number, or a character string.

i

An index. For most container objects x, indices must be integers. If x is a table, you can use arbitrary MuPAD objects as indices.

i1,i2,...

Multiple indices for matrices and multidimensional arrays. For these containers, the indices must be integers. For tables, you can also use multiple indices given by arbitrary MuPAD objects.

i1..i2

Indices, specified as a range.

[i1,i2,...]

Indices, specified as a list of integers. In this case, x must be a list, matrix, or vector. (In MuPAD, a vector is a 1#n or n×1 matrix.)

[i1,i2...], [k1,k2...]

Indices, specified as two lists of integers. In this case, x must be a matrix.

Return Values

Entry of x corresponding to the index. Calls with lists of indices can return a sequence, a list, a vector, or a matrix of entries corresponding to the indices. If x is not a list, a set,

an array, or any other container object, then the return value is an indexed object of type `"_index"`.

Overloaded By

x

See Also

MuPAD Domains

DOM_ARRAY | DOM_HFARRAY | DOM_LIST | DOM_SET | DOM_STRING | DOM_TABLE

MuPAD Functions

:= | _assign | array | contains | hfarray | indexval | op | slot | table | Type::Indeterminate

indexval

Indexed access to arrays and tables without evaluation

Syntax

```
indexval(x, i)
```

```
indexval(x, i1, i2, …)
```

Description

`indexval(x, i)` and `indexval(x, i1, i2, ...)` yields the entry of `x` corresponding to the indices `i` and `i1, i2, ...`, respectively, without evaluation.

The three calls `indexval(x, i)`, `_index(x, i)`, and `x[i]` all return the element of index `i` in the array or hfarray or list or table `x`. In contrast to `_index` and the equivalent index operator `[ ]`, however, `indexval` returns the corresponding entry without evaluating it. This is sometimes desirable for efficiency reasons.

The arguments `i` or i1, i2, ... must be a valid indices of `x`, otherwise an error message is printed (see “Example 3” on page 1-1024). When several indices i1, i2, ... are given, they are interpreted as a higher-dimensional index (see “Example 4” on page 1-1025).

The first argument `x` may also be a set, a string, or an expression sequence. However, in these cases `indexval` behaves exactly like `_index` and the index operator `[ ]`: it returns the evaluation of the corresponding element. In particular, `indexval` does not flatten its first argument.

For all other basic domains, `indexval` behaves exactly like `_index`: either an error occurs, or a symbolic `indexval` call is returned (see “Example 3” on page 1-1024).

Examples

Example 1

`indexval` works with tables:

```
T := table("1" = a, Be = b, `+` = a + b):
a := 1: b := 2:
indexval(T, Be), indexval(T, "1"), indexval(T, `+`)
```

$$b,\ a,\ a+b$$

In contrast `_index` evaluates returned entries:

```
_index(T, Be), _index(T, "1"), _index(T, `+`)
```

$$2,\ 1,\ 3$$

The next input line has the same meaning as the last:

```
T[Be], T["1"], T[`+`]
```

$$2,\ 1,\ 3$$

`indexval` works with arrays, too. The behavior is the same, but the indices must be positive integers:

```
delete a, b:
A := array(1..2, 1..2, [[a, a + b], [a - b, b]]):
a := 1: b := 2:
indexval(A, 2, 2), indexval(A, 1, 1), indexval(A, 1, 2)
```

$$b,\ a,\ a+b$$

```
_index(A, 2, 2), _index(A, 1, 1), _index(A, 1, 2)
```

$$2,\ 1,\ 3$$

```
A[2, 2], A[1, 1], A[1, 2]
```

$$2,\ 1,\ 3$$

```
delete A, T, a, b:
```

`indexval` works lists, too:

```
delete a, b:
L := [a, b, 2]:
b := 5:
L[2], _index(L, 2), indexval(L, 2), op(L, 2)
```

$$5, 5, b, 5$$

Example 2

However, there is no difference between `indexval` and `_index` for all other valid objects, e.g., sets:

```
delete a, b:
S := {a, b, 2}:
b := 5:
S[2], _index(S, 2), indexval(S, 2), op(S, 2)
```

$$5, 5, 5, 5$$

Similarly, there is no difference when the first argument is an expression sequence (which is not flattened by `indexval`):

```
delete a, b: S := a, b, 2:
b := 5:
S[2], _index(S, 2), indexval(S, 2), op(S, 2)
```

$$5, 5, 5, 5$$

```
delete L, S, a, b:
```

Example 3

If the second argument is not a valid index, an error occurs:

```
A := array(1..2, 1..2, [[a, b], [a, b]]):
indexval(A, 3)
```

```
Error: Index dimension mismatch. [array]
```

```
indexval(A, 1, 0)
```

```
Error: The argument is invalid. [array]
```

```
indexval("12345", 6)
```

```
Error: The index is invalid. [string]
```

However, the result of `indexval` can also be a symbolic `indexval` call:

```
T := table(1 = a, 2 = b):
indexval(T, 3)
```

$$\text{indexval}(T, 3)$$

```
delete X, i:
indexval(X, i)
```

$$\text{indexval}(X, i)$$

```
delete A, T:
```

Example 4

For arrays the number of indices must be equal to the number of dimensions of the array:

```
A := array(1..2, 1..2, [[a, b], [a, b]]):
a := 1: b := 2:
indexval(A, 1, 2), indexval(A, 2, 1)
```

$$b, a$$

Otherwise an error occurs:

```
indexval(A, 1)
```

```
Error: Index dimension mismatch. [array]
```

Tables can have expression sequences as indices, too:

```
delete a, b:
```

```
T := table((1, 1) = a, (2, 2) = b):
a := 1: b := 2:
indexval(T, 1, 1), indexval(T, 2, 2)
```

$$a, b$$

```
delete A, T, a, b:
```

Parameters

x

Essentially a table, a list, or an array. Also allowed: a hfarray, a finite set, an expression sequence, or a character string

i, i1, i2, …

Indices. For most “containers” x, indices must be integers. If x is a table, arbitrary MuPAD objects can be used as indices.

Return Values

Entry of x corresponding to the index. When x is a table, a list or an array, the returned entry is not evaluated again.

Overloaded By

x

See Also

MuPAD Domains
DOM_ARRAY | DOM_HFARRAY | DOM_LIST | DOM_SET | DOM_STRING | DOM_TABLE

MuPAD Functions
:= | _assign | _index | array | contains | op | table

infinity

Real positive infinity

Syntax

```
infinity
```

Description

`infinity` represents the infinite point on the positive real semi-axis.

`infinity` is an element of the domain `stdlib::Infinity`. It may be used in arithmetical operations. Some system functions accept `infinity` as a parameter or return it as a result.

Examples

Example 1

`infinity` can be used in arithmetical operations with real numbers:

```
7*infinity + 3, -3.0*infinity, 1/infinity,
infinity*infinity, infinity^2, sqrt(infinity)
```

$$\infty,\ -\infty,\ 0,\ \infty,\ \infty,\ \infty$$

Arithmetic with complex numbers or symbolic objects yields symbolic expressions:

```
I*infinity + b
```

$$i\,\infty + b$$

The arithmetic responds to properties:

```
assume(a > 0): a*infinity
```

∞

```
assume(a < 0): a*infinity
```

$-\infty$

```
unassume(a): a*infinity
```

$a\,\infty$

Cancellation of infinities yields `undefined`:

```
infinity - infinity, infinity/infinity
```

undefined, undefined

Some system functions accept `infinity` as a parameter or return it as result:

```
exp(infinity), sum(1/n, n = 1..infinity),
int(exp(-x^2), x = -infinity..infinity),
limit(x, x = infinity)
```

$\infty, \infty, \sqrt{\pi}, \infty$

See Also

MuPAD Functions

`complexInfinity` | `undefined`

info

Prints short information

Syntax

```
info(object)

info()
```

Description

`info(object)` prints short information about `object`.

`info` prints a short descriptive information about `object`.

If `object` is a domain, additional information is given about the methods of the domain.

A call to `info` without arguments prints a reference to a random help page.

Users can add information about their own functions and domains by overloading`info`. If `object` is a user-defined domain or function environment providing a slot`"info"`, whose value is a string, then the call `info(object)` prints this string. See “Example 2” on page 1-1030.

Examples

Example 1

With `info()`, you obtain a reference to a random help page:

```
info()


-- Help page of the day:
```

```
?input
```

The next example shows information about the library property:

```
info(property)
```

```
Library 'property': properties of identifiers

-- Interface:

property::depends, property::hasprop,
```

`info` prints information about preferences:

```
info(Pref::autoPlot)
```

```
Automatically plot graphical objects instead of typesetting
```

If no more information is available, a short type description is given:

```
info(a + b):
info([a, b]):
```

```
a + b -- an expression of type "_plus"

[a, b] -- of domain type 'DOM_LIST'
```

Example 2

`info` prints information about a function environment:

```
info(sqrt)
```

```
sqrt -- the square root
```

`sqrt` is a function environment and has a slot named `"info"`:

```
domtype(sqrt), sqrt::info
```

```
DOM_FUNC_ENV, "sqrt -- the square root"
```

User-defined procedures can contain short information. By default, `info` does only return some general information:

```
f := x -> x^2:  info(f):
```

```
f(x) -- a procedure of domain type 'DOM_PROC'
```

To improve this, we embed the function `f` into a function environment and store an information string in its `"info"` slot:

```
f := funcenv(f):
f::info := "f -- the squaring function":
info(f)
```

```
f -- the squaring function
```

```
delete f:
```

Parameters

`object`

Any MuPAD object

Return Values

Void object `null()` of type `DOM_NULL`.

Algorithms

If the argument `object` of `info` is a domain, then the call `info(object)` first prints the entry `"info"`, which must be a string. Then the entry `"interface"`, which must be a set of identifiers, is used to display all public methods, and the entry `"exported"`,

which is a set of identifiers created by `export::stl`, is used to display all exported methods.

See Also

MuPAD Functions

`help` | `print`

input

Interactive input of objects

Syntax

```
input(<prompt1>)
```

```
input(<prompt1>, x1, <prompt2>, x2, …)
```

Description

`input` allows interactive input of MuPAD objects.

`input()` displays the prompt "`Please enter expression:`" and waits for input by the user. The input, terminated by pressing the Return key, is parsed and returned *unevaluatedly*.

`input(prompt1)` uses the character string `prompt1` instead of the default prompt "`Please enter expression:`".

`input( prompt1 x1)` assigns the input to the identifier or local variable `x1`. The default prompt is used, if no prompt string is specified.

Several objects can be read with a single `input` command. Each identifier or variable in the sequence of arguments makes `input` return a prompt, waiting for input to be assigned to it. A character string preceding an identifier or variable in the argument sequence replaces the default prompt (see "Example 2" on page 1-1035). Arguments that are neither prompt strings nor identifiers or variables are ignored.

The identifiers or variables `x1` etc. may have values. These are overwritten by `input`.

`input` only parses the input objects for syntactical correctness. It does not evaluate them. Use `eval` to evaluate the results (see "Example 3" on page 1-1035).

Examples

Example 1

The default prompt is displayed. The input is returned without evaluation:

```
input()

Please enter expression: << 1 + 2 >>

     1 + 2
```

A character string is used as a prompt:

```
input("enter a number: ")

enter a number: << 5
>>

     5
```

The input may be assigned to an identifier:

```
input(x)

Please enter expression: << 5 >>

     5

x

     5
```

A user-defined prompt is used, the input is assigned to an identifier:

```
input("enter a number: ", x)

enter a number: << 6
>>
```

```
    6

x

    6

delete x:
```

Example 2

If several objects are to be read, for each object a separate prompt can be defined:

```
input("enter a matrix: ", A, "enter a vector: ", x)

enter a matrix: << matrix([[a11,
a12], [a21, a22]]) >>

enter
a vector: << matrix([x1, x2]) >>

matrix([x1, x2])

A, x
```

$$\begin{pmatrix} a11 & a12 \\ a21 & a22 \end{pmatrix}, \begin{pmatrix} x1 \\ x2 \end{pmatrix}$$

```
delete A, x:
```

Example 3

The following procedure asks for an expression and a variable. After interactive input, the derivative of the expression with respect to the variable is computed:

```
interactiveDiff :=
  proc()
    local f, x;
  begin
     f := input("enter an expression: ");
     x := input("enter an identifier: ");
     print(Unquoted, "The derivative of " . expr2text(f) .
```

```
           " with respect to ". expr2text(x) . " is:");
     diff(f, x)
  end_proc:

interactiveDiff()

enter an expression: <<
x^2 + x*y^3 >>

enter an identifier:
<< x >>

 The derivative
of x^2 + x*y^3 with respect to x is:
```

$$2x + y^3$$

The function `input` does not evaluate the input. This leads to the following unexpected result:

```
f := x^2 + x*y^3:
z := x:
interactiveDiff()

enter
an expression: << f >>

enter
an identifier: << z >>

 The
derivative of f with respect to z is:
```

$$0$$

The following modification enforces full evaluation via `eval`:

```
interactiveDiff :=
  proc()
    local f, x;
  begin
     f := eval(input("enter an expression: "));
     x := eval(input("enter an identifier: "));
     print(Unquoted, "The derivative of " . expr2text(f) .
```

```
          " with respect to ". expr2text(x) . " is:");
    diff(f, x)
  end_proc:

interactiveDiff()

enter an expression: <<
f >>

enter an identifier:
<< z >>

 The derivative
of x^2 + x*y^3 with respect to x is:

```

$$2x + y^3$$

```
delete interactiveDiff, f, z:
```

Parameters

prompt1, prompt2, …

Input prompts: character strings

x1, x2, …

identifiers or local variables

Return Values

Last input

See Also

MuPAD Functions

finput | fprint | fread | ftextinput | import::readbitmap | import::readdata | print | read | text2expr | textinput | write

int

Definite and indefinite integrals

Syntax

```
int(f, x)
```

```
int(f, x = a .. b, options)
```

Description

`int(f, x)` computes the indefinite integral $\int f(x)\,\mathrm{d}x$.

`int(f, x = a..b)` computes the definite integral $\int_a^b f(x)\,\mathrm{d}x$.

`int(f, x)` determines a function F such that $\frac{\partial}{\partial x}F(x) = f(x)$. The function $F(x)$ is called the antiderivative of $f(x)$. Results returned by `int` do not include integration constants.

For indefinite integrals, `int` implicitly assumes that the integration variable `x` is real. For definite integrals, `int` restricts the integration variable `x` to the specified integration interval `[a, b]` of the type `Type::Interval`. If one or both integration bounds `a` and `b` are not numeric, `int` assumes that `a <= b` unless you explicitly specify otherwise. By default, `int` does not issue warnings about these assumptions. To display the warnings about using implicit assumptions, set the value of `intlib::printWarnings` to `TRUE`.

In general, the result of `int` is not required to be valid for all complex values of `x`. For example, the identity $\ln(e^x) = x$ is only valid for real values of x. Therefore, $\int \ln(e^x)\,\mathrm{d}x = \frac{x^2}{2}$ is also valid only for real values of x.

You can specify your own assumptions of the integration variable. If these assumptions do not conflict with the default assumption that the variable is real or with the integration interval, then MuPAD uses your assumptions. Otherwise, the `int` function issues a warning and changes the assumptions. Use `intlib::printWarnings` to enable or disable the warnings.

If you compute an indefinite integral and specify properties of the integration variable that describe a subset of the real numbers, MuPAD assumes that the variable is real. Otherwise, the system uses temporary assumption that the integration variable is complex. This assumption holds only during this particular integration.

If you compute a definite integral and specify properties that conflict with the integration interval, `int` uses the integration interval.

`int` can return results with discontinuities even if the integrand is continuous.

Integration techniques, such as table lookup or Risch integration for an indefinite integral, can add new discontinuities during the integration process. These new discontinuities appear because the antiderivatives can require the introduction of complex logarithms. Complex logarithms have a jump discontinuity when the argument crosses the negative real axis, and the integration algorithms sometimes cannot find a representation where these jumps cancel.

If you compute a definite integral by first computing an indefinite integral and then substituting the integration boundaries into the result, remember that indefinite integration can produce discontinuities. If it does, you must investigate the discontinuities in the integration interval.

If MuPAD cannot find a closed-form solution for the integral and cannot prove that such form does not exist, it returns an unresolved integral. In this case, you can approximate the integral numerically or try computing a series expansion of the integral. See “Example 2” on page 1-1041 and “Example 3” on page 1-1042.

You can approximate a definite integral numerically using `numeric::int` or `float`. Numeric approximation of a definite integral only works when the `float` function can convert the boundaries `a` and `b` of the integration interval to floating-point numbers. See “Example 2” on page 1-1041.

`int` might not find a closed form of a definite integral because of singularities of the integrand in the interval of integration. If the integral does not exist in a strict mathematical sense, `int` returns the value `undefined`. In this case, try using assumptions. Alternatively, use the `PrincipalValue` option to compute a weaker form of a definite integral called the Cauchy principal value. This form of an integral can exist even though the standard integral value is undefined. See “Example 6” on page 1-1043.

In general, the derivative of the result coincides with *f* on a dense subset of the real numbers (or, if you use assumptions on the integration variable, the subset of real numbers specified by these assumptions).

It is not always possible to decide algorithmically if $\frac{\partial}{\partial x} F(x)$ and f are equivalent. The reason is the so-called zero equivalence problem, which in general is undecidable.

Environment Interactions

`int` is sensitive to properties of identifiers set by `assume`. See “Example 6” on page 1-1043.

Examples

Example 1

Compute the indefinite integrals $\int \frac{1}{x \ln(x)}\, dx$ and $\int \frac{1}{x^2 - 8}\, dx$:

```
int(1/x/ln(x), x)
```

$$\ln(\ln(x))$$

```
int(1/(x^2 - 8), x)
```

$$-\frac{\sqrt{2}\,\operatorname{arctanh}\left(\frac{\sqrt{2}\,x}{4}\right)}{4}$$

Compute the definite integral $\frac{1}{(x \ln(x))}$ over the interval $[e, e^2]$:

```
int(1/x/ln(x), x = exp(1)..exp(2))
```

$$\ln(2)$$

When computing definite integrals, you can use infinities as the boundaries of the integration interval:

```
int(exp(-x^2), x = 0..infinity)
```

$$\frac{\sqrt{\pi}}{2}$$

You can compute multiple integrals. For example, compute the following definite multiple integral:

```
int(int(int(1, z = 0..c*(1 - x/a - y/b)),
        y = 0..b*(1 - x/a)), x = 0..a)
```

$$\frac{a\,b\,c}{6}$$

Example 2

Use `int` to compute this definite integral. Since `int` cannot find a closed form of this integral, it returns an unresolved integral:

```
S := int(sin(cos(x)), x = 0..1)
```

$$\int_0^1 \sin(\cos(x))\,dx$$

Use the `float` function to approximate the integral numerically:

```
float(S)
```

$$0.738642998$$

Alternatively, use the `numeric::int` function, which is faster because it does not involve any symbolic preprocessing:

```
numeric::int(sin(cos(x)), x = 0..1)
```

$$0.738642998$$

Example 3

Use `int` to compute this indefinite integral. Since `int` cannot find a closed form of this integral, it returns an unresolved integral:

```
int((x^2 + 1)/sqrt(sqrt(x + 1) + 1), x)
```

$$\int \frac{x^2+1}{\sqrt{\sqrt{x+1}+1}}\,dx$$

Use the `series` function to compute a series expansion of the integral:

```
series(%, x = 0)
```

$$\frac{\sqrt{2}\,x}{2} - \frac{\sqrt{2}\,x^2}{32} + \frac{45\sqrt{2}\,x^3}{256} - \frac{161\sqrt{2}\,x^4}{8192} + \frac{2507\sqrt{2}\,x^5}{327680} - \frac{12647\sqrt{2}\,x^6}{3145728} + O(x^7)$$

Alternatively, compute a series expansion of the integrand, and then integrate the result. This approach is faster because it does not try to integrate the original expression. It integrates an approximation (the series expansion) of the original expression:

```
int(series((x^2 + 1)/sqrt(sqrt(x + 1) + 1), x = 0), x)
```

$$\frac{\sqrt{2}\,x}{2} - \frac{\sqrt{2}\,x^2}{32} + \frac{45\sqrt{2}\,x^3}{256} - \frac{161\sqrt{2}\,x^4}{8192} + \frac{2507\sqrt{2}\,x^5}{327680} - \frac{12647\sqrt{2}\,x^6}{3145728} + O(x^7)$$

Example 4

The `IgnoreAnalyticConstraints` option applies a set of purely algebraic simplifications including the equality of sum of logarithms and a logarithm of a product. Using this option, you get a simpler result, but one that might be incorrect for some of the values of the variables:

```
int(ln(x) + ln(y) - ln(x*y), x, IgnoreAnalyticConstraints)
```

$$0$$

Without using this option, you get the following result, which is valid for all values of the parameters:

```
int(ln(x) + ln(y) - ln(x*y), x)
```

$$x\,(\ln(x) - \ln(x\,y) + \ln(y))$$

The results obtained with `IgnoreAnalyticConstraints` might be not generally valid:

```
f := int(ln(x) + ln(y) - ln(x*y), x):
g := int(ln(x) + ln(y) - ln(x*y), x, IgnoreAnalyticConstraints):
simplify([f, g]) assuming x = -1 and y = -1
```

$$[-2\,\pi\,i,\ 0]$$

Example 5

By default, `int` returns this integral as a piecewise object where every branch corresponds to a particular value (or a range of values) of the symbolic parameter `t`:

```
int(x^t, x)
```

$$\begin{cases} \ln(x) & \text{if } t = -1 \\ \frac{x^{t+1}}{t+1} & \text{if } t \neq -1 \end{cases}$$

To ignore special cases of parameter values, use `IgnoreSpecialCases`:

```
int(x^t, x, IgnoreSpecialCases)
```

$$\frac{x^{t+1}}{t+1}$$

Example 6

Compute this definite integral, where the integrand has a pole in the interior of the interval of integration. Mathematically, this integral is not defined:

```
int(1/(x - 1), x = 0..2)
```

$$\text{undefined}$$

However, the Cauchy principal value of the integral exists. Use the `PrincipalValue` option to compute the Cauchy principal value of the integral:

```
hold(int)(1/(x - 1), x = 0..2, PrincipalValue) =
     int( 1/(x - 1), x = 0..2, PrincipalValue)
```

$$\int_0^2 \frac{1}{x-1}\,dx = 0$$

For integrands with parameters, `int` might be unable to decide if the integrand has poles in the interval of integration. In this case, `int` returns a piecewise-defined function or an unresolved integral:

```
int(1/(x - a), x = 0..2)
```

$$\begin{cases} \int_0^2 \frac{1}{x-a}\,dx & \text{if } a \in [0, 2] \\ \ln(2-a) - \ln(-a) & \text{if } a < 0 \vee 2 < a \vee a \notin \mathbb{R} \end{cases}$$

`int` does not call simplification functions for its results. To simplify results returned by `int`, use `eval`, `simplify`, or `Simplify`:

```
Simplify(eval(%))
```

$$\{ \ln(2-a) - \ln(-a) \quad \text{if } a < 0 \vee 2 < a \vee a \notin \mathbb{R}$$

The resulting piecewise expression has only one branch. If the parameter `a` does not satisfy this condition, the integral is `undefined`.

Parameters

`f`

The integrand: an arithmetical expression representing a function in `x`

`x`

The integration variable: an identifier

a, b

The boundaries: arithmetical expressions

Options

IgnoreAnalyticConstraints

When you use this option, `int` applies these simplifications rules to the integrand:

- $ln(a) + ln(b) = ln(a\,b)$ for all values of a and b. In particular:

  $(a\,b)^c = e^{c\,\ln(a\,b)} = e^{c\,(\ln(a)+\ln(b))} = a^c\,b^c$ for all values of a, b, and c
- $ln(a^b) = b\,ln(a)$ for all values of a and b. In particular:

  $(a^b)^c = e^{b\,c\,\ln(a)} = e^{\ln(a)^{b\,c}} = a^{b\,c}$ for all values of a, b, and c
- If f and g are standard mathematical functions and $f(g(x)) = x$ for all small positive numbers, $f(g(x)) = x$ is assumed to be valid for all complex x. In particular:
- - $\ln(e^x) = x$
  - $arcsin(sin(x)) = x$, $arccos(cos(x)) = x$, $arctan(tan(x)) = x$
  - $arcsinh(sinh(x)) = x$, $arccosh(cosh(x)) = x$, $arctanh(tanh(x)) = x$
  - $W_k(x\,e^x) = x$ for all values of k

Using this option, you can get simpler solutions for some integrals for which the direct call of the integrator returns complicated results. With this option the integrator does not verify the correctness and completeness of the result. See “Example 4” on page 1-1042.

IgnoreSpecialCases

If integration requires case analysis, ignore cases that require one or more parameters to be elements of a comparatively small set, such as a fixed finite set or a set of integers.

With this option, `int` tries to reduce the number of branches in piecewise objects. MuPAD finds equations and memberships in comparatively small sets. First, MuPAD tries to prove such equations and memberships by using the property mechanism. If the property mechanism proves an equation or a membership is true, MuPAD keeps that statement. Otherwise, MuPAD can replace that statement with the value FALSE.

For example, if the property mechanism cannot prove that a denominator is equal to zero, MuPAD regards this denominator as nonzero. This option can significantly reduce the number of piecewise objects in the result.

See “Example 5” on page 1-1043.

`PrincipalValue`

Compute the Cauchy principal value of the integral.

If the interior of the integration interval contains poles of the integrand or the boundaries are $a = -\infty$ and $b = \infty$, then the definite integral might not exist in a strict mathematical sense. However, if the integrand changes sign at all poles in the integration interval, you can compute a weaker form of a definite integral called the *Cauchy principal value*. In this form, the so-called infinite parts of the integral to the left and to the right of a pole cancel each other. When you use the `PrincipalValue` option, `int` computes the Cauchy principal value. If the definite integral exists in a strict mathematical sense, it coincides with the Cauchy principal value. See “Example 6” on page 1-1043.

Return Values

arithmetical expression

Overloaded By

f

References

[1] Bronstein, M. “A Unification of Liouvillian Extension.” AAECC Applicable Algebra in Engineering, Communication and Computing. 1: 5–24, 1990.

[2] Bronstein, M. “The Transcendental Risch Differential Equation.” Journal of Symbolic Computation. 9: 49–60, 1990.

[3] Bronstein, M. “Symbolic Integration I: Transcendental Functions.” Springer. 1997.

[4] Epstein, H. I. and B. F. Caviness. “A Structure Theorem for the Elementary Functions and its Application to the Identity Problem.” International Journal of Computer and Information Science. 8: 9–37, 1979.

[5] Fakler, W. “Vereinfachen von komplexen Integralen reeller Funktionen.” mathPAD 9 No. 1: 5-9, 1999.

[6] Geddes, K. O., S. R. Czapor and G. Labahn. “Algorithms for Computer Algebra.” 1992.

See Also

MuPAD Functions

`D` | `diff` | `limit` | `numeric::int` | `sum`

int::addpattern

Add patterns for integration

Syntax

```
int::addpattern(pat, x, res, <[var, …], <[cond, …]>>)
```

```
int::addpattern(pat, x = u .. v, res, <[var, …], <[cond, …]>>)
```

Description

`int::addpattern(pat, x, res)` teaches `int` to make use of $\frac{\partial}{\partial x}\,\mathrm{res} = \mathrm{pat}$.

`int::addpattern(pat, x=u..v, res)` teaches `int` that $\int_u^v \mathrm{pat}\,\mathrm{d}x = \mathrm{res}$.

A large part of a computer algebra system's integration abilities stems from mathematical pattern matching. The MuPAD pattern matcher can be extended at runtime with `int::addpattern`.

Unless further limited by conditions in the fifth argument, pattern variables listed in the fourth argument represent arbitrary MuPAD expressions not containing the variable of integration, `x`.

Any identifier can be used as the variable of integration in a call to `int::addpattern`, and any identifier can be used in calls to `int`. They need not be identical.

For definite integration, each integration bound is either an arithmetical expression which may contain pattern variables, or an identifier which can be used as a variable in the result and condition terms.

Users can include additional conditions by giving additional arguments. These conditions, as well as the result, are protected from premature evaluation, i.e., it is not necessary to write `hold( _not @ iszero )(a^2-b)`, a simple `not iszero(a^2-b)` suffices.

The difference between `not iszero(a^2-b)` and `a^2-b <> 0` when given as a condition is that the latter takes into account asuumptions on the identifiers encountered, while the first does not. Cf. “Example 4” on page 1-1051.

Patterns introduced by `int::addpattern` are also used in recursive calls of the integrator and are automatically extended to include simple applications of integration by change of variables. Cf. “Example 1” on page 1-1049.

Patterns added by `int::addpattern` are *not* replaced by later calls, they remain active. `int` selects the most simple result found. There is no way to remove patterns once added. Cf. “Example 5” on page 1-1052.

Environment Interactions

Calling `int::addpattern` changes the expressions returned by future calls to `int`. Additionally, the remembered values of previous calls to `int` are forgotten.

Examples

Example 1

Not surprisingly, MuPAD does not know how to integrate the function *foo*:

```
int(foo(x), x)
```

$$\int \mathrm{foo}(x)\,\mathrm{d}x$$

We add a pattern for this function:

```
int::addpattern(foo(x), x, foo(x)^x)
```

```
int(foo(x), x)
```

$$\mathrm{foo}(x)^x$$

Note that this pattern is also used indirectly:

```
int(x*foo(x^2), x)
```

$$\frac{\mathrm{foo}(x^2)^{x^2}}{2}$$

```
intlib::byparts(int(foo(x)*sin(x), x), foo(x))
```

$$\mathrm{foo}(x)^x \sin(x) - \int \cos(x)\, \mathrm{foo}(x)^x \, dx$$

Example 2

Definite integrals can be added similarly. Note that the result does not depend on the integration variable:

```
int::addpattern(wilma(x), x=0..1, fred)
```

```
int(wilma(x), x=0..1)
```

$$\mathrm{fred}$$

The above pattern will not match integrals with different integration bounds:

```
int(wilma(x), x=0..2)
```

$$\int_0^2 \mathrm{wilma}(x)\, dx$$

Integration bounds may also contain variables occurring in the pattern or result:

```
int::addpattern(wilma(x, a), x=0..a, fred(a), [a])
```

```
int(wilma(x,2), x=0..2)
```

$$\mathrm{fred}(2)$$

Example 3

The integration variable in the call to `int::addpattern` need not be the same as used in the integration call:

```
int::addpattern(1/(t^2*(ln(t)+1)), t, -E*Ei(ln(t)+1))
```

```
int(cos(y)/sin(y)^2/(ln(sin(y)) + 1), y)
```

$$-e\,\mathrm{Ei}(\ln(\sin(y))+1)$$

Example 4

Conditions are checked using `is` and therefore react to assumptions:

```
int::addpattern(1/(a+b*tan(x)^2), x,
                x/(a-b)
                - b/(2*(a-b)*sqrt(-a*b))
                 * ln((b*tan(x)-sqrt(-a*b))
                     /(b*tan(x)+sqrt(-a*b))),
                [a, b],
                [a*b < 0])

int::addpattern(1/(a+b*tan(x)^2), x,
                x/(a-b)
                - b/((a-b)*sqrt(a*b))
                 * arctan(b*tan(x)/
                          sqrt(a*b)),
                [a, b],
                [a*b > 0])

int(1/(3+a*tan(x)^2), x) assuming a > 0
```

$$-\frac{3\,x-\sqrt{3}\,\sqrt{a}\,\arctan\left(\frac{\sqrt{3}\,\sqrt{a}\,\tan(x)}{3}\right)}{3\,a-9}$$

```
int(1/(3+a*tan(x)^2), x) assuming a < 0
```

$$-\frac{x}{a-3}-\frac{\ln\left(\frac{\sqrt{-3\,a}-a\,\tan(x)}{\sqrt{-3\,a}+a\,\tan(x)}\right)\sqrt{-3\,a}}{6\,(a-3)}$$

If either the conditions are not satisfied or substituting the values into the result yields an error, the pattern is ignored. In the patterns above, the case $a = b$ causes a division by zero. There is no need to include a condition to guard against this case, though, MuPAD simply computes the integral as usual:

```
int(1/(3+3*tan(x)^2), x)
```

$$\frac{x}{6} + \frac{\sin(2\,x)}{12}$$

Example 5

Assume we have added the following pattern:

```
int::addpattern(f(x), x, f(x)^x):
```

Now, *f* is a pretty generic name, so we could later regard it as a different function and attempt to redefine its antiderivative:

```
int::addpattern(f(x), x, 1/sin(f(x))):

int(f(x), x)
```

$$f(x)^x$$

What happened?

As it turns out, `int::addpattern` has simply *added* the new pattern, and since $f(x)^x$ is considered "simpler" than $\frac{1}{\sin(f(x))}$, the result of the first pattern added is still returned.

This behavior is reasonable, since there may be multiple ways of representing an antiderivative and depending on parameter values, one or the other may be preferable:

```
int::addpattern(f(a, x), x, x*f1(a, x^a), [a]):
int::addpattern(f(a, x), x, x*f2(a, x^(1-a)), [a]):

int(f(0, x), x)
```

$$x\,\mathrm{f1}(0, 1)$$

```
int(f(1, x), x)
```

$$x\,\mathrm{f2}(1, 1)$$

```
int(f(a, x), x)
```

$x\ \mathrm{fl}(a, x^a)$

Parameters

`pat`

The pattern to match: an arithmetical expression in `x`.

`x`

The variable of integration: an `identifier`.

`u .. v`

The interval of integration for a definite integral: arithmetical expressions or identifiers.

`res`

The antiderivative pattern: an arithmetical expression

`[var, …]`

"pattern variables": placeholders in `pat` and `ret`, i.e., `identifiers` that do not represent themselves but almost arbitrary MuPAD expressions not containing `x` and restricted by the conditions in the fifth parameter.

`[cond, …]`

Conditions on the pattern variables

Return Values

Object of type `DOM_NULL`

See Also

MuPAD Functions

`int`

int2text

Convert an integer to a character string

Syntax

```
int2text(n, <b>)
```

Description

`int2text(n, b)` converts the integer `n` to a string that corresponds to the `b`-adic representation of `n`.

The string returned by `int2text` consists of the first `b` characters in 0, 1, …, 9, *A*, *B*, …, *Z*, *a*, *b*, …, *z*. For bases larger than 10, the letters represent the `b`-adic digits larger than 9: *A* = 10, *B* = 11, …, *Z* = 35, *a* = 36, *b* = 37, …, *z* = 61.

For the bases 2, 8, or 16, `int2text` provides the conversion from decimal representation to binary, octal, or hexadecimal representation, respectively.

`int2text` is the inverse of `text2int`.

Since the output of the numerical datatypes in MuPAD uses the decimal representation, strings are used by `int2text` to represent `b`-adic numbers. The function `numlib::g_adic` provides an alternative representation via lists.

Examples

Example 1

Relative to the default base 10, `int2text` provides a mere datatype conversion from `DOM_INT` to `DOM_STRING`:

```
int2text(123), int2text(-45678)
```

```
"123", "-45678"
```

Example 2

The decimal integer 32 has the following binary representation:

```
int2text(32, 2)
```

```
"100000"
```

The decimal integer 10^9 has the following hexadecimal representation:

```
int2text(10^9, 16)
```

```
"3B9ACA00"
```

... and with the base 62:

```
int2text(10^9, 62)
```

```
"15ftgG"
```

Example 3

Negative integers can be converted as well:

```
int2text(-15, 8)
```

```
"-17"
```

Parameters

n

An integer

b

The base: an integer between 2 and 62. The default base is 10.

Return Values

character string.

See Also

MuPAD Functions

`coerce` | `expr2text` | `genpoly` | `numlib::g_adic` | `tbl2text` | `text2expr` | `text2int` | `text2list` | `text2tbl`

interpolate

Polynomial interpolation

Syntax

```
interpolate(xList, yList, X, <F>)
```

```
interpolate(nodes, values, ind, <F>)
```

Description

`interpolate` computes an interpolating polynomial through data over a rectangular grid.

The call `interpolate(xList, yList, X)` with `xList = [x1, …, xn]` and `yList = [y1, …, yn]` returns the polynomial of degree less than n in the variable `X` which interpolates the points (x_1, y_1), …, (x_n, y_n).

This call with a 1-dimensional grid `xList` is equivalent to the corresponding 'multi-dimensional' call `interpolate([xList], array(1..n, [yList]), [X])`.

For d-dimensional interpolation, assume that indeterminates $ind = [X_1, \ldots, X_d]$ are specified. The interpolating polynomial $P = poly(\ldots, [X_1, \ldots, X_d], F)$ satisfies

$$\mathrm{evalp}\left(P, X_1 = L_1[i_1], \ldots, X_d = L_d[i_d]\right) = \mathrm{value}[i_1, \ldots, i_d]$$

for all points $\left[(L_1)_{i_1}, \ldots, L_d[i_d]\right]$ in the grid. P is the polynomial of minimal degree satisfying the interpolation conditions, i.e., `degree`$(P, X_i) < n_i$.

If only interpolating values at concrete numerical points $X_1 = v_1$, …, $X_d = v_d$ are required, we recommend not to compute P with symbolic indeterminates `ind` $= [X_1, \ldots, X_d]$ and then evaluate $P(v_1, \ldots, v_d)$. It is faster to compute this value directly by `interpolate` with `ind` $= [v_1, \ldots, v_d]$. Cf. examples "Example 1" on page 1-1058 and "Example 3" on page 1-1059.

Examples

Example 1

We consider a 1-dimensional interpolation problem. To each node x_i, a value y_i is associated. The interpolation polynomial P with $P(x_i) = y_i$ is:

```
xList := [1, 2, 3]:
yList := [y1, y2, y3]:
P := interpolate(xList, yList, X)
```

$$\text{poly}\left(\left(\frac{\text{y1}}{2} - \text{y2} + \frac{\text{y3}}{2}\right) X^2 + \left(-\frac{5\,\text{y1}}{2} + 4\,\text{y2} - \frac{3\,\text{y3}}{2}\right) X + 3\,\text{y1} - 3\,\text{y2} + \text{y3}, [X]\right)$$

The evaluation of P at the point $X = \frac{5}{2}$ is given by:

```
evalp(P, X = 5/2)
```

$$\frac{3\,\text{y2}}{4} - \frac{\text{y1}}{8} + \frac{3\,\text{y3}}{8}$$

This value can also be computed directly without the symbolic polynomial:

```
interpolate(xList,  yList, 5/2)
```

$$\frac{3\,\text{y2}}{4} - \frac{\text{y1}}{8} + \frac{3\,\text{y3}}{8}$$

```
delete xList, yList, P:
```

Example 2

We demonstrate multi-dimensional interpolation. Consider data over the following 2-dimensional 2 ×3 grid:

```
XList := [1, 2]: YList := [1, 2, 3]:
values := array(1..2, 1..3, [[1, 2, 3], [3, 2, 1]]):
P := interpolate([XList, YList], values, [X, Y])
```

$$\text{poly}(-2\,X\,Y+4\,X+3\,Y-4,\,[X,\,Y])$$

Next, interpolation over a 3-dimensional 2 ×3×2 grid is demonstrated:

```
L1 := [1, 2]: L2 := [1, 2, 3]: L3 := [1, 2]:
values := array(1..2, 1..3, 1..2,
   [[[1, 4], [1, 2], [3, 3]], [[1, 4], [1, 3], [4, 0]]]):
interpolate([L1, L2, L3], values, [X, Y, Z])
```

$$\text{poly}\left(-3\,X\,Y^2\,Z+\frac{7\,X\,Y^2}{2}+10\,X\,Y\,Z-\frac{23\,X\,Y}{2}-7\,X\,Z+8\,X+\frac{7\,Y^2\,Z}{2}-3\,Y^2-\frac{27\,Y\,Z}{2}\right.$$

$$\left.+12\,Y+13\,Z-11,\,[X,\,Y,\,Z]\right)$$

```
delete XList, values, P, L1, L2, L3:
```

Example 3

We interpolate data over a 2-dimensional grid:

```
n1 := 4: L1 := [i $ i = 1..n1]:
n2 := 5: L2 := [i $ i = 1..n2]:
f := (X, Y) -> 1/(1 + X^2 + Y^2):
values := array(1..n1, 1..n2,
                [[f(L1[i], L2[j]) $ j=1..n2] $ i=1..n1]):
```

First, we compute the symbolic polynomial:

```
P := interpolate([L1, L2], values, [X, Y])
```

$$\text{poly}\left(-\frac{5563\,X^3\,Y^4}{23108085}+\frac{16376\,X^3\,Y^3}{4621617}-\ldots-\frac{4401895\,Y}{3081078}+\frac{4199983}{2567565},\,[X,\,Y]\right)$$

```
poly(- (5563*X^3*Y^4)/23108085 + (16376*X^3*Y^3)/4621617 -
(176747*X^3*Y^2)/9243234 + (29440*X^3*Y)/660231 - (40922*X^3)/1100385 +
(22397*X^2*Y^4)/10270260 - (16691*X^2*Y^3)/513513 + (367151*X^2*Y^2)/2054052
- (220525*X^2*Y)/513513 + (645283*X^2)/1711710 - (1009*X*Y^4)/161595 +
(439024*X*Y^3)/4621617 - (452873*X*Y^2)/840294 + (6293720*X*Y)/4621617
```

```
- (1438453*X)/1100385 + (15199*Y^4)/2800980 - (263969*Y^3)/3081078 +
(452975*Y^2)/880308 - (4401895*Y)/3081078 + 4199983/2567565, [X, Y])
```

Fixing the value $Y = 2.5$, this yields a polynomial in X.

```
evalp(P, Y = 2.5)
```

$$\text{poly}(0.0007372500794\ X^3 - 0.002155538175\ X^2 - 0.03076935248\ X + 0.1533997618, [X])$$

It can also be computed directly by using an evaluation point for the indeterminate Y:

```
interpolate([L1, L2], values, [X, 2.5])
```

$$\text{poly}(0.0007372500794\ X^3 - 0.002155538175\ X^2 - 0.03076935248\ X + 0.1533997618, [X])$$

If all indeterminates are replaced by evaluation points, the corresponding interpolation value is returned:

```
interpolate([L1, L2], values, [1.2, 2.5])
```

$$0.114646532$$

```
delete n1, n2, f, values, P:
```

Example 4

We demonstrate interpolation over a special coefficient field. Consider the following data over a 2-dimensional 2 ×3 grid:

```
XList := [3, 4]: YList := [1, 2, 3]:
values := array(1..2, 1..3, [[0, 1, 2], [3, 2, 1]]):
```

With the following call, these data are converted to integers modulo 7. Arithmetic over this field is used:

```
F := Dom::IntegerMod(7):
P := interpolate([XList, YList], values, [X, Y], F)
```

$$\text{poly}(5\ X\ Y + 5\ X + 5, [X, Y], \text{Dom::IntegerMod}(7))$$

Evaluation of P at grid points reproduces the associated values converted to the field:

```
evalp(P, X = XList[2], Y = YList[3]) = F(values[2, 3])
```

$$1 \bmod 7 = 1 \bmod 7$$

```
delete XList, YList, values, F, P:
```

Parameters

xList

The nodes: a list `[x1, x2, ...]` of distinct arithmetical expressions

yList

The values: a list `[y1, y2, ...]` of arithmetical expressions. This list must have the same length as `xList`.

X

An indeterminate or an arithmetical expression. An indeterminate is either an identifier (of domain type `DOM_IDENT`) or an indexed identifier (of type `"_index"`).

nodes

A list $[L_1, \ldots, L_d]$ of d lists L_i defining a d-dimensional rectangular grid

$$\{(x_1, \ldots, x_d) \mid x_1 \in L_1, \ldots, x_d \in L_d\}.$$

The lists L_i may have different lengths $n_i = |L_i|$. The elements of each L_i must be distinct.

values

A d-dimensional `array(1..n[1],...,1..n[d], [...])` or `hfarray(1..n[1],...,1..n[d], [...])` associating a value with each grid point:

$$\left[(L_1)_{i_1}, \ldots, (L_d)_{i_d}\right] \to \text{values}[i_1, \ldots, i_d],\ i_1 = 1 \ldots n_1, \ldots, i_d = 1 \ldots n_d,$$

ind

A list of *d* indeterminates or arithmetical expressions. Indeterminates are either identifiers (of domain type `DOM_IDENT`) or indexed identifiers (of type `"_index"`).

F

Either `Expr` or any field of category `Cat::Field`

The returned polynomial is of type `poly(..., F)`.

For the default field `Expr`, all input data may be arbitrary MuPAD expressions. Standard arithmetic over such expressions is used to compute the polynomial.

For `F` not being `Expr`, the grid nodes as well as the entries of `values` must be elements of `F` or must be convertible to such elements. Conversion of the input data to elements of `F` is done automatically.

Return Values

Interpolating polynomial *P* of domain type `DOM_POLY` in the indeterminates specified by `ind` over the coefficient field `F` is returned. The elements in `ind` that are not indeterminates but arithmetical expressions are not used as indeterminates in *P*, but enter its coefficients: the polynomial is "evaluated" at these points. If no element of `ind` is an indeterminate, the value of the polynomial at the point specified by `ind` is returned. This is an element of the field `F` or an arithmetical expression if `F = Expr`.

Algorithms

For a *d*-dimensional rectangular grid

$$L_j = \left[x_{j,1}, \ldots, x_{j,n_j}\right], j = 1, \ldots, d$$

specified by the lists

$$\left\{(x_1, \ldots, x_d) \mid x_1 \in L_1, \ldots, x_d \in L_d\right\}$$

with associated values

$$P\left(x_{i_1}, \ldots, x_{i_d}\right) = v_{i_1, \ldots, i_d},$$

the interpolating polynomial in the indeterminates $X_1, \ldots, X_d$ is given by

$$P(X_1, \ldots, X_d) = \sum_{i_1=1}^{n_1} \cdots \sum_{i_d=1}^{n_d} v_{i_1, \ldots, i_d} \times p_{1, i_1}(X_1) \times p_{d, i_d}(X_d)$$

with the Lagrange polynomials

$$p_{j,k}(X) = \prod_{l=1}^{n_j} \frac{X - x_{j,l}}{x_{j,k} - x_{j,l}}, \; l \neq k, \; j = 1, \ldots, d, \; k = 1, \ldots, d$$

associated with the k-th node of the j-th coordinate.

See Also

MuPAD Functions

`genpoly` | `numeric::cubicSpline` | `numeric::cubicSpline2d` | `poly`

intersect, _intersect

Intersection of sets and/or intervals

Syntax

```
set1intersect set2

_intersect(set1, set2, …)
```

Description

`intersect` computes the intersection of sets and intervals.

`set1 intersect set2` is equivalent to `_intersect(set1, set2)`.

The precedences of `intersect`, `minus`, `union` are as follows: The operator `intersect` is stronger binding than `minus`, i.e, `set1 intersect set2 minus set3` = `(set 1 intersect set2) minus set3`. The operator `minus` is stronger binding than `union`, i.e., `set1 minus set2 union set3` = `(set1 minus set2) union set3`. Further, `set1 minus set2 minus set3` = `(set 1 minus set2) minus set3`. If in doubt, use brackets to make sure that the expression is parsed as desired.

If sets or intervals are specified by symbolic expressions involving identifiers or indexed identifiers, then symbolic calls of `_intersect`, `_minus`, `_union` are returned. On the screen, they are represented via the operator notation `set1 intersect set2` etc.

Note: On finite sets of type `DOM_SET`, these operators act in a purely *syntactical* way. E.g., `{1}minus {x}` simplifies to `{1}`. Mathematically, this result may not be correct in general, because `x` might represent the value 1.

On intervals of type `Dom::Interval`, these operators act in a *semantical* way. In particular, properties of identifiers are taken into account.

`_intersect()` returns `universe` (of type `stdlib::Universe`) which represents the set of all mathematical objects.

`_union()` returns the empty set `{}`.

Examples

Example 1

`intersect`, `minus`, and `union` operate on finite sets:

```
{x, 1, 5} intersect {x, 1, 3, 4},
{x, 1, 5} union {x, 1, 3, 4},
{x, 1, 5} minus {x, 1, 3, 4}
```

$$\{1, x\}, \{1, 3, 4, 5, x\}, \{5\}$$

For symbolic sets, specified as identifiers or indexed identifiers, symbolic calls are returned:

```
{1, 2} union A union {2, 3}
```

$$\{1, 2, 3\} \cup A$$

Note that the set operations act on finite sets in a purely syntactical way. In the following call, `x` does not match any of the numbers 1, 2, 3 syntactically:

```
{1, 2, 3} minus {1, x}
```

$$\{2, 3\}$$

Example 2

`intersect`, `minus`, and `union` are overloaded by the domain `Dom::Interval`:

```
Dom::Interval([0, 1]) union Dom::Interval(1, 4)
```

$$[0, 4)$$

```
Dom::Interval([0, 1]) union Dom::Interval(4, infinity)
```

$$[0, 1] \cup (4, \infty)$$

```
Dom::Interval(2, infinity) intersect Dom::Interval([1, 3])
```

$$(2, 3]$$

```
{PI/2, 2, 2.5, 3} intersect Dom::Interval(1,3)
```

$$\left\{2, 2.5, \frac{\pi}{2}\right\}$$

```
Dom::Interval(1, PI) minus {2, 3}
```

$$(1, 2) \cup (2, 3) \cup (3, \pi)$$

In contrast to finite sets of type `DOM_SET`, the interval domain works semantically. It takes `properties` into account:

```
Dom::Interval(-1, 1) minus {x}
```

$$(-1, 1) \setminus \{x\}$$

```
assume(x > 2): Dom::Interval(-1, 1) minus {x}
```

$$(-1, 1)$$

```
unassume(x):
```

Example 3

The following list provides a collection of sets:

```
L := [{a, b}, {1, 2, a, c}, {3, a, b}, {a, c}]:
```

The functional equivalent `_intersect` of the `intersect` operator accepts an arbitray number of arguments. Thus, the intersection of all sets in `L` can be computed as follows:

```
_intersect(op(L))
```

$$\{a\}$$

The union of all sets in `L` is:

```
_union(op(L))
```

$$\{1, 2, 3, a, b, c\}$$

```
delete L:
```

Example 4

`universe` represents the set of all mathematical objects:

```
_intersect()
```

$$\text{universe}$$

Parameters

$\mathtt{set}_1$, $\mathtt{set}_2$, …

Finite sets of type `DOM_SET`, or intervals of type `Dom::Interval`, or arithmetical expressions

Return Values

Set, an interval, a symbolic expression of type `"_intersect"`, `"_minus"`, `"_union"`, or `universe`.

Overloaded By

set1, set2

See Also

MuPAD Functions
`minus` | `subset` | `union` | `universe`

minus, _minus

Difference of sets and/or intervals

Syntax

```
set1minus set2
```

```
_minus(set1, set2)
```

Description

`minus` computes the difference between sets and intervals.

`set1 minus set2` is equivalent to `_minus(set1, set2)`.

The precedences of `intersect`, `minus`, `union` are as follows: The operator `intersect` is stronger binding than `minus`, i.e, `set1 intersect set2 minus set3` = `(set 1 intersect set2) minus set3`. The operator `minus` is stronger binding than `union`, i.e., `set1 minus set2 union set3` = `(set1 minus set2) union set3`. Further, `set1 minus set2 minus set3` = `(set 1 minus set2) minus set3`. If in doubt, use brackets to make sure that the expression is parsed as desired.

If sets or intervals are specified by symbolic expressions involving identifiers or indexed identifiers, then symbolic calls of `_intersect`, `_minus`, `_union` are returned. On the screen, they are represented via the operator notation `set1 intersect set2` etc.

Note: On finite sets of type `DOM_SET`, these operators act in a purely *syntactical* way. E.g., `{1}minus {x}` simplifies to `{1}`. Mathematically, this result may not be correct in general, because `x` might represent the value 1.

On intervals of type `Dom::Interval`, these operators act in a *semantical* way. In particular, properties of identifiers are taken into account.

`_intersect()` returns `universe` (of type `stdlib::Universe`) which represents the set of all mathematical objects.

`_union()` returns the empty set `{}`.

Examples

Example 1

`intersect`, `minus`, and `union` operate on finite sets:

```
{x, 1, 5} intersect {x, 1, 3, 4},
{x, 1, 5} union {x, 1, 3, 4},
{x, 1, 5} minus {x, 1, 3, 4}
```

$$\{1, x\}, \{1, 3, 4, 5, x\}, \{5\}$$

For symbolic sets, specified as identifiers or indexed identifiers, symbolic calls are returned:

```
{1, 2} union A union {2, 3}
```

$$\{1, 2, 3\} \cup A$$

Note that the set operations act on finite sets in a purely syntactical way. In the following call, `x` does not match any of the numbers 1, 2, 3 syntactically:

```
{1, 2, 3} minus {1, x}
```

$$\{2, 3\}$$

Example 2

`intersect`, `minus`, and `union` are overloaded by the domain `Dom::Interval`:

```
Dom::Interval([0, 1]) union Dom::Interval(1, 4)
```

$$[0, 4)$$

```
Dom::Interval([0, 1]) union Dom::Interval(4, infinity)
```

$$[0, 1] \cup (4, \infty)$$

```
Dom::Interval(2, infinity) intersect Dom::Interval([1, 3])
```

$$(2, 3]$$

```
{PI/2, 2, 2.5, 3} intersect Dom::Interval(1,3)
```

$$\left\{2, 2.5, \frac{\pi}{2}\right\}$$

```
Dom::Interval(1, PI) minus {2, 3}
```

$$(1, 2) \cup (2, 3) \cup (3, \pi)$$

In contrast to finite sets of type `DOM_SET`, the interval domain works semantically. It takes `properties` into account:

```
Dom::Interval(-1, 1) minus {x}
```

$$(-1, 1) \setminus \{x\}$$

```
assume(x > 2): Dom::Interval(-1, 1) minus {x}
```

$$(-1, 1)$$

```
unassume(x):
```

Example 3

The following list provides a collection of sets:

```
L := [{a, b}, {1, 2, a, c}, {3, a, b}, {a, c}]:
```

The functional equivalent `_intersect` of the `intersect` operator accepts an arbitray number of arguments. Thus, the intersection of all sets in `L` can be computed as follows:

```
_intersect(op(L))
```

$$\{a\}$$

The union of all sets in `L` is:

```
_union(op(L))
```

$$\{1, 2, 3, a, b, c\}$$

```
delete L:
```

Example 4

`universe` represents the set of all mathematical objects:

```
_intersect()
```

$$\text{universe}$$

Parameters

`set`$_1$`, set`$_2$`, …`

Finite sets of type `DOM_SET`, or intervals of type `Dom::Interval`, or arithmetical expressions

Return Values

Set, an interval, a symbolic expression of type `"_intersect"`, `"_minus"`, `"_union"`, or `universe`.

Overloaded By

set1, set2

See Also

MuPAD Functions
`intersect` | `subset` | `union` | `universe`

union, _union

Union of sets and/or intervals

Syntax

```
set1union set2
```

```
_union(set1, set2, …)
```

Description

`union` computes the union of sets and intervals.

`set1 union set2` is equivalent to `_union(set1, set2)`.

The precedences of `intersect`, `minus`, `union` are as follows: The operator `intersect` is stronger binding than `minus`, i.e, `set1 intersect set2 minus set3` = `(set 1 intersect set2) minus set3`. The operator `minus` is stronger binding than `union`, i.e., `set1 minus set2 union set3` = `(set1 minus set2) union set3`. Further, `set1 minus set2 minus set3` = `(set 1 minus set2) minus set3`. If in doubt, use brackets to make sure that the expression is parsed as desired.

If sets or intervals are specified by symbolic expressions involving identifiers or indexed identifiers, then symbolic calls of `_intersect`, `_minus`, `_union` are returned. On the screen, they are represented via the operator notation `set1 intersect set2` etc.

Note: On finite sets of type `DOM_SET`, these operators act in a purely *syntactical* way. E.g., `{1}minus {x}` simplifies to `{1}`. Mathematically, this result may not be correct in general, because `x` might represent the value 1.

On intervals of type `Dom::Interval`, these operators act in a *semantical* way. In particular, properties of identifiers are taken into account.

`_intersect()` returns `universe` (of type `stdlib::Universe`) which represents the set of all mathematical objects.

`_union()` returns the empty set `{}`.

Examples

Example 1

`intersect`, `minus`, and `union` operate on finite sets:

```
{x, 1, 5} intersect {x, 1, 3, 4},
{x, 1, 5} union {x, 1, 3, 4},
{x, 1, 5} minus {x, 1, 3, 4}
```

$$\{1, x\}, \{1, 3, 4, 5, x\}, \{5\}$$

For symbolic sets, specified as identifiers or indexed identifiers, symbolic calls are returned:

```
{1, 2} union A union {2, 3}
```

$$\{1, 2, 3\} \cup A$$

Note that the set operations act on finite sets in a purely syntactical way. In the following call, `x` does not match any of the numbers 1, 2, 3 syntactically:

```
{1, 2, 3} minus {1, x}
```

$$\{2, 3\}$$

Example 2

`intersect`, `minus`, and `union` are overloaded by the domain `Dom::Interval`:

```
Dom::Interval([0, 1]) union Dom::Interval(1, 4)
```

$$[0, 4)$$

```
Dom::Interval([0, 1]) union Dom::Interval(4, infinity)
```

$$[0, 1] \cup (4, \infty)$$

```
Dom::Interval(2, infinity) intersect Dom::Interval([1, 3])
```

$$(2, 3]$$

```
{PI/2, 2, 2.5, 3} intersect Dom::Interval(1,3)
```

$$\left\{2, 2.5, \frac{\pi}{2}\right\}$$

```
Dom::Interval(1, PI) minus {2, 3}
```

$$(1, 2) \cup (2, 3) \cup (3, \pi)$$

In contrast to finite sets of type `DOM_SET`, the interval domain works semantically. It takes `properties` into account:

```
Dom::Interval(-1, 1) minus {x}
```

$$(-1, 1) \setminus \{x\}$$

```
assume(x > 2): Dom::Interval(-1, 1) minus {x}
```

$$(-1, 1)$$

```
unassume(x):
```

Example 3

The following list provides a collection of sets:

```
L := [{a, b}, {1, 2, a, c}, {3, a, b}, {a, c}]:
```

The functional equivalent `_intersect` of the `intersect` operator accepts an arbitray number of arguments. Thus, the intersection of all sets in `L` can be computed as follows:

```
_intersect(op(L))
```

$$\{a\}$$

The union of all sets in `L` is:

```
_union(op(L))
```

$$\{1, 2, 3, a, b, c\}$$

```
delete L:
```

Example 4

`universe` represents the set of all mathematical objects:

```
_intersect()
```

$$\text{universe}$$

Parameters

$\mathtt{set}_1$, $\mathtt{set}_2$, …

Finite sets of type `DOM_SET`, or intervals of type `Dom::Interval`, or arithmetical expressions

Return Values

Set, an interval, a symbolic expression of type `"_intersect"`, `"_minus"`, `"_union"`, or `universe`.

Overloaded By

set1, set2

See Also

MuPAD Functions
`intersect` | `minus` | `subset` | `universe`

interval

Convert constant subexpressions to intervals

Syntax

```
interval(object)
```

Description

`interval(object)` converts all constant subexpressions of `object` to floating point intervals.

`interval` is the analogue of `float`. While the latter converts exact numbers and numerical expressions to floating-point approximations, `interval` converts numbers and numerical expressions to enclosing floating-point intervals.

If `object` is an arithmetical expression, `interval(object)` recursively descends into the subexpressions of `object` and replaces all integers,rationals, and floating point numbers as well as the constants CATALAN, EULER and PI by floating-point intervals enclosing them. Afterwards, the resulting expression is evaluated via interval arithmetic.

If `object` is not an arithmetical expression, `interval` returns the object unchanged.

Examples

Example 1

Only constant expressions such as numbers 1, $\frac{2}{3}$, $0.123 + 4.5\,i$ etc. and numerical expressions `PI + sqrt(2)`, `sin(PI/24)` etc. are converted to floating-point intervals. Symbolic objects such as identifiers, indexed identifiers etc. are left untouched:

```
interval(4*x[1] + PI*x[2]^2/sin(1) + 1/4)
```

$$3.733453333 \ldots 3.733453334\, x_2^{\,2} + x_1\; 4.0 \ldots 4.0 + 0.25 \ldots 0.25$$

```
interval(f(g(2 + x) + sin(1)*sqrt(PI)))
```

$$f(g(x + 2.0 \ldots 2.0) + 1.491468487 \ldots 1.491468488)$$

Example 2

The special MuPAD constants CATALAN, EULER and PI can be converted to an enclosing floating-point interval:

```
interval(CATALAN), interval(EULER), interval(PI)
```

$$0.9159655941 \ldots 0.9159655942,\ 0.5772156649 \ldots 0.577215665,\ 3.141592653 \ldots 3.141592654$$

Parameters

`object`

An arbitrary MuPAD object

Return Values

MuPAD object

See Also

MuPAD Domains
`Dom::FloatIV`

MuPAD Functions
`float` | `hull` | `misc::maprec`

inverse

Inverse of a matrix

Syntax

```
inverse(A, <Normal>)
```

Description

`inverse(A)` returns the inverse of the matrix *A*.

If the input is a matrix *A* of category `Cat::Matrix`, then `A^(-1)` is called to compute the result. In contrast to the overloaded arithmetics, the function `inverse` also operates on `array`s and `hfarray`s.

If the input matrix is an `array` of domain type `DOM_ARRAY`, then `numeric::inverse(A, Symbolic)` is called to compute the result.

The inverse of `hfarray`s of domain type `DOM_HFARRAY` is internally computed via `numeric::inverse(A)`.

If the argument does not evaluate to a matrix of one of the types mentioned above, a symbolic call `inverse(A)` is returned.

By default, `inverse` calls `normal` before returning results. This additional internal call ensures that the final result is normalized. This call can be computationally expensive. It also affects the result returned by `inverse` only if a matrix contains variables or exact expressions, such as `sqrt(5)` or `sin(PI/7)`.

To avoid this additional call, specify `Normal = FALSE`. In this case, `inverse` also can return normalized results, but does not guarantee such normalization. See “Example 4” on page 1-1083.

Examples

Example 1

Compute the inverse of a matrix given by various data types:

```
A := array(1..2, 1..2, [[1, 2], [3, PI]]);
inverse(A)
```

$$\begin{pmatrix} 1 & 2 \\ 3 & \pi \end{pmatrix}$$

$$\begin{pmatrix} \frac{\pi}{\pi-6} & -\frac{2}{\pi-6} \\ -\frac{3}{\pi-6} & \frac{1}{\pi-6} \end{pmatrix}$$

```
B := hfarray(1..2, 1..2, [[1, 2], [3, PI]]);
inverse(B)
```

$$\begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 3.141592654 \end{pmatrix}$$

$$\begin{pmatrix} -1.099071012 & 0.6996903372 \\ 1.049535506 & -0.3498451686 \end{pmatrix}$$

```
C := matrix(2, 2, [[1, 2], [3, PI]]);
inverse(C)
```

$$\begin{pmatrix} 1 & 2 \\ 3 & \pi \end{pmatrix}$$

$$\begin{pmatrix} \frac{\pi}{\pi-6} & -\frac{2}{\pi-6} \\ -\frac{3}{\pi-6} & \frac{1}{\pi-6} \end{pmatrix}$$

```
delete A, B, C:
```

Example 2

The following matrix is not invertible:

```
inverse(matrix([[1, 2], [3, 6]]))
```

FAIL

Example 3

If the input does not evaluate to a matrix, then symbolic calls are returned:

```
delete A, B:
inverse(A + 2*B)
```

$$(A+2\,B)^{-1}$$

Example 4

Using `Normal` can significantly decrease the performance of `inverse`. For example, computing the inverse of this matrix takes a long time:

```
n := 5:
inv5 := inverse(matrix(n, n, [[1/(x[i] + x[j]) $
                                    j = 1..n] $
                                    i = 1..n])):
```

For better performance, specify `Normal = FALSE`:

```
n := 5:
inv5 := inverse(matrix(n, n, [[1/(x[i] + x[j]) $
                                    j = 1..n] $
                                    i = 1..n]),
                               Normal = FALSE):
```

Parameters

A

Square matrix: either a two-dimensional `array`, a two-dimensional `hfarray`, or an object of the category `Cat::Matrix`

Options

`Normal`

Option, specified as `Normal = b`

Return normalized results. The value `b` must be `TRUE` or `FALSE`. By default, `Normal = TRUE`, meaning that `inverse` guarantees normalization of the returned results. Normalizing results can be computationally expensive.

Return Values

The inverse is returned as a matrix of the same type as the input matrix. If the matrix is not invertible, then `FAIL` is returned. If the input does not evaluate to a matrix, then a symbolic call of `inverse` is returned.

Overloaded By

A

See Also

MuPAD Functions
`numeric::inverse`

_invert

Reciprocal of an expression

Syntax

```
1/ x

_invert(x)
```

Description

`_invert(x)` computes the reciprocal `1/x` of `x`.

`1/x` is equivalent to the function call `_invert(x)`. It represents the inverse of the element `x` with respect to multiplication, i.e., `x * (1/x) = 1`.

The reciprocal of a number of type `Type::Numeric` is returned as a number.

`1/x` is overloaded for matrix domains (`matrix`) and returns the inverse of the matrix `x`.

If `x` is not an element of a library domain with an `"_invert"` method, `1/x` is internally represented as `x^(-1) = _power(x, -1)`.

If `x` is an element of a domain with a slot`"_invert"`, then this method is used to compute `1/x`. Many library domains overload the `/` operator by an appropriate `"_invert"` slot. Note that `a/x` calls the overloading slot `x::dom::_invert(x)` only for `a = 1`.

If neither `x` nor `y` overload the binary operator `/` by a `"_divide"` method, the quotient `x/y` is equivalent to `x * y^(-1) = _mult(x, _power(y, -1))`.

For finite sets, `1/X` is the set $\left\{\frac{1}{x} \middle| x \in X\right\}$.

Examples

Example 1

The reciprocal of an expression is the inverse with respect to `*`:

```
_invert(x), x * (1/x) = x * _invert(x)
```

$$\frac{1}{x}, 1 = 1$$

```
3 * y * x^2 / 27 / x
```

$$\frac{x\,y}{9}$$

Internally, a symbolic expression `1/x` is represented as `x^(-1) = _power(x, -1)`:

```
type(1/x), op(1/x, 0), op(1/x, 1), op(1/x, 2)
```

"_power", _power, x, -1

Example 2

For finite sets, `1/X` is the set $\left\{\frac{1}{x} \middle| x \in X\right\}$:

```
1/{a, b, c}
```

$$\left\{\frac{1}{a}, \frac{1}{b}, \frac{1}{c}\right\}$$

Example 3

Various library domains such as matrix domains or residue class domains overload `_invert`:

```
x := Dom::Matrix(Dom::IntegerMod(7))([[2, 3], [3, 4]]):
```

```
x, 1/x, x * (1/x)
```

$$\begin{pmatrix} 2 \bmod 7 & 3 \bmod 7 \\ 3 \bmod 7 & 4 \bmod 7 \end{pmatrix}, \begin{pmatrix} 3 \bmod 7 & 3 \bmod 7 \\ 3 \bmod 7 & 5 \bmod 7 \end{pmatrix}, \begin{pmatrix} 1 \bmod 7 & 0 \bmod 7 \\ 0 \bmod 7 & 1 \bmod 7 \end{pmatrix}$$

```
delete x:
```

Parameters

x

An arithmetical expression or a set

Return Values

Arithmetical expression or a set.

Overloaded By

x

See Also

MuPAD Functions

`*` | `+` | `-` | `/` | `^` | `_divide` | `_subtract`

irreducible

Test irreducibility of a polynomial

Syntax

```
irreducible(p)
```

Description

`irreducible(p)` tests if the polynomial `p` is irreducible.

A polynomial $p \in k[x_1, \ldots, x_n]$ is irreducible over the field k if p is nonconstant and is not a product of two nonconstant polynomials in $k[x_1, \ldots, x_n]$.

`irreducible` returns `TRUE` if the polynomial is irreducible over the field implied by its coefficients. Otherwise, `FALSE` is returned. See the function `factor` for details on the coefficient field that is assumed implicitly.

The polynomial may be either a (multivariate) polynomial over the rationals, a (multivariate) polynomial over a field (such as the residue class ring `IntMod(n)` with a prime number `n`) or a univariate polynomial over an algebraic extension (see `Dom::AlgebraicExtension`).

Internally, a polynomial expression is converted to a polynomial of type `DOM_POLY` before irreducibility is tested.

Examples

Example 1

With the following call, we test if the polynomial expression x^2 - 2 is irreducible. Implicitly, the coefficient field is assumed to consist of the rational numbers:

```
irreducible(x^2 - 2)
```

```
irreducible(x^2 - 2)
```

TRUE

```
factor(x^2 - 2)
```

$x^2 - 2$

Since x^2 - 2 factors over a field extension of the rationals containing the radical $\sqrt{2}$, the following irreducibility test is negative:

```
irreducible(sqrt(2)*(x^2 - 2))
```

FALSE

```
factor(sqrt(2)*(x^2 - 2))
```

$\sqrt{2}\,(x-\sqrt{2})\,(x+\sqrt{2})$

The following calls use polynomials of type `DOM_POLY`. The coefficient field is given explicitly by the polynomials:

```
irreducible(poly(6*x^3 + 4*x^2 + 2*x - 4, IntMod(13)))
```

TRUE

```
factor(poly(6*x^3 + 4*x^2 + 2*x - 4, IntMod(13)))
```

$6\,\mathrm{poly}(x^3+5\,x^2-4\,x-5,\,[x],\,\mathrm{IntMod}(13))$

```
irreducible(poly(3*x^2 + 5*x + 2, IntMod(13)))
```

FALSE

```
factor(poly(3*x^2 + 5*x + 2, IntMod(13)))
```

$3\,\mathrm{poly}(x+5,\,[x],\,\mathrm{IntMod}(13))\,\mathrm{poly}(x+1,\,[x],\,\mathrm{IntMod}(13))$

Parameters

p

A polynomial of type DOM_POLY or a polynomial expression

Return Values

TRUE or FALSE.

Overloaded By

p

See Also

MuPAD Functions

content | factor | gcd | icontent | ifactor | igcd | ilcm | isprime | lcm | poly | polylib::divisors | polylib::primpart | polylib::sqrfree

is

Check a mathematical property of an expression

Syntax

```
is(cond)
```

```
is(ex, set)
```

Description

`is(cond)` checks whether the condition `cond` holds for all possible values.

`is(ex, set)` checks whether the expression `ex` lies in the set `set`.

The property mechanism helps to simplify expressions involving expressions that carry “mathematical properties”. The function `assume` allows to assume “assumptions” such as `` `x is a real number' `` or `` `x is an odd integer' `` to an identifier `x`, say. Arithmetical expressions involving `x` may inherit such properties. E.g., `` `1 + x^2 is positive' `` if `` `x is a real number' ``. The function `is` is the basic tool for querying mathematical properties.

`is` queries the assumptions of all involved identifiers and checks whether the condition `cond` holds for all possible values. If this is the case, then `is` returns `TRUE`. If `is` derives that cond is not satisfied by any possible value it returns `FALSE`. Otherwise, `is` returns `UNKNOWN`.

If a relation is given to `is`, and the operands are complex numbers or identifiers with this property, `is` returns `FALSE`, because a relations holds only with real objects. Cf. “Example 4” on page 1-1094.

It may happen that `is` returns `UNKNOWN`, although the queried property holds mathematically. Cf. “Example 5” on page 1-1094.

In MuPAD, there also exists the function `bool` to check a relation `y rel z`. However, there are two main differences between `bool` and `is`:

1. `bool` produces an error if it cannot decide whether the relation holds or not; `is(y rel z)` returns `UNKNOWN` in this case.

2 `bool` does not take properties into account.

Cf. “Example 3” on page 1-1093.

If `bool(y rel z)` returns `TRUE`, then so does `is(y rel z)`. However, `is` is more powerful than `bool`, even when no properties are involved. Cf. “Example 3” on page 1-1093. On the other hand, `is` is usually much slower than `bool`.

Note: Be careful when using `is` in a condition of an `if` statement or a `for`, `while`, or `repeat` loop: these constructs cannot handle the value `UNKNOWN`. Use either `is(...) = TRUE` or a `case` statement. Cf. “Example 6” on page 1-1095.

If `is` needs to check whether a constant symbolic expression is zero, then it may employ a heuristic numerical zero test based on floating-point evaluation. Despite internal numerical stabilization, this zero test may return the wrong answer in exceptional pathological cases; in such a case, `is` may return a wrong result as well.

Examples

Example 1

The identifier `x` is assumed to be an integer:

```
assume(x, Type::Integer):
is(x, Type::Integer), is(x > 0), is(x^2 >= 0)
```

```
TRUE, UNKNOWN, TRUE
```

The identifier `x` is assumed to be a positive real number:

```
assume(x > 0):
is(x > 1), is(x >= 0), is(x < 0)
```

```
UNKNOWN, TRUE, FALSE
```

```
unassume(x):
```

Example 2

`is` can derive certain facts even when no properties were assumed explicitly:

```
is(x > x + 1), is(abs(x) >= 0)
```

```
FALSE, TRUE
```

```
is(Re(exp(x)), Type::Real)
```

```
TRUE
```

Example 3

For relations between numbers, `is` yields the same answers as `bool`:

```
bool(1 > 0), is(1 > 0)
```

```
TRUE, TRUE
```

`is` resolves more constant symbolic expressions than `bool`:

```
is(sqrt(14) <= sqrt(2)*sqrt(7)),
is(sin(10^20) > 0),
is(sqrt(2) > 1.41)
```

```
TRUE, FALSE, TRUE
```

```
bool(sqrt(14) <= sqrt(2)*sqrt(7))
```

```
Error: Cannot evaluate to Boolean. [_leequal]
```

```
bool(sin(10^20) > 0)
```

```
Error: Cannot evaluate to Boolean. [_less]
```

```
is(exp(5), Type::Real), is(PI, Type::PosInt)
```

```
TRUE, FALSE
```

Example 4

In the next example a relation with complex objects is given, the returned value is FALSE:

```
is(0 < I), is(I + 1 > I), is(1 + 2*I <= 2 + 3*I)
```

```
FALSE, FALSE, FALSE
```

The identifier in the next example is assumed to be complex, but it could be real too:

```
assume(x, Type::Complex):
is(x > 0)
```

```
UNKNOWN
```

The next relation is false, either the identifier `x` is real, then the relation is false, or the identifiers is not real, then the comparison is illegal:

```
unassume(x):
is(x + 1 < x)
```

```
FALSE
```

```
unassume(x):
```

Example 5

Here are some examples where the queried property can be derived mathematically. However, the current implementation of `is` is not yet strong enough to derive the property:

```
assume(x in Z_ and y in Z_ and x^2 + y^2 = 2);
is(x > 1)
```

```
UNKNOWN
```

```
unassume(x):
```

Example 6

Care must be taken when using `is` in `if` statements or `for`, `repeat`, `while` loops:

```
myabs := proc(x)
         begin
           if is(x >= 0) then
             x
           elif is(x < 0) then
             -x
           else
             procname(x)
           end_if
         end_proc:

assume(x < 0): myabs(1), myabs(-2), myabs(x)
```

$$1, 2, -x$$

When the call of `is` returns `UNKNOWN`, an error occurs because `if` expects `TRUE` or `FALSE`:

```
unassume(x): myabs(x)


Error: Cannot evaluate to Boolean. [if]
  Evaluating: myabs
```

The easiest way to achieve the desired functionality is a comparison of the result of `is` with `TRUE`:

```
myabs := proc(x)
         begin
           if is(x >= 0) = TRUE then
             x
           elif is(x < 0) = TRUE then
             -x
           else
             procname(x)
           end_if
         end_proc:

myabs(x)
```

$$\text{myabs}(x)$$

```
delete myabs:
```

Example 7

`is` can handle sets returned by `solve`. These include intervals of type `Dom::Interval` and `R_` = `solvelib::BasicSet(Dom::Real)`:

```
assume(x >= 0 and x <= 1):
is(x in Dom::Interval([0, 1])), is(x in R_)
```

$$\text{TRUE, TRUE}$$

The following `solve` command returns the solution as an infinite parameterized set of type `Dom::ImageSet`:

```
unassume(x): solutionset := solve(sin(x) = 0, x)
```

$$\{\pi\, k \mid k \in \mathbb{Z}\}$$

```
domtype(solutionset)
```

$$\text{DomImageSet}$$

`is` can be used to check whether an expression is contained in this set:

```
is(20*PI in solutionset), is(PI/2 in solutionset)
```

$$\text{TRUE, FALSE}$$

```
delete solutionset:
```

Parameters

`cond`

A condition

ex

arithmetical expression

set

A property representing a set of numbers (e.g., `Type::PosInt`) or a set returned by `solve`; such a set can be an element of `Dom::Interval`, `Dom::ImageSet`, `piecewise`, or one of `C_`, `R_`, `Q_`, `Z_`.

Return Values

`TRUE`, `FALSE`, or `UNKNOWN`.

See Also

MuPAD Functions
`assume` | `bool` | `getprop` | `unassume`

isolate

Isolate variable or expression from equation

Syntax

```
isolate(eq, expr)
```

Description

`isolate(eq, expr)` rearranges the equation eq so that the expression expr appears on the left side. The result is similar to solving eq for expr. However, `isolate` returns only one solution even if multiple solutions exist. If `isolate` cannot isolate expr from eq, it moves all terms containing expr to the left side. You can use the output of `isolate` as input to `subs` to eliminate expr from eq.

If eq has no solution, `isolate` returns an error. The `isolate` function also ignores special cases. If the only solutions to eq are special cases, then `isolate` ignores those special cases and returns an error. Additionally, if the solution returned contains parameters, the parameters might not be valid for special cases.

You cannot specify expr as a mathematical constant such as PI, EULER, and so on.

By default, `isolate(eq, expr)` returns only solutions consistent with the properties of `expr`.

If the input contains floating-point numbers, the solver replaces them by approximate rational values. The accuracy of these approximate values depends on the environment variable `DIGITS`. If `isolate` finds a solution, MuPAD internally calls the `float` function for that solution, and then returns the result.

Environment Interactions

`isolate` reacts to properties of identifiers.

Examples

Example 1

Isolate `x` from the equation `a*x^2 + b*x + c = 0`.

```
eqn := a*x^2 + b*x + c = 0:
xSol := isolate(eqn, x)
```

$$x = -\frac{b + \sqrt{b^2 - 4\,a\,c}}{2\,a}$$

Even though the equation has multiple solutions, `isolate` returns only one solution.

Eliminate `x` from `eqn` by calling `subs` to substitute for `x` using `xSol`.

```
subs(eqn, xSol)
```

$$c + \frac{\left(b + \sqrt{b^2 - 4\,a\,c}\right)^2}{4\,a} - \frac{b\left(b + \sqrt{b^2 - 4\,a\,c}\right)}{2\,a} = 0$$

You can also isolate expressions. Isolate `x(t)` from the following equation.

```
isolate(a*x(t)^2 + b*c = 0, x(t))
```

$$x(t) = \frac{\sqrt{-b}\,\sqrt{c}}{\sqrt{a}}$$

Isolate `a*x(t)` from the same equation.

```
isolate(a*x(t)^2 + b*c = 0, a*x(t))
```

$$a\,x(t) = -\frac{b\,c}{x(t)}$$

If `isolate` cannot find a symbolic solution, it returns an error. Because `isolate` does not return special cases, it also returns an error if the only solutions are special cases.

Compare `isolate` with `solve` for an equation whose only solution is a special case.

```
solve(x = x+a, x);
isolate(x = x+a, x)
```

$$\begin{cases} \mathbb{C} & \text{if } a = 0 \\ \varnothing & \text{if } a \neq 0 \end{cases}$$

```
Error: The equation has no solution. [isolate]
```

`solve` returns the special case while `isolate` ignores the special case and returns an error.

Example 2

For equations with multiple solutions, `isolate` returns the 'simplest' solution.

Isolate `x` from equations with many solutions to demonstrate this behavior of `isolate`.

```
isolate(cos(x) = x, x)
```

$$\cos(x) - x = 0$$

```
isolate(x^2 = 1, x)
```

$$x = 1$$

```
isolate(sin(x) = 0, x)
```

$$x = 0$$

```
isolate(sqrt(x) = C, x)
```

$$x = C^2$$

Example 3

`isolate` only returns results compatible with assumptions on the variable to be isolated. For example, assume that `x` represents a real negative number. Then, isolate it from the following equation.

```
assume(x, Type::Negative):
isolate(x^4 = 1, x)
```

$$x = -1$$

Remove the assumption. `isolate` chooses a different solution to return.

```
unassume(x):
isolate(x^4 = 1, x)
```

$$x = 1$$

Example 4

If the input contains floating-point numbers, MuPAD calls the `float` function for the obtained solution.

Isolate `x` from an equation with floating-point numbers.

```
isolate(x^3 + 3.0*x + 1 = 0, x)
```

$$x = -0.3221853546$$

Example 5

You can isolate an expression in an equation with symbolic parameters. The `isolate` function returns a general solution where the parameter values are not guaranteed to hold for special cases.

Isolate `x` in the equation.

```
isolate(a*x^2/(x-a) = 1, x)
```

$$x = \frac{\sqrt{-(2a-1)(2a+1)+1}}{2a}$$

The returned value of `x` does not hold in the special case that parameter `a` has value `0`.

Parameters

`eq`

An equation.

`expr`

The variable or expression to be isolated.

Return Values

`isolate(eq, expr)` returns an equation where the right side does not contain the variable or expression to be isolated. `isolate` does not introduce newly generated parameters. The returned equation is always a valid input to `subs`..

See Also

MuPAD Functions

`float` | `lhs` | `linsolve` | `numeric::linsolve` | `numeric::solve` | `rhs` | `RootOf` | `solve`

Introduced in R2015a

isprime

Primality test

Syntax

```
isprime(n)
```

Description

`isprime(n)` checks whether `n` is a prime number.

`isprime` is a fast probabilistic prime number test (Miller-Rabin test). The function returns `TRUE` when the positive integer`n` is either a prime number or a strong pseudo-prime for 10 independently and randomly chosen bases. Otherwise, `isprime` returns `FALSE`.

If `n` is positive and `isprime` returns `FALSE`, then `n` is guaranteed to be composite. If `n` is positive and `isprime` returns `TRUE`, then `n` is prime with a very high probability.

Use `numlib::proveprime` for a prime number test that always returns the correct answer. Note, however, that it is usually much slower than `isprime`.

`isprime()` and `isprime(1)` return `FALSE`. `isprime` returns always `FALSE` if `n` is a negative integer.

`isprime` returns an error message if its argument is a number but not an integer. `isprime` returns a symbolic `isprime` call if the argument is not a number.

Examples

Example 1

The number `989999` is prime:

```
isprime(989999)
```

```
TRUE
```

```
ifactor(989999)
```

```
989999
```

In contrast to `ifactor`, `isprime` can handle large numbers:

```
isprime(2^(2^11) + 1)
```

```
FALSE
```

`isprime()` and `isprime(1)` return FALSE:

```
isprime(0), isprime(1)
```

```
FALSE, FALSE
```

Negative numbers yield FALSE as well:

```
isprime(-13)
```

```
FALSE
```

For non-numeric arguments, a symbolic `isprime` call is returned:

```
delete n: isprime(n)
```

```
isprime(n)
```

Parameters

n

An arithmetical expression representing an integer

Return Values

Either `TRUE` or `FALSE`, or a symbolic `isprime` call.

References

Reference: Michael O. Rabin, Probabilistic algorithms, in J. F. Traub, ed., *Algorithms and Complexity*, Academic Press, New York, 1976, pp. 21–39.

See Also

MuPAD Functions

`factor` | `ifactor` | `igcd` | `ilcm` | `irreducible` | `ithprime` | `nextprime` | `numlib::primedivisors` | `numlib::proveprime` | `prevprime`

isqrt

Integer square root

Syntax

```
isqrt(n)
```

Description

`isqrt(n)` computes an integer approximation to the square root of the integer `n`.

If `n` is a perfect square, then `isqrt` returns the unique nonnegative integer whose square is `n`. More generally, if `n` is a nonnegative integer, then `isqrt` computes `trunc(sqrt(n))`. Thus the approximation error is less than `1`.

If `n` is a negative integer, then `isqrt` computes `trunc(sqrt(-n)) *I`.

`isqrt` returns an error message if its argument is a number but not an integer. `isqrt` returns a symbolic `isqrt` call if the argument is not a number.

Examples

Example 1

We compute some integer square roots:

```
isqrt(4), isqrt(5)
```

```
2, 2
```

The approximation error is less than `1`:

```
isqrt(99), float(sqrt(99))
```

```
9, 9.949874371
```

The integer square root of a negative integer is an integral multiple of `I`:

```
isqrt(-4), isqrt(-5)
```

$$2\,i,\ 2\,i$$

If the argument is not a number, the result is a symbolic `isqrt` call:

```
delete n: isqrt(n)
```

$$\mathrm{isqrt}(n)$$

```
type(%)
```

"isqrt"

Parameters

n

An arithmetical expression representing an integer

Return Values

Nonnegative integer, an integral multiple of `I`, or a symbolic `isqrt` call.

Overloaded By

n

See Also

MuPAD Functions

`_power` | `icontent` | `ifactor` | `igcd` | `ilcm` | `numlib::ispower` | `numlib::issqr` | `sqrt` | `trunc`

iszero

Generic zero test

Syntax

```
iszero(object)
```

Description

`iszero(object)` checks whether `object` is the zero element in the domain of `object`.

Use the condition `iszero(object)` instead of `object = 0` to decide whether `object` is the zero element, because `iszero(object)` is more general than `object = 0`. If the call `bool(object = 0)` returns `TRUE`, then `iszero(object)` returns `TRUE` as well, but in general not vice versa (see "Example 1" on page 1-1109).

If `object` is an element of a basic type, then `iszero` returns `TRUE` precisely if one of the following is true: `object` is the integer `0` (of domain type `DOM_INT`), the floating-point value `0.0` (of domain type `DOM_FLOAT`), the floating-point interval (of domain type `DOM_INTERVAL`) `0...0`, or the zero polynomial (of domain type `DOM_POLY`). In the case of a polynomial, the result `FALSE` is guaranteed to be correct only if the coefficients of the polynomial are in normal form (i.e., if zero has a unique representation in the coefficient ring). See also `Ax::normalRep`.

If `object` is an element of a library domain, then the method `"iszero"` of the domain is called and the result is returned. If this method does not exist, then the function `iszero` returns `FALSE`.

`iszero` performs a purely syntactical zero test. If `iszero` returns `TRUE`, then the answer is always correct. If `iszero` returns `FALSE`, however, then it may still be true that mathematically `object` represents zero (see "Example 3" on page 1-1110). In such cases, the MuPAD functions `normal` or `simplify` may be able to recognize this.

Note: `iszero` does *not* take into account properties of identifiers in `object` that have been set via `assume`. In particular, you should not use `iszero` in an argument passed to `assume` or `is`; use the form `object = 0` instead (see "Example 2" on page 1-1110).

Note: Do not use `iszero` in a condition passed to `piecewise`. In contrast to `object = 0`, the command `iszero(object)` is evaluated immediately, before it is passed to `piecewise`, while the evaluation of `object = 0` is handled by `piecewise` itself. Thus using `iszero` in a `piecewise` command usually leads to unwanted effects (see "Example 4" on page 1-1110).

Examples

Example 1

`iszero` handles the basic data types:

```
iszero(0), iszero(1/2), iszero(0.0), iszero(I), iszero(-1...1)
```

```
TRUE, FALSE, TRUE, FALSE, FALSE
```

`iszero` works for polynomials:

```
p:= poly(x^2 + y, [x]):
iszero(p)
```

```
FALSE
```

```
iszero(poly(0, [x, y]))
```

```
TRUE
```

`iszero` is more general than =:

```
bool(0 = 0), bool(0.0 = 0), bool(poly(0, [x]) = 0)
```

```
TRUE, FALSE, FALSE
```

```
iszero(0), iszero(0.0), iszero(poly(0, [x]))
```

```
TRUE, TRUE, TRUE
```

Example 2

`iszero` does not react to properties:

```
assume(a = b): is(a - b = 0)
```

```
    TRUE
```

```
iszero(a - b)
```

```
    FALSE
```

Example 3

Although `iszero` returns `FALSE` in the following example, the expression in question mathematically represents zero:

```
iszero(sin(x)^2 + cos(x)^2 - 1)
```

```
    FALSE
```

In this case `simplify` is able to decide this:

```
simplify(sin(x)^2 + cos(x)^2 - 1)
```

```
    0
```

Example 4

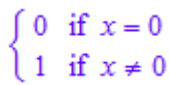
`iszero` should not be used in a condition passed to `piecewise`:

```
delete x:
piecewise([iszero(x), 0], [x <> 0, 1])
```

$$\begin{cases} 1 & \text{if } x \neq 0 \end{cases}$$

The first branch was discarded because `iszero(x)` immediately evaluates to `FALSE`. Instead, use the condition `x = 0`, which is passed unevaluated to `piecewise`:

```
piecewise([x = 0, 0], [x <> 0, 1])
```



Parameters

`object`

An arbitrary MuPAD object

Return Values

Either TRUE or FALSE

Overloaded By

object

See Also

MuPAD Axioms
`Ax::normalRep`

MuPAD Functions
`_equal` | `bool` | `is` | `normal` | `sign` | `simplify`

ithprime

I-th prime number

Syntax

```
ithprime(i)
```

```
ithprime(<PrimeLimit>)
```

Description

`ithprime(i)` returns the `i`-th prime number.

If the argument `i` is a positive integer, then `ithprime` returns the `i`-th prime number. An unevaluated call is returned, if the argument is not of type `Type::Numeric`. An error occurs if the argument is a number that is not a positive integer.

The first prime number `ithprime(1)` is 2.

If the `i`-th prime number is contained in the system's internal prime number table (see the help page for `ifactor`), then it is returned by a fast kernel function. Otherwise, MuPAD iteratively calls `nextprime`, using some suitable pre-computed value of `ithprime` as starting point. This is still reasonably fast for $i \leq 1000000$. If `i` exceeds this value, however, then the run time grows exponentially with the number of digits of `i`.

Examples

Example 1

The first 10 prime numbers:

```
ithprime(i) $ i = 1..10
```

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29$$

A larger prime:

```
ithprime(123456)
```

```
1632899
```

Symbolic arguments lead to an unevaluated call:

```
ithprime(i)
```

```
ithprime(i)
```

Parameters

`i`

An arithmetical expression

Options

`PrimeLimit`

Return the number of primes in the internal prime table

`ithprime(PrimeLimit)` returns an integer, namely the number of primes in the internal prime number table. The table contains all primes below some bound which can be obtained by calling `ifactor(PrimeLimit)`. On UNIX platforms, the size of this table can be changed via the MuPAD command line flag `-L`.

Return Values

Prime number or an unevaluated call to `ithprime`

See Also

MuPAD Functions

`ifactor` | `igcd` | `ilcm` | `isprime` | `nextprime` | `numlib::pi` | `prevprime`

iztrans

Inverse Z transform

Syntax

```
iztrans(F, z, k)
```

Description

`iztrans(F, z, k)` computes the inverse Z transform of the expression `F = F(z)` with respect to the variable `z` at the point `k`.

If `R` is a positive number, such that the function `F(Z)` is analytic on and outside the circle `|z| = R`, then the inverse Z-transform is defined as follows:

$$f(k) = \frac{1}{2\pi i} \oint_{|z|=R} F(z) z^{k-1} dz, \quad k = 0,1,2...$$

If `iztrans` cannot find an explicit representation of the transform, it returns an unevaluated function call. See “Example 3” on page 1-1115.

If `F` is a matrix, `iztrans` applies the inverse Z transform to all components of the matrix.

To compute the direct Z transform, use `ztrans`.

Examples

Example 1

Compute the inverse Z transform of these expressions:

```
iztrans(exp(1/z), z, k)
```

$$\frac{1}{k!}$$

```
iztrans((z*sin(1))/(z^2 - 2*cos(1)*z + 1), z, k)
```

$$\sin(k)$$

Example 2

Compute the inverse Z transform of this expression with respect to the variable `z`:

```
f := iztrans((3*z)/(z - 1) + (2*z)/(z - 1)^2, z, k)
```

$$2k+3$$

Evaluate the inverse Z transform of the expression at the points $k = 2a + 3$ and $k = 1 + i$. You can evaluate the resulting expression `f` using `|` (or its functional form `evalAt`):

```
f | k = 2*a + 3
```

$$4a+9$$

Also, you can evaluate the inverse Z transform at a particular point directly:

```
iztrans((3*z)/(z - 1) + (2*z)/(z - 1)^2, z, 1 + I)
```

$$5+2\,i$$

Example 3

If `iztrans` cannot find an explicit representation of the transform, it returns an unevaluated call:

```
iztrans(F(z), z, k)
```

$$\text{iztrans}(F(z), z, k)$$

`ztrans` returns the original expression:

```
ztrans(%, k, z)
```

$$F(z)$$

Example 4

Compute the inverse Z transforms of these expressions. The results involve the `kroneckerDelta` function:

```
iztrans(1/z, z, k)
```

$$\delta_{k-1,0}$$

```
iztrans((z^3 + 3*z^2 + 6*z + 5)/z^5, z, k)
```

$$\delta_{k-2,0} + 3\,\delta_{k-3,0} + 6\,\delta_{k-4,0} + 5\,\delta_{k-5,0}$$

Example 5

Compute the inverse Z tranform of this expression:

```
iztrans(z*diff(g(z), z), z, k)
```

$$-k\,\mathrm{iztrans}(g(z),\, z,\, k)$$

Parameters

F

Arithmetical expression or matrix of such expressions

z

Identifier or indexed identifier

k

Arithmetical expression representing the evaluation point

Return Values

Arithmetical expression or unevaluated function call of type `iztrans`. An explicit result can be a `piecewise` object. If the first argument is a matrix, then the result is returned as a matrix.

Overloaded By

F

See Also

MuPAD Functions

`iztrans::addpattern` | `ztrans` | `ztrans::addpattern`

More About

- "Z-Transforms"

iztrans::addpattern

Add patterns for the inverse Z transform

Syntax

```
iztrans::addpattern(pat, z, k, res, <vars, <conds>>)
```

Description

`iztrans::addpattern(pat, z, k, res)` teaches `iztrans` to return *iztrans*(*pat*, *z*, *k*) = *res*.

The `iztrans` function uses a set of patterns for computing inverse Z transforms. You can extend the set by adding your own patterns. To add a new pattern to the pattern matcher, use `iztrans::addpattern`. MuPAD does not save custom patterns permanently. The new patterns are available in the *current* MuPAD session only.

Variable names that you use when calling `iztrans::addpattern` can differ from the names that you use when calling `iztrans`. See “Example 2” on page 1-1119.

You can include a list of free parameters and a list of conditions on these parameters. These conditions and the result are protected from premature evaluation. This means that you can use `not iszero(a^2 - b)` instead of `hold( _not @ iszero )(a^2 - b)`.

The following conditions treat assumptions on identifiers differently:

- `a^2 - b <> 0` takes into account assumptions on identifiers.
- `not iszero(a^2 - b)` disregards assumptions on identifiers.

See “Example 3” on page 1-1119.

Environment Interactions

Calling `iztrans::addpattern` changes the expressions returned by future calls to `iztrans`.

Examples

Example 1

Compute the inverse Z transform of the function `bar`. By default, MuPAD does not have a pattern for this function:

```
iztrans(bar(z), z, k)
```

$$\text{iztrans}(\text{bar}(z), z, k)$$

Add a pattern for the inverse Z transform of `bar` using `iztrans::addpattern`:

```
iztrans::addpattern(bar(z), z, k, foo(k)):
```

Now `iztrans` returns the inverse Z transform of `bar`:

```
iztrans(bar(z), z, k)
```

$$\text{foo}(k)$$

Example 2

Define the inverse Z transform of `bar(y)` using the variables `x` and `y` as parameters:

```
iztrans::addpattern(bar(y), y, x, foo(x))
```

The `iztrans` function recognizes the added pattern even if you use other variables as parameters:

```
iztrans(bar(z), z, k)
```

$$\text{foo}(k)$$

Example 3

Use assumptions when adding this pattern for the inverse Z transform:

```
iztrans::addpattern(BAR(x*z), z, k, FOO(k/(x - 1/2))*sin(x),
                                    [x], [abs(x) < 1]):
```

```
iztrans(BAR(x*z), z, k) assuming -1 < x < 1
```

$$\mathrm{FOO}\left(\frac{k}{x-\frac{1}{2}}\right)\sin(x)$$

If $|x| \geq 1$, you cannot apply this pattern:

```
iztrans(BAR(x*z), z, k) assuming x >= 1
```

$$\mathrm{iztrans}(\mathrm{BAR}(x\ z),\ z,\ k)$$

If MuPAD cannot determine whether the conditions are satisfied, it returns a `piecewise` object:

```
iztrans(BAR(x*z), z, k)
```

$$\begin{cases} \mathrm{FOO}\left(\frac{k}{x-\frac{1}{2}}\right)\sin(x) & \text{if } |x| < 1 \end{cases}$$

Note that the resulting expression defining the inverse Z transform of `BAR(x*z)` implicitly assumes that the value of `x` is not 1/2. A strict definition of the pattern is:

```
ztrans::addpattern(BAR(x*z), z, k, FOO(k/(x - 1/2))*sin(x),
                   [x], [abs(x) < 1, x <> 1/2]):
```

If either the conditions are not satisfied or substituting the values into the result gives an error, `iztrans` ignores the pattern. For this particular pattern, you can omit specifying the assumption `x <> 1/2`. If `x = 1/2`, MuPAD throws an internal "Division by zero." error and ignores the pattern:

```
iztrans(BAR(z/2), z, k)
```

$$\mathrm{iztrans}\left(\mathrm{BAR}\left(\frac{z}{2}\right),\ z,\ k\right)$$

Parameters

`pat`

Arithmetical expression in the variable `z` representing the pattern to match

`z`

Identifier used as a variable in the pattern

`k`

Identifier used as a variable in the result

`res`

Arithmetical expression in the variable `k` representing the pattern for the result of the transformation

`vars`

List of identifiers or indexed identifiers used as "pattern variables" (placeholders in `pat` and `res`). You can use pattern variables as placeholders for almost arbitrary MuPAD expressions not containing `z` or `k`. You can restrict them by conditions given in the optional parameter `conds`.

`conds`

List of conditions on the pattern variables

Return Values

Object of type `DOM_NULL`

See Also

MuPAD Functions

`iztrans` | `ztrans` | `ztrans::addpattern`

jacobiAM

Jacobi amplitude function am

Syntax

```
jacobiAM(u,m)
```

Description

`jacobiAM(u,m)` represents the Jacobi amplitude function which is defined as the solution $\mathrm{am}(u \mid m) := \varphi$ of $\mathrm{F}(\varphi \mid m) = u$.

The Jacobi amplitude $\mathrm{am}(u \mid m)$ is defined for complex arguments u and m.

Exact results are returned for $m = 0$, $m = 1$ or $u = 0$. In all other cases an unevaluated symbolic call is returned.

A floating-point value is computed if both arguments are numerical and at least one is a floating-point number.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Most calls with exact arguments are returned evaluated:

```
jacobiAM(PI/3, 1/2)
```

$$\mathrm{am}\left(\frac{\pi}{3} \mid \frac{1}{2}\right)$$

If $m = 0$, $m = 1$ or $u = 0$, an exact result is returned:

```
jacobiAM(PI/2, 0)
```

$$\frac{\pi}{2}$$

```
jacobiAM(2, 1)
```

$$2\arctan(e^2) - \frac{\pi}{2}$$

```
jacobiAM(0, 1/2)
```

$$0$$

Parameters

u

An arithmetical expression.

m

An arithmetical expression specifying the parameter.

Return Values

Arithmetical expression.

See Also

MuPAD Functions
`ellipticF`

jacobiSN

Jacobi elliptic function sn

Syntax

```
jacobiSN(u,m)
```

Description

`jacobiSN(u,m)` represents the Jacobi elliptic function sn.

Let $u = \mathrm{F}(\varphi \mid m)$. Then the Jacobi elliptic function SN is defined as follows:

$$\mathrm{sn}(u \mid m) = \sin(\varphi)$$

The Jacobi functions are defined for complex values of u and m.

The Jacobi functions are meromorphic and doubly periodic with periods $4\,\mathrm{K}(m)$ and $4\,\mathrm{i}\,\mathrm{K}'(m)$ with respect to u.

For $m = 0$ and $m = 1$, the Jacobi functions reduce to trigonometric or constant functions.

If one argument is a floating-point number, and the other one can be converted to a floating-point number, then a floating-point number is returned.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

For most arguments, the Jacobi functions return themselves unevaluated:

```
jacobiSN(2,1/2)
```

$$\mathrm{sn}\left(2 \mid \frac{1}{2}\right)$$

Floating-point numbers are returned if at least one of the arguments is a floating-point number:

```
jacobiCN(1.5,1/2)
```

$$0.2502702593$$

Floating-point evaluation can be enforced by using `float`:

```
float(jacobiND(1,-1))
```

$$0.7404586624$$

Example 2

For $m = 0$ and $m = 1$, the result is expressed using a trigonometric function:

```
jacobiSC(u,0)
```

$$\tan(u)$$

```
jacobiND(u,1)
```

$$\cosh(u)$$

Parameters

`m`

An arithmetical expression specifying the parameter.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`ellipticCK` | `ellipticF` | `ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSD`

jacobiCN

Jacobi elliptic function cn

Syntax

```
jacobiCN(u,m)
```

Description

`jacobiCN(u,m)` represents the Jacobi elliptic function cn.

Let $u = \mathrm{F}(\varphi \mid m)$. Then the Jacobi elliptic function cn is defined as follows:

$$\mathrm{cn}(u \mid m) = \cos(\varphi)$$

The Jacobi functions are defined for complex values of u and m.

The Jacobi functions are meromorphic and doubly periodic with periods $4\,\mathrm{K}(m)$ and $4\,\mathrm{i}\,\mathrm{K}'(m)$ with respect to u.

For $m = 0$ and $m = 1$, the Jacobi functions reduce to trigonometric or constant functions.

If one argument is a floating-point number, and the other one can be converted to a floating-point number, then a floating-point number is returned.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

For most arguments, the Jacobi elliptic functions return themselves unevaluated:

```
jacobiSN(2,1/2)
```

$$\mathrm{sn}\left(2 \mid \frac{1}{2}\right)$$

Floating-point numbers are returned if at least one of the arguments is a floating-point number:

```
jacobiCN(1.5,1/2)
```

$$0.2502702593$$

Floating-point evaluation can be enforced by using `float`:

```
float(jacobiND(1,-1))
```

$$0.7404586624$$

Example 2

For $m = 0$ and $m = 1$, the result is expressed using a trigonometric function:

```
jacobiSC(u,0)
```

$$\tan(u)$$

```
jacobiND(u,1)
```

$$\cosh(u)$$

Parameters

`m`

An arithmetical expression specifying the parameter.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`ellipticCK` | `ellipticF` | `ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSD` | `jacobiSN`

jacobiDN

Jacobi elliptic function dn

Syntax

```
jacobiDN(u,m)
```

Description

`jacobiDN(u,m)` represents the Jacobi elliptic function dn.

Let $u = \mathrm{F}(\varphi \mid m)$. Then the Jacobi elliptic function dn is defined as follows:

$$\mathrm{dn}(u \mid m) = \sqrt{1 - m \sin(\varphi)^2}$$

The Jacobi functions are defined for complex values of u and m.

The Jacobi functions are meromorphic and doubly periodic with periods $4\,\mathrm{K}(m)$ and $4\,\mathrm{i}\,\mathrm{K}'(m)$ with respect to u.

For $m = 0$ and $m = 1$, the Jacobi functions reduce to trigonometric or constant functions.

If one argument is a floating-point number, and the other one can be converted to a floating-point number, then a floating-point number is returned.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

For most arguments, the Jacobi functions return themselves unevaluated:

```
jacobiSN(2,1/2)
```

$$\mathrm{sn}\left(2 \mid \frac{1}{2}\right)$$

Floating-point numbers are returned if at least one of the arguments is a floating-point number:

```
jacobiCN(1.5,1/2)
```

$$0.2502702593$$

Floating-point evaluation can be enforced by using `float`:

```
float(jacobiND(1,-1))
```

$$0.7404586624$$

Example 2

For $m = 0$ and $m = 1$, the result is expressed using a trigonometric function:

```
jacobiSC(u,0)
```

$$\tan(u)$$

```
jacobiND(u,1)
```

$$\cosh(u)$$

Parameters

m

An arithmetical expression specifying the parameter.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`ellipticCK` | `ellipticF` | `ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSD` | `jacobiSN`

jacobiCD

Jacobi elliptic function cd

Syntax

```
jacobiCD(u,m)
```

Description

`jacobiCD(u,m)` represents the Jacobi elliptic function cd.

Let $u = \mathrm{F}(\varphi \mid m)$. Then the Jacobi elliptic function cd is defined as follows:

$$\mathrm{cd}(u \mid m) = \frac{\mathrm{cn}(u \mid m)}{\mathrm{dn}(u \mid m)}$$

The Jacobi functions are defined for complex values of u and m.

The Jacobi functions are meromorphic and doubly periodic with periods $4\,\mathrm{K}(m)$ and $4\,\mathrm{i}\,\mathrm{K}'(m)$ with respect to u.

For $m = 0$ and $m = 1$, the Jacobi functions reduce to trigonometric or constant functions.

If one argument is a floating-point number, and the other one can be converted to a floating-point number, then a floating-point number is returned.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

For most arguments, the Jacobi functions return themselves unevaluated:

```
jacobiSN(2,1/2)
```

$$\mathrm{sn}\left(2 \mid \frac{1}{2}\right)$$

Floating-point numbers are returned if at least one of the arguments is a floating-point number:

```
jacobiCN(1.5,1/2)
```

$$0.2502702593$$

Floating-point evaluation can be enforced by using `float`:

```
float(jacobiND(1,-1))
```

$$0.7404586624$$

Example 2

For $m = 0$ and $m = 1$, the result is expressed using a trigonometric function:

```
jacobiSC(u,0)
```

$$\tan(u)$$

```
jacobiND(u,1)
```

$$\cosh(u)$$

Parameters

m

An arithmetical expression specifying the parameter.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`ellipticCK` | `ellipticF` | `ellipticK` | `jacobiAM` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSD` | `jacobiSN`

jacobiSD

Jacobi elliptic function sd

Syntax

```
jacobiSD(u,m)
```

Description

`jacobiSD(u,m)` represents the Jacobi elliptic function sd.

Let $u = \mathrm{F}(\varphi \mid m)$. Then the Jacobi function SD is defined as follows:

$$\mathrm{sd}(u \mid m) = \frac{\mathrm{sn}(u \mid m)}{\mathrm{dn}(u \mid m)}$$

The Jacobi functions are defined for complex values of u and m.

The Jacobi functions are meromorphic and doubly periodic with periods $4\,\mathrm{K}(m)$ and $4\,\mathrm{i}\,\mathrm{K}'(m)$ with respect to u.

For $m = 0$ and $m = 1$, the Jacobi functions reduce to trigonometric or constant functions.

If one argument is a floating-point number, and the other one can be converted to a floating-point number, then a floating-point number is returned.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

For most arguments, the Jacobi functions return themselves unevaluated:

```
jacobiSN(2,1/2)
```

$$\mathrm{sn}\left(2 \mid \frac{1}{2}\right)$$

Floating-point numbers are returned if at least one of the arguments is a floating-point number:

```
jacobiCN(1.5,1/2)
```

$$0.2502702593$$

Floating-point evaluation can be enforced by using `float`:

```
float(jacobiND(1,-1))
```

$$0.7404586624$$

Example 2

For $m = 0$ and $m = 1$, the result is expressed using a trigonometric function:

```
jacobiSC(u,0)
```

$$\tan(u)$$

```
jacobiND(u,1)
```

$$\cosh(u)$$

Parameters

m

An arithmetical expression specifying the parameter.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`ellipticCK` | `ellipticF` | `ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSN`

jacobiND

Jacobi elliptic function nd

Syntax

```
jacobiND(u,m)
```

Description

`jacobiND(u,m)` represents the Jacobi elliptic function nd.

Let $u = \mathrm{F}(\varphi \mid m)$. Then the Jacobi elliptic function nd is defined as follows:

$$\mathrm{nd}(u \mid m) = \frac{1}{\mathrm{dn}(u \mid m)}$$

The Jacobi functions are defined for complex values of u and m.

The Jacobi functions are meromorphic and doubly periodic with periods $4\,\mathrm{K}(m)$ and $4\,\mathrm{i}\,\mathrm{K}'(m)$ with respect to u.

For $m = 0$ and $m = 1$, the Jacobi functions reduce to trigonometric or constant functions.

If one argument is a floating-point number, and the other one can be converted to a floating-point number, then a floating-point number is returned.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

For most arguments, the Jacobi functions return themselves unevaluated:

```
jacobiSN(2,1/2)
```

$$\mathrm{sn}\left(2 \mid \frac{1}{2}\right)$$

Floating-point numbers are returned if at least one of the arguments is a floating-point number:

```
jacobiCN(1.5,1/2)
```

$$0.2502702593$$

Floating-point evaluation can be enforced by using `float`:

```
float(jacobiND(1,-1))
```

$$0.7404586624$$

Example 2

For $m = 0$ and $m = 1$, the result is expressed using a trigonometric function:

```
jacobiSC(u,0)
```

$$\tan(u)$$

```
jacobiND(u,1)
```

$$\cosh(u)$$

Parameters

m

An arithmetical expression specifying the parameter.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`ellipticCK` | `ellipticF` | `ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiNS` | `jacobiSC` | `jacobiSD` | `jacobiSN`

jacobiDC

Jacobi elliptic function dc

Syntax

```
jacobiDC(u,m)
```

Description

`jacobiDC(u,m)` represents the Jacobi elliptic function dc.

Let $u = \mathrm{F}(\varphi \mid m)$. Then the Jacobi elliptic function dc is defined as follows:

$$\mathrm{dc}(u \mid m) = \frac{\mathrm{dn}(u \mid m)}{\mathrm{cn}(u \mid m)}$$

The Jacobi functions are defined for complex values of u and m.

The Jacobi functions are meromorphic and doubly periodic with periods $4\,\mathrm{K}(m)$ and $4\,\mathrm{i}\,\mathrm{K}'(m)$ with respect to u.

For $m = 0$ and $m = 1$, the Jacobi functions reduce to trigonometric or constant functions.

If one argument is a floating-point number, and the other one can be converted to a floating-point number, then a floating-point number is returned.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

For most arguments, the Jacobi functions return themselves unevaluated:

```
jacobiSN(2,1/2)
```

$$\mathrm{sn}\left(2 \mid \frac{1}{2}\right)$$

Floating-point numbers are returned if at least one of the arguments is a floating-point number:

```
jacobiCN(1.5,1/2)
```

$$0.2502702593$$

Floating-point evaluation can be enforced by using `float`:

```
float(jacobiND(1,-1))
```

$$0.7404586624$$

Example 2

For $m = 0$ and $m = 1$, the result is expressed using a trigonometric function:

```
jacobiSC(u,0)
```

$$\tan(u)$$

```
jacobiND(u,1)
```

$$\cosh(u)$$

Parameters

m

An arithmetical expression specifying the parameter.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`ellipticCK` | `ellipticF` | `ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSD` | `jacobiSN`

jacobiNC

Jacobi elliptic function nc

Syntax

```
jacobiNC(u,m)
```

Description

`jacobiNC(u,m)` represents the Jacobi elliptic function nc.

Let $u = \mathrm{F}(\varphi \mid m)$. Then the Jacobi elliptic function nc is defined as follows:

$$\mathrm{nc}(u \mid m) = \frac{1}{\mathrm{cn}(u \mid m)}$$

The Jacobi functions are defined for complex values of u and m.

The Jacobi functions are meromorphic and doubly periodic with periods $4\,\mathrm{K}(m)$ and $4\,\mathrm{i}\,\mathrm{K}'(m)$ with respect to u.

For $m = 0$ and $m = 1$, the Jacobi functions reduce to trigonometric or constant functions.

If one argument is a floating-point number, and the other one can be converted to a floating-point number, then a floating-point number is returned.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

For most arguments, the Jacobi functions return themselves unevaluated:

```
jacobiSN(2,1/2)
```

$$\mathrm{sn}\left(2 \mid \frac{1}{2}\right)$$

Floating-point numbers are returned if at least one of the arguments is a floating-point number:

```
jacobiCN(1.5,1/2)
```

$$0.2502702593$$

Floating-point evaluation can be enforced by using `float`:

```
float(jacobiND(1,-1))
```

$$0.7404586624$$

Example 2

For $m = 0$ and $m = 1$, the result is expressed using a trigonometric function:

```
jacobiSC(u,0)
```

$$\tan(u)$$

```
jacobiND(u,1)
```

$$\cosh(u)$$

Parameters

m

An arithmetical expression specifying the parameter.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`ellipticCK` | `ellipticF` | `ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSD` | `jacobiSN`

jacobiSC

Jacobi elliptic function sc

Syntax

```
jacobiSC(u,m)
```

Description

`jacobiSC(u,m)` represents the Jacobi elliptic function sc.

Let $u = \mathrm{F}(\varphi \mid m)$. Then the Jacobi elliptic function sc is defined as follows:

$$\mathrm{sc}(u \mid m) = \frac{\mathrm{sn}(u \mid m)}{\mathrm{cn}(u \mid m)}$$

The Jacobi functions are defined for complex values of u and m.

The Jacobi functions are meromorphic and doubly periodic with periods $4\,\mathrm{K}(m)$ and $4\,\mathrm{i}\,\mathrm{K}'(m)$ with respect to u.

For $m = 0$ and $m = 1$, the Jacobi functions reduce to trigonometric or constant functions.

If one argument is a floating-point number, and the other one can be converted to a floating-point number, then a floating-point number is returned.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

For most arguments, the Jacobi functions return themselves unevaluated:

```
jacobiSN(2,1/2)
```

$$\mathrm{sn}\left(2 \mid \frac{1}{2}\right)$$

Floating-point numbers are returned if at least one of the arguments is a floating-point number:

```
jacobiCN(1.5,1/2)
```

$$0.2502702593$$

Floating-point evaluation can be enforced by using `float`:

```
float(jacobiND(1,-1))
```

$$0.7404586624$$

Example 2

For $m = 0$ and $m = 1$, the result is expressed using a trigonometric function:

```
jacobiSC(u,0)
```

$$\tan(u)$$

```
jacobiND(u,1)
```

$$\cosh(u)$$

Parameters

m

An arithmetical expression specifying the parameter.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`ellipticCK` | `ellipticF` | `ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSD` | `jacobiSN`

jacobiNS

Jacobi elliptic function ns

Syntax

```
jacobiNS(u,m)
```

Description

`jacobiNS(u,m)` represents the Jacobi elliptic function ns.

Let $u = \mathrm{F}(\varphi \mid m)$. Then the Jacobi elliptic function ns is defined as follows:

$$\mathrm{ns}(u \mid m) = \frac{1}{\mathrm{sn}(u \mid m)}$$

The Jacobi functions are defined for complex values of u and m.

The Jacobi functions are meromorphic and doubly periodic with periods $4\,\mathrm{K}(m)$ and $4\,\mathrm{i}\,\mathrm{K}'(m)$ with respect to u.

For $m = 0$ and $m = 1$, the Jacobi functions reduce to trigonometric or constant functions.

If one argument is a floating-point number, and the other one can be converted to a floating-point number, then a floating-point number is returned.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

For most arguments, the Jacobi functions return themselves unevaluated:

```
jacobiSN(2,1/2)
```

$$\mathrm{sn}\left(2 \mid \frac{1}{2}\right)$$

Floating-point numbers are returned if at least one of the arguments is a floating-point number:

```
jacobiCN(1.5,1/2)
```

$$0.2502702593$$

Floating-point evaluation can be enforced by using `float`:

```
float(jacobiND(1,-1))
```

$$0.7404586624$$

Example 2

For $m = 0$ and $m = 1$, the result is expressed using a trigonometric function:

```
jacobiSC(u,0)
```

$$\tan(u)$$

```
jacobiND(u,1)
```

$$\cosh(u)$$

Parameters

m

An arithmetical expression specifying the parameter.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`ellipticCK` | `ellipticF` | `ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiSC` | `jacobiSD` | `jacobiSN`

jacobiDS

Jacobi elliptic function ds

Syntax

```
jacobiDS(u,m)
```

Description

`jacobiDS(u,m)` represents the Jacobi elliptic function ds.

Let $u = \mathrm{F}(\varphi \mid m)$. Then the Jacobi elliptic function ds is defined as follows:

$$\mathrm{ds}(u \mid m) = \frac{\mathrm{dn}(u \mid m)}{\mathrm{sn}(u \mid m)}$$

The Jacobi functions are defined for complex values of u and m.

The Jacobi functions are meromorphic and doubly periodic with periods $4\,\mathrm{K}(m)$ and $4\,\mathrm{i}\,\mathrm{K}'(m)$ with respect to u.

For $m = 0$ and $m = 1$, the Jacobi functions reduce to trigonometric or constant functions.

If one argument is a floating-point number, and the other one can be converted to a floating-point number, then a floating-point number is returned.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

For most arguments, the Jacobi functions return themselves unevaluated:

```
jacobiSN(2,1/2)
```

$$\mathrm{sn}\left(2 \mid \frac{1}{2}\right)$$

Floating-point numbers are returned if at least one of the arguments is a floating-point number:

```
jacobiCN(1.5,1/2)
```

$$0.2502702593$$

Floating-point evaluation can be enforced by using `float`:

```
float(jacobiND(1,-1))
```

$$0.7404586624$$

Example 2

For $m = 0$ and $m = 1$, the result is expressed using a trigonometric function:

```
jacobiSC(u,0)
```

$$\tan(u)$$

```
jacobiND(u,1)
```

$$\cosh(u)$$

Parameters

m

An arithmetical expression specifying the parameter.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`ellipticCK` | `ellipticF` | `ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSD` | `jacobiSN`

jacobiCS

Jacobi elliptic function cs

Syntax

```
jacobiCS(u,m)
```

Description

`jacobiCS(u,m)` represents the Jacobi elliptic function cs.

Let $u = \mathrm{F}(\varphi \mid m)$. Then the Jacobi elliptic function cs is defined as follows:

$$\mathrm{cs}(u \mid m) = \frac{\mathrm{cn}(u \mid m)}{\mathrm{sn}(u \mid m)}$$

The Jacobi functions are defined for complex values of u and m.

The Jacobi functions are meromorphic and doubly periodic with periods $4\,\mathrm{K}(m)$ and $4\,\mathrm{i}\,\mathrm{K}'(m)$ with respect to u.

For $m = 0$ and $m = 1$, the Jacobi functions reduce to trigonometric or constant functions.

If one argument is a floating-point number, and the other one can be converted to a floating-point number, then a floating-point number is returned.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

For most arguments, the Jacobi functions return themselves unevaluated:

```
jacobiSN(2,1/2)
```

$$\mathrm{sn}\left(2 \mid \frac{1}{2}\right)$$

Floating-point numbers are returned if at least one of the arguments is a floating-point number:

```
jacobiCN(1.5,1/2)
```

$$0.2502702593$$

Floating-point evaluation can be enforced by using `float`:

```
float(jacobiND(1,-1))
```

$$0.7404586624$$

Example 2

For $m = 0$ and $m = 1$, the result is expressed using a trigonometric function:

```
jacobiSC(u,0)
```

$$\tan(u)$$

```
jacobiND(u,1)
```

$$\cosh(u)$$

Parameters

m

An arithmetical expression specifying the parameter.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`ellipticCK` | `ellipticF` | `ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSD` | `jacobiSN`

jacobian

Jacobian matrix of a vector function

Syntax

```
jacobian(v, x)
```

Description

`jacobian(v, x)` computes the Jacobian matrix of the vector function $\vec{v}$ with respect to $\vec{x}$.

If `v` is a vector then the component ring of `v` must be a field (i.e., a domain of category `Cat::Field`) for which differentiation with respect to `x` is defined.

If `v` is given as a list of arithmetical expressions, then `jacobian` returns a matrix with the standard component ring `Dom::ExpressionField()`.

Examples

Example 1

The Jacobian matrix of the vector function $\vec{v} = \begin{pmatrix} x^3 \\ x\,z \\ y+z \end{pmatrix}$ is:

```
delete x, y, z:
jacobian([x^3, x*z, y+z], [x, y, z])
```

$$\begin{pmatrix} 3x^2 & 0 & 0 \\ z & 0 & x \\ 0 & 1 & 1 \end{pmatrix}$$

Parameters

v

A list of arithmetical expressions, or a vector (i.e., an $n \times 1$ or $1 \times n$ matrix of a domain of category `Cat::Matrix`)

x

A list of (indexed) identifiers

Return Values

Matrix of the domain `Dom::Matrix(R)`, where `R` is the component ring of `v` or the domain `Dom::ExpressionField()`.

Algorithms

For a vector function $\vec{v}: G \to R^m$, where G is a subset of $\mathbb{R}^n$ the matrix

$$H_f(\vec{x}) = \begin{pmatrix} \frac{\partial}{\partial x_1} v_1 & \frac{\partial}{\partial x_2} v_1 & \cdots & \frac{\partial}{\partial x_n} v_1 \\ \frac{\partial}{\partial x_1} v_2 & \frac{\partial}{\partial x_2} v_2 & \cdots & \frac{\partial}{\partial x_n} v_2 \\ \vdots & \vdots & & \vdots \\ \frac{\partial}{\partial x_1} v_m & \frac{\partial}{\partial x_2} v_m & \cdots & \frac{\partial}{\partial x_n} v_m \end{pmatrix}$$

is the *Jacobian matrix* of $\vec{v}$.

See Also

MuPAD Functions

`gradient` | `hessian`

jacobiZeta

Jacobi Zeta function

Syntax

```
jacobiZeta(u, m)
```

Description

`jacobiZeta(u,m)` represents the Jacobi Zeta function $Z(u \mid m)$ which is defined as

$$Z(u \mid m) = \frac{2\,\pi}{K(m)} \left(\sum_{s=1}^{\infty} \frac{q(m)^s}{1 - q(m)^{2\,s}} \sin\left(\frac{2\,\pi}{K(m)}\, s\, u \right) \right).$$

The Jacobi Zeta function $Z(u \mid m)$ is defined for complex arguments u and m.

Exact results are returned for $m = 0$, $m = 1$ or $u = 0$. In all other cases an unevaluated symbolic call is returned.

A floating-point value is computed if both arguments are numerical and at least one is a floating-point number.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Most calls with exact arguments are returned unevaluated:

```
jacobiZeta(2, -1)
```

$$Z(2 \mid -1)$$

If $m = 0$, $m = 1$ or $u = 0$, an exact result is returned:

```
jacobiZeta(0, 3)
```

$$0$$

```
jacobiZeta(1, 0)
```

$$0$$

```
jacobiZeta(2, 1)
```

$$\tanh(2)$$

Parameters

u

An arithmetical expression.

m

An arithmetical expression specifying the parameter.

Return Values

Arithmetical expression.

kroneckerDelta

Kronecker's delta symbol

Syntax

```
kroneckerDelta(m, <n>)
```

Description

`kroneckerDelta(m, n)` is Kronecker's delta symbol. It represents 1 if $m = n$ and 0 if $m \neq n$.

`kroneckerDelta(m)` represents 1 if $m = 0$ and 0 if $m \neq 0$.

The calls `kroneckerDelta(m, n)` and `kroneckerDelta(m - n)` are equivalent.

`kroneckerDelta(m, n)` yields `1` if the arguments `m`, `n` coincide.

It yields `0` if `m - n` yields a non-zero numerical value.

If either `m` or `n` contain symbolic objects and `m - n` does not yield a numerical value, then the symbolic call `kroneckerDelta(m, n)` or the equivalent call `kroneckerDelta(n, m)` is returned.

Floating point numbers such as 1.0, 2.0 etc. are treated like integers.

Note that `kroneckerDelta(m,n)` = `kroneckerDelta(n,m)` for arbitary arguments `m`, `n`. In symbolic return values, the ordering of the input arguments may be exchanged.

`kroneckerDelta` is used and processed by `sum` and `ztrans`, `iztrans`.

Examples

Example 1

`kroneckerDelta` returns 1 or 0, respectively, for arguments that definitely coincide or do not coincide:

```
kroneckerDelta(2, 2), kroneckerDelta(n, n),
kroneckerDelta(2, 3), kroneckerDelta(n - 1, n + 1)
```

$$1, 1, 0, 0$$

A symbolic call is returned if the system cannot decide whether the arguments coincide:

```
kroneckerDelta(m, n), kroneckerDelta(m, 3), kroneckerDelta(3, n)
```

$$\delta_{m-n,0}, \delta_{m-3,0}, \delta_{n-3,0}$$

Example 2

`kroneckerDelta` is processed by `sum`:

```
sum(a[n]*kroneckerDelta(n, 3), n = 0..infinity)
```

$$a_3$$

```
sum(a[n]*kroneckerDelta(n, m), n = 0..infinity)
```

$$\begin{cases} a_m & \text{if } 0 \le m \wedge m \in \mathbb{Z} \\ 0 & \text{if } m \le -1 \vee m \notin \mathbb{Z} \end{cases}$$

`iztrans` may produce terms involving `kroneckerDelta`:

```
iztrans(1/(z - 1), z, n)
```

$$1 - \delta_{n,0}$$

```
ztrans(%, n, z)
```

$$\frac{z}{z-1} - 1$$

Parameters

`m, n`

arithmetical expressions. The default value for n is 0.

Return Values

Arithmetical expression.

Overloaded By

m, n

See Also

MuPAD Functions
`iztrans` | `sum` | `ztrans`

kummerU

Confluent hypergeometric KummerU function

Syntax

```
kummerU(a, b, z)
```

Description

`kummerU(a, b, z)` represents the KummerU function $U(a, b, z)$, whose integral representation is given by

$$\frac{1}{\Gamma(a)} \int_0^\infty e^{-z\,t}\, t^{a-1}\, (1+t)^{b-a-1}\, dt$$

for $\Re(a) > 0$ and $\Re(z) > 0$.

`kummerU` is defined for complex arguments a, b, and z.

For most parameter values, an unevaluated function call is returned. Cf. “Example 1” on page 1-1168.

Explicit symbolic expressions are returned for some particular values of the parameters:

- If $b = 2\,a$, the `besselK` function may appear.
- If a is a negative integer, the result is a polynomial.
- If $a = 1$ or $b = a$, the `igamma` function may appear.

Cf. “Example 2” on page 1-1168.

Environment Interactions

When called with floating-point arguments, this function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Unevaluated calls are returned for exact or symbolic arguments:

```
kummerU(a, b, x), kummerU(1/2, -1, 0)
```

$$\text{kummerU}(a, b, x), \text{kummerU}\left(\frac{1}{2}, -1, 0\right)$$

Floating point values are returned for floating-point arguments:

```
kummerU(1/3, 2.0, -50), kummerU(1/2, -1, 0.0)
```

$$0.1351149381 - 0.2340259377\,i,\ 0.7522527781$$

Example 2

Explicit expressions are returned for some specific values of the parameters:

```
kummerU(1/2, 1, x), kummerU(-2, b, x),
kummerU(1, 1/3, x), kummerU(a, a, x)
```

$$\frac{e^{\frac{x}{2}}\,K_0\left(\frac{x}{2}\right)}{\sqrt{\pi}},\ b - 2x(b+1) + b^2 + x^2,\ x^{2/3}\,e^x\left(\frac{3\,e^{-x}}{2\,x^{2/3}} - \frac{3\,\Gamma\left(\frac{1}{3}, x\right)}{2}\right),\ e^x\,\Gamma(1-a, x)$$

Example 3

The functions `diff`, `float`, `limit`, and `series` handle expressions involving the `kummerU` function

```
diff(kummerU(a, b, z), z), float(kummerU(1/2, -1, 0))
```

$$\frac{a\,\text{kummerU}(a+1, b, z)\,(a-b+1)}{z} - \frac{a\,\text{kummerU}(a, b, z)}{z},\ 0.7522527781$$

```
limit(kummerU(1/2, -1, x), x),
series(kummerU(1/2, -1, x), x = infinity, 3)
```

$$\frac{4}{3\sqrt{\pi}},\ \frac{1}{\sqrt{x}} - \frac{5}{4x^{3/2}} + \frac{105}{32x^{5/2}} + O\left(\frac{1}{x^{7/2}}\right)$$

Parameters

a, b, z

arithmetical expressions

Return Values

Arithmetical expression.

Overloaded By

z

Algorithms

$U(a, b, z)$ satisfies Kummer's differential equation:

$$z\frac{\partial^2}{\partial z^2}y + (b-z)\frac{\partial}{\partial z}y - a\,y = 0$$

,

for which the hypergeometric function ${}_1F_1(a, b, z)$ is another solution.

$U(a, b, z)$ is related to the `whittakerW` function $W_{a,b}(z)$ by the formula:

$$W_{a,b}(z) = e^{-\frac{z}{2}} z^{\frac{1}{2}+b} U\left(\frac{1}{2}+b-a,\ 1+2b,\ z\right)$$

See Also

MuPAD Functions
`hypergeom` | `whittakerM` | `whittakerW`

laguerreL

Laguerre polynomials and L function

Syntax

```
laguerreL(n, x)
```

```
laguerreL(n, a, x)
```

Description

`laguerreL(n, a, x)` represents Laguerre's L function. When `n` is a nonnegative integer, this is the classical Laguerre polynomial of degree n.

Laguerre's L function is defined in terms of hypergeometric functions by

$$\text{laguerreL}(n, a, x) = \binom{n+a}{a} {}_1F_1(-n; a+1; x)$$

.

For nonnegative integer values of n, the function returns the classical (generalized) polynomials that are orthogonal with respect to the scalar product $\langle \text{f1}, \text{f2} \rangle = \int_0^\infty e^{-x} x^a \, \text{f1}(x) \, \text{f2}(x) \, dx$. In particular:

$$\langle \text{laguerreL}(n, a, x), \text{laguerreL}(m, a, x) \rangle = \begin{cases} 0 & \text{if } n \neq m \\ \frac{\Gamma(a+n+1)}{n!} & \text{if } n = m \end{cases}$$

.

The Laguerre's L function is not well defined for all values of the parameters n and a, because certain restrictions on the parameters exist in the definition of the hypergeometric functions . If the Laguerre's L function is not defined for a particular pair n and a, the call `laguerreL(n, a, x)` returns 0 or issues an error message.

The calls `laguerre(n, x)` and `laguerre(n, 0, x)` are equivalent.

If n is a nonnegative integer, the function `laguerreL` returns the explicit form of the corresponding Laguerre polynomial. The special values $\text{laguerreL}(n, a, 0) = \binom{n+a}{a}$ are implemented for arbitrary values of n and a. If n is a negative integer and a is a numerical noninteger value satisfying $a \geq -n$, then the function `laguerreL` returns 0. If n is a negative integer and a is an integer satisfying $a < -n$, then the function returns an explicit expression defined by the reflection rule

$$\text{laguerreL}(n, a, x) = (-1)^a e^x \text{laguerreL}(-n-a-1, a, -x).$$

If all arguments are numerical and at least one of the arguments is a floating-point number, then `laguerreL(x)` returns a floating-point number. For all other arguments, `laguerreL(n, a, x)` returns a symbolic function call.

Environment Interactions

When called with floating-point arguments, the function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

You can call the `laguerreL` function with exact and symbolic arguments:

```
laguerreL(2, a, x), laguerreL(-2, -2, PI)
```

$$\frac{3a}{2} - x(a+2) + \frac{a^2}{2} + \frac{x^2}{2} + 1,\ e^{\pi}\left(\frac{\pi^2}{2} + \frac{\pi^3}{6}\right)$$

If the first argument is a nonnegative integer, the function returns a polynomial:

```
laguerreL(3, x)
```

$$-\frac{x^3}{6} + \frac{3x^2}{2} - 3x + 1$$

```
laguerreL(3, a, x)
```

$$\frac{11\,a}{6} - x\left(\frac{a^2}{2} + \frac{5\,a}{2} + 3\right) + x^2\left(\frac{a}{2} + \frac{3}{2}\right) + a^2 + \frac{a^3}{6} - \frac{x^3}{6} + 1$$

Floating-point values are computed for floating-point arguments:

```
laguerreL(2, 3, 4.0), laguerreL(5.0, sqrt(2), PI)
```

$$-2.0,\ 1.851157209$$

```
laguerreL(1 + I, 1.0), laguerreL(-2.0, exp(I))
```

$$-0.2457246594 - 0.6867435489\,i,\ 0.6848682701 + 2.933911244\,i$$

Example 2

The Laguerre function is not defined for all parameter values:

```
laguerreL(-5/2, -3/2, x)
```

```
Error: The function 'laguerreL' is not defined for parameter values '-5/2' and '-3/2'.
```

Example 3

System functions such as `diff`, `float`, `limit`, and `series` handle expressions involving `laguerreL`:

```
diff(laguerreL(n, a, x), x, x, x), float(laguerreL(2, 3, sqrt(PI)))
```

$$-\mathrm{laguerreL}(n-3,\ a+3,\ x),\ 2.708527072$$

```
limit(laguerreL(3, 4, x^2/(1+x)), x = infinity)
```

$$-\infty$$

```
limit(laguerreL(4, 3, x^2/(1+x)), x = infinity)
```

$$\infty$$

```
series(laguerreL(n, a, x), x = 0, 3)
```

$$\binom{a+n}{n} - \frac{n\,x\,\binom{a+n}{n}}{a+1} + \frac{n\,x^2\,(n-1)\,\binom{a+n}{n}}{2\,(a+1)\,(a+2)} + O(x^3)$$

```
series(laguerreL(3/2, x), x = infinity, 3)
```

$$\frac{3\,e^x}{4\,\sqrt{\pi}\,x^{5/2}} + \frac{75\,e^x}{16\,\sqrt{\pi}\,x^{7/2}} + \frac{3675\,e^x}{128\,\sqrt{\pi}\,x^{9/2}} + O\left(\frac{e^x}{x^{11/2}}\right)$$

Parameters

n, a, x

arithmetical expressions

Return Values

Arithmetical expression.

Overloaded By

x

See Also

MuPAD Functions
`hypergeom` | `orthpoly::laguerre`

lambertW

The Lambert function

Syntax

```
lambertW(x)
```

```
lambertW(k, x)
```

Description

For integer k, the values $y = \mathrm{W}_k(x)$ represent the solutions of the equation $y\,e^y = x$.

`lambertW` is the inverse function of $y \to y\,e^y$.

In the complex plane, the equation $y\,e^y = x$ has a countably infinite number of solutions. They are represented by `lambertW(k, x)` with `k` ranging over the integers.

For all real $x \geq 0$, the equation $y\,\mathrm{e}^y = x$ has exactly one real solution. It is represented by `y=lambertW(x)` or, equivalently, `y=lambertW(0, x)`.

For all real x in the range $0 > x$, there are exactly two real solutions. The larger one is represented by `y=lambertW(x)`, the smaller one by `y=lambertW(-1, x)`.

Exactly one real solution `lambertW(0, -exp(-1))= lambertW(-1, -exp(-1))= -1` exists for $x = -\mathrm{e}^{-1}$.

For $k \notin \{0, -1\}$, `lambertW(k, x)` takes no real value.

The values `lambertW(-1, 0)=- infinity` and `lambertW(0, 0)=0` are implemented. Further, the result y is returned for some exact arguments of the form $x = y\,\mathrm{e}^y$. For floating-point arguments a floating-point value is returned. For all other arguments, unevaluated function calls are returned.

The `float` attributes are kernel functions, i.e., floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
lambertW(-3), lambertW(-1, -5/2), lambertW(1/2),
lambertW(5, I), lambertW(3, 1 + I), lambertW(-1, x + 1)
```

$$W_0(-3),\ W_{-1}\left(-\frac{5}{2}\right),\ W_0\left(\frac{1}{2}\right),\ W_5(i),\ W_3(1+i),\ W_{-1}(x+1)$$

Some exact values are found:

```
lambertW(-1, -exp(-1)), lambertW(-1, -2*exp(-2)),
lambertW(-1, -3/2*exp(-3/2)), lambertW(exp(1)),
lambertW(2*exp(2)), lambertW(5/2*exp(5/2)),
lambertW(1, (3+4*I)*exp(3+4*I))
```

$$-1,\ -2,\ -\frac{3}{2},\ 1,\ 2,\ \frac{5}{2},\ 3+4\,i$$

Floating point values are computed for floating-point arguments:

```
lambertW(-1, -0.3), lambertW(2000.0)
```

$$-1.781337023,\ 5.836731495$$

```
lambertW(-3, -0.277), lambertW(1, 2345.6)
```

$$-3.951784369 - 13.85940405\,i,\ 5.690470601 + 5.513574431\,i$$

Example 2

The functions `diff`, `float`, and `series` handle expressions involving the Lambert function:

```
diff(lambertW(k, x), x)
```

$$\frac{W_k(x)}{x\,(W_k(x)+1)}$$

```
float(ln(3 + lambertW(sqrt(PI))))
```

$$1.334475971$$

```
series(lambertW(x), x = 0);
series(lambertW(x), x = -1/exp(1), 3);
series(lambertW(-1, x), x = -1/exp(1), 3);
```

$$x - x^2 + \frac{3\,x^3}{2} - \frac{8\,x^4}{3} + \frac{125\,x^5}{24} - \frac{54\,x^6}{5} + O(x^7)$$

$$-1 + \sqrt{2}\,\sqrt{e}\,\sqrt{x + e^{-1}} - \frac{2\,e\left(x + e^{-1}\right)}{3} + O\left(\left(x + e^{-1}\right)^{3/2}\right)$$

$$-1 - \sqrt{2}\,\sqrt{e}\,\sqrt{x + e^{-1}} - \frac{2\,e\left(x + e^{-1}\right)}{3} + O\left(\left(x + e^{-1}\right)^{3/2}\right)$$

Parameters

x

An arithmetical expression, the “argument”

k

An arithmetical expression representing an integer, the “branch”

Return Values

Arithmetical expression.

References

R.M. Corless, D.J. Jeffrey and D.E. Knuth: “A sequence of Series for the Lambert W Function”, in: Proceedings of ISSAC'97, Maui, Hawaii. W.W. Kuechlin (ed.). New York: ACM, pp. 197-204, 1997.

laplace

Laplace transform

Syntax

```
laplace(f, t, s)
```

Description

`laplace(f, t, s)` computes the Laplace transform of the expression `f = f(t)` with respect to the variable *t* at the point *s*.

The Laplace transform is defined as follows:

$$F(s) = \int_0^\infty f(t)\, e^{-s\,t}\, dt$$

.

If `laplace` cannot find an explicit representation of the transform, it returns an unevaluated function call. See "Example 3" on page 1-1180.

If `f` is a matrix, `laplace` applies the Laplace transform to all components of the matrix.

To compute the inverse Laplace transform, use `ilaplace`.

Examples

Example 1

Compute the Laplace transforms of these expressions with respect to the variable `t`:

```
laplace(exp(-a*t), t, s)
```

$$\frac{1}{a+s}$$

```
laplace(1 + exp(-a*t)*sin(b*t), t, s)
```

$$\frac{1}{s} + \frac{b}{(a+s)^2 + b^2}$$

Example 2

Compute the Laplace transform of this expression with respect to the variable `t`:

```
F := laplace(t^10*exp(-s_0*t), t, s)
```

$$\frac{3628800}{(s+s_0)^{11}}$$

Evaluate the Laplace transform of the expression at the points $s = -2\,s_0$ and $s = 1 + \pi$. You can evaluate the resulting expression `F` using `|` (or its functional form `evalAt`):

```
F | s = -2*s_0
```

$$-\frac{3628800}{s_0^{11}}$$

Also, you can evaluate the Laplace transform at a particular point directly:

```
laplace(t^10*exp(-s_0*t), t, 1 + PI)
```

$$\frac{3628800}{(\pi + s_0 + 1)^{11}}$$

Example 3

If `laplace` cannot find an explicit representation of the transform, it returns an unevaluated call:

```
laplace(exp(-t^3), t, s)
```

$$\text{laplace}\left(e^{-t^3}, t, s\right)$$

`ilaplace` returns the original expression:

```
ilaplace(%, s, t)
```

$$e^{-t^3}$$

Example 4

Compute the folllowing Laplace transforms that involve the Dirac and the Heaviside functions:

```
laplace(dirac(t - 3), t, s)
```

$$e^{-3s}$$

```
laplace(heaviside(t - PI), t, s)
```

$$\frac{e^{-\pi s}}{s}$$

Example 5

The Laplace transform of a function is related to the Laplace transform of its derivative:

```
laplace(diff(f(t), t), t, s)
```

$$s\,\mathrm{laplace}(f(t), t, s) - f(0)$$

Parameters

`f`

Arithmetical expression or matrix of such expressions

`t`

Identifier or indexed identifier representing the transformation variable

s

Arithmetical expression representing the evaluation point

Return Values

Arithmetical expression or unevaluated function call of type `laplace`. If the first argument is a matrix, then the result is returned as a matrix.

Overloaded By

f

See Also

MuPAD Functions

`ilaplace` | `ilaplace::addpattern` | `laplace::addpattern`

More About

- "Integral Transforms"

laplace::addpattern

Add patterns for the Laplace transform

Syntax

```
laplace::addpattern(pat, t, s, res, <vars, <conds>>)
```

Description

`laplace::addpattern(pat, t, s, res)` teaches `laplace` to return

$\mathrm{laplace}(\mathrm{pat}, t, s) = \int_0^{\infty} \mathrm{pat}\, e^{-s\,t}\, dt = \mathrm{res}$.

The `laplace` function uses a set of patterns for computing Laplace transforms. You can extend the set by adding your own patterns. To add a new pattern to the pattern matcher, use `laplace::addpattern`. MuPAD does not save custom patterns permanently. The new patterns are available in the *current* MuPAD session only.

Variable names that you use when calling `laplace::addpattern` can differ from the names that you use when calling `laplace`. See “Example 2” on page 1-1184.

You can include a list of free parameters and a list of conditions on these parameters. These conditions and the result are protected from premature evaluation. This means that you can use `not iszero(a^2 - b)` instead of `hold( _not @ iszero )(a^2 - b)`.

The following conditions treat assumptions on identifiers differently:

- `a^2 - b <> 0` takes into account assumptions on identifiers.
- `not iszero(a^2 - b)` disregards assumptions on identifiers.

See “Example 4” on page 1-1185.

Environment Interactions

Calling `laplace::addpattern` changes the expressions returned by future calls to `laplace`.

Examples

Example 1

Compute the Laplace transform of the function `foo`. By default, MuPAD does not have a pattern for this function:

```
laplace(foo(t), t, s)
```

$$\text{laplace}(\text{foo}(t), t, s)$$

Add a pattern for the Laplace transform of `foo` using `laplace::addpattern`:

```
laplace::addpattern(foo(t), t, s, bar(s)):
```

Now `laplace` returns the Laplace transform of `foo`:

```
laplace(foo(t), t, s)
```

$$\text{bar}(s)$$

After you add a new transform pattern, MuPAD can use that pattern indirectly:

```
laplace(t^3 + exp(2*t)*foo(t), t, s)
```

$$\text{bar}(s-2)+\frac{6}{s^4}$$

Example 2

Define the Laplace transform of `foo(x)` using the variables `x` and `y` as parameters:

```
laplace::addpattern(foo(x), x, y, bar(y)):
```

The `laplace` function recognizes the added pattern even if you use other variables as parameters:

```
laplace(foo(t), t, s)
```

$$\text{bar}(s)$$

Example 3

Add this pattern for the Laplace transform of `f`:

```
laplace::addpattern(f(a*x)*g(a*x), x, y, y/(y^4 + 4*a^4)):
laplace(f(a*v)*g(a*v), v, w)
```

$$\frac{w}{4\,a^4 + w^4}$$

This pattern holds only when the first argument of `f` is the symbolic parameter `a`. If you use any other value of this parameter, `laplace` ignores the pattern:

```
laplace(f(A*v)*g(A*v), v, w)
```

$$\text{laplace}(f(A\,v)\,g(A\,v),\,v,\,w)$$

To use the pattern for arbitrary values of the parameter, declare the parameter `a` as an additional pattern variable:

```
laplace::addpattern(f(a*x)*g(a*x), x, y, y/(y^4 + 4*a^4), [a]):
```

Now `laplace` applies the specified pattern for an arbitrary value of `a`:

```
laplace(f(A*v)*g(A*v), v, w)
```

$$\frac{w}{4\,A^4 + w^4}$$

Example 4

Use assumptions when adding the following pattern for the Laplace transform:

```
laplace::addpattern(FOO(x*t), t, s, sin(1/(x-1/2))*BAR(s),
                                    [x], [abs(x) < 1]):
laplace(FOO(x*t),t,s) assuming -1 < x < 1
```

$$\sin\left(\frac{1}{x-\frac{1}{2}}\right)\,\text{BAR}(s)$$

If $|x| \geq 1$, you cannot apply this pattern:

```
laplace(FOO(x*t),t,s) assuming x >= 1
```

$$\text{laplace}(\text{FOO}(t\,x),\ t,\ s)$$

If MuPAD cannot determine whether the conditions are satisfied, it returns a `piecewise` object:

```
laplace(FOO(x*t), t, s)
```

$$\begin{cases} \sin\left(\frac{1}{x-\frac{1}{2}}\right)\ \text{BAR}(s) & \text{if } |x| < 1 \end{cases}$$

Note that the resulting expression defining the Laplace transform of `FOO(x*t)` implicitly assumes that the value of `x` is not 1/2. A strict definition of the pattern is:

```
laplace::addpattern(FOO(x*t), t, s, sin(1/(x-1/2))*BAR(s),
                    [x], [abs(x) < 1, x <> 1/2]):
```

If either the conditions are not satisfied or substituting the values into the result gives an error, `laplace` ignores the pattern. For this particular pattern, you can omit specifying the assumption `x <> 1/2`. If `x = 1/2`, MuPAD throws an internal "Division by zero." error and ignores the pattern:

```
laplace(FOO(1/2*t), t, s)
```

$$\text{laplace}\left(\text{FOO}\left(\frac{t}{2}\right),\ t,\ s\right)$$

Parameters

`pat`

Arithmetical expression in the variable `t` representing the pattern to match

`t`

Identifier or indexed identifier used as a variable in the pattern

`s`

Identifier or indexed identifier used as a variable in the result

`res`

Arithmetical expression in the variable `s` representing the pattern for the result of the transformation

`vars`

List of identifiers or indexed identifiers used as "pattern variables" (placeholders in `pat` and `res`). You can use pattern variables as placeholders for almost arbitrary MuPAD expressions not containing `t` or `s`. You can restrict them by conditions given in the optional parameter `conds`.

`conds`

List of conditions on the pattern variables

Return Values

Object of type `DOM_NULL`

See Also

MuPAD Functions

`ilaplace` | `ilaplace::addpattern` | `laplace`

laplacian

The Laplacian

Syntax

```
laplacian(f, [x1, x2, …])
```

```
laplacian(f, [x1, x2, …], ogCoord, <c>)
```

Description

`laplacian(f, [ x1, x2, ...])` computes the Laplacian $\Delta f = \sum_i \frac{\partial^2}{\partial x_i^2} f$, i.e. *div*(*grad*(*f*)), of the function $f = f(x_1, x_2, \ldots)$ in the Cartesian coordinates $x_1, x_2, \ldots$.

The table `linalg::ogCoordTab` provides some predefined three-dimensional orthogonal coordinate transformations. Presently, its entries are `Cartesian`, `Cylindrical`, `Spherical[RightHanded]` (or its equivalent `Spherical[RightHanded]`), `EllipticCylindrical`, `ParabolicCylindrical`, `RotationParabolic`, `Torus`. See `linalg::ogCoordTab` for details. For example, the command

```
laplacian(f(r, Theta, phi), [r, Theta, phi], Spherical[RightHanded])
```

produces the Laplacian of *f* in spherical coordinates r, θ, ϕ defined by the transformation

$$x = r \sin(\theta) \cos(\phi),\ y = r \sin(\theta) \sin(\phi),\ z = r \cos(\theta)$$

.

Arbitrary orthogonal systems $u = (u_1, \ldots, u_n)$ (in any dimension n) can be used by passing corresponding "scale parameters" as third argument to `laplacian`. These are defined as follows. Let $\vec{x} = (x_1, \ldots, x_n)$ be Cartesian coordinates, let $\vec{x}(\vec{u})$ be an orthognal transformation (i.e., the vectors $\frac{\partial}{\partial u_i} \vec{x}$ are orthogonal). The Euclidean lengths $s_i = \left| \frac{\partial}{\partial u_i} \vec{x} \right|$ of the vectors define the "scales". The list $s = [s_1, \ldots, s_n]$ can be passed as a

third argument to `laplacian`. For example, the usual two-dimensional polar coordinates $x = r\,cos(\phi)$, $y = r\,sin(\phi)$ lead to scale factors

$$s = \left[\left|\left(\frac{\partial}{\partial r}x, \frac{\partial}{\partial r}y\right)\right|, \left|\left(\frac{\partial}{\partial \phi}x, \frac{\partial}{\partial \phi}y\right)\right|\right] = [|(\cos(\phi), \sin(\phi))|, |(-r\sin(\phi), r\cos(\phi))|] =$$

$$[1, r]$$

.

Thus, `laplacian(f(r, phi), [r, phi], [1, r])` produces the Laplacian of $f(r, \phi)$ in polar coordinates r and ϕ.

Examples

Example 1

Compute the Laplacian in Cartesian coordinates:

```
laplacian(f(x[1], x[2]), [x[1], x[2]])
```

$$\frac{\partial^2}{\partial x_1^2} f(x_1, x_2) + \frac{\partial^2}{\partial x_2^2} f(x_1, x_2)$$

```
laplacian(x^2*y + c*exp(y) + u*v^2, [x, y, u, v])
```

$$2u + 2y + c\,e^y$$

Example 2

Compute the Laplacian in cylindrical coordinates (r, ϕ, z) given by

$$x = r\cos(\phi),\ y = r\sin(\phi),\ z = z$$

.

```
expand(laplacian(f(r, phi, z), [r, phi, z], Cylindrical))
```

$$\frac{\frac{\partial^2}{\partial \mathrm{phi}^2} f(r, \mathrm{phi}, z)}{r^2} + \frac{\frac{\partial}{\partial r} f(r, \mathrm{phi}, z)}{r} + \frac{\partial^2}{\partial r^2} f(r, \mathrm{phi}, z) + \frac{\partial^2}{\partial z^2} f(r, \mathrm{phi}, z)$$

```
laplacian(r*cos(phi)*z^3, [r, phi, z], Cylindrical)
```

$$6\,r\,z\,\cos(\mathrm{phi})$$

Passing the name `Cylindrical` of the orthogonal system predefined in `linalg::ogCoordTab` is the simplest way of using cylindrical coordinates. Alternatively, one may pass appropriate `scale parameters' explicitly. They are stored in `linalg::ogCoordTab` and can be called in the following way:

```
linalg::ogCoordTab[Cylindrical, Scales](r, phi, z)
```

$$[1,\ r,\ 1]$$

```
laplacian(r*cos(phi)*z^3, [r, phi, z], %)
```

$$6\,r\,z\,\cos(\mathrm{phi})$$

Example 3

Consider Torus coordinates (r, θ, ϕ) introduced by

$$x = (c - r\cos(\theta))\cos(\phi),\ y = (c - r\cos(\theta))\sin(\phi),\ z = r\sin(\theta)$$

Here, c is a real constant and $0 \le r < c$, $0 \le \theta \le 2\,\pi$, $0 \le \phi \le 2\,\pi$ is assumed. The “scale parameters” are stored in `linalg::ogCoordTab`:

```
linalg::ogCoordTab[Torus, Scales](r, thet, phi, c)
```

$$[1,\ r,\ c - r\cos(\mathrm{thet})]$$

The Laplacian of the function $f(r, \phi, z) = r$ in these coordinates is:

```
laplacian(r, [r, thet, phi], %)
```

$$\frac{c - 2\,r\cos(\mathrm{thet})}{r\,(c - r\cos(\mathrm{thet}))}$$

Example 4

You can introduce new orthogonal systems. For example, consider the orthogonal "6-sphere coordinates" (u, v, w) introduced by

$$x = \frac{u}{u^2+v^2+w^2},\ y = \frac{v}{u^2+v^2+w^2},\ z = \frac{w}{u^2+v^2+w^2}$$

.

This transformation $(u, v, w) \to \vec{x} = (x, y, z)$ is not stored in `linalg::ogCoordTab`, hence the corresponding "scale factors" of the metric have to be computed first:

$$\left|\frac{\partial}{\partial u}\vec{x}\right| = \left|\frac{\partial}{\partial v}\vec{x}\right| = \left|\frac{\partial}{\partial w}\vec{x}\right| = \frac{1}{u^2+v^2+w^2}$$

.

With these "scales", the Laplacian can be computed via `laplacian`:

```
s := 1/(u^2 + v^2 + w^2):
factor(laplacian(f(u, v, w), [u, v, w], [s, s, s]))
```

$$(u^2+v^2+w^2)\left(u^2\,\sigma_3+v^2\,\sigma_3+w^2\,\sigma_3-2\,u\,\frac{\partial}{\partial u}f(u, v, w)+u^2\,\sigma_2+v^2\,\sigma_2+w^2\,\sigma_2\right.$$

$$\left.-2\,v\,\frac{\partial}{\partial v}f(u, v, w)+u^2\,\sigma_1+v^2\,\sigma_1+w^2\,\sigma_1-2\,w\,\frac{\partial}{\partial w}f(u, v, w)\right)$$

where

$$\sigma_1 = \frac{\partial^2}{\partial w^2}f(u, v, w)$$

$$\sigma_2 = \frac{\partial^2}{\partial v^2}f(u, v, w)$$

$$\sigma_3 = \frac{\partial^2}{\partial u^2}f(u, v, w)$$

Since the Laplacian is the divergence of the gradient, you can compute it in the following way, too:

```
divergence(gradient(f(u, v, w), [u, v, w], [s, s, s]),
                    [u, v, w], [s, s, s])
```

$$\sigma_1^2 \frac{\partial^2}{\partial u^2} f(u, v, w) + \sigma_1^2 \frac{\partial^2}{\partial v^2} f(u, v, w) + \sigma_1^2 \frac{\partial^2}{\partial w^2} f(u, v, w) - 2\, u\, \sigma_1 \frac{\partial}{\partial u} f(u, v, w)$$

$$- 2\, v\, \sigma_1 \frac{\partial}{\partial v} f(u, v, w) - 2\, w\, \sigma_1 \frac{\partial}{\partial w} f(u, v, w)$$

where

$$\sigma_1 = u^2 + v^2 + w^2$$

```
expand(% - %2)
```

$$0$$

```
delete s:
```

Parameters

`f`

An arithmetical expression in the variables x_1, x_2 etc.

x_1, x_2, …

identifiers or indexed identifiers

`ogCoord`

The name of a three-dimensional orthogonal coordinate system predefined in the table `linalg::ogCoordTab`, or a list of algebraic expressions representing the scale factors of an orthogonal coordinate system.

`c`

The parameter of the coordinate systems EllipticCylindrical and Torus, respectively: an arithmetical expression. The default value is `c` = 1.

Return Values

Arithmetical expression.

Algorithms

Orthogonal coordinates $\vec{u}$ on $\mathbb{R}^n$ are defined by a transformation $\vec{x}(\vec{u})$ to Cartesian coordinates $\vec{x}$ on $\mathbb{R}^n$. The metric tensor associated with the coordinates $\vec{u}$ is given by

$$(g_{i,j}) = \left(\frac{\partial}{\partial u_i}\vec{x}, \frac{\partial}{\partial u_j}\vec{x}\right) = \mathrm{diag}\left({s_1}^2, \ldots, {s_n}^2\right), s_i = \left|\frac{\partial}{\partial u_i}\vec{x}\right|.$$

The Laplacian of a function *f* is given by the divergence

$$\Delta f = \mathrm{div}\left(\vec{F}\right) = \sum_{j=1}^{n}\left(\frac{\partial}{\partial u_j}\frac{F_j}{s_j} + \frac{F_j}{s_j}\left(\sum_{k=1}^{n}\frac{1}{s_k}\frac{\partial}{\partial u_j}s_k\right)\right),$$

where $F_j = \frac{1}{s_j}\frac{\partial}{\partial u_j}f$ are the components of the gradient $\vec{F} = \mathrm{grad}(f)$.

See Also

MuPAD Functions

`curl` | `divergence` | `gradient` | `linalg::ogCoordTab` | `potential` | `vectorPotential`

%, last

Access a previously computed object

Syntax

```
%
```

```
% n
```

```
last(n)
```

Description

`last()` or `%` returns the result of the last command.

`last(n)` or `%n` returns the result of the `n`th previous command.

By default, MuPAD stores the last 20 commands and their results in an internal history table. `last(n)` returns the result entry of the `n`th element in this table, counted from the end of the table. Thus `last(1)` returns the result of the last command, `last(2)` returns the result of the next to last one, etc. Instead of `last(n)` one can also write more briefly `%n`. Instead of `last(1)` or `%1`, one can use even more briefly `%`.

The environment variable `HISTORY` determines the number of previous results that can be accessed at interactive level, i.e., the number of entries in the history table. In procedures, the length of this table is always 3, independent of the value of `HISTORY`. Thus admissible values for `n` are the integers between `1` and `HISTORY` at interactive level, and the integers `1, 2, 3` inside a procedure.

Use `history` to access entries of the history table at interactive level directly, including the command that produced the corresponding result.

The result returned by `last` or `%` is not evaluated again. Use the function `eval` to force a subsequent evaluation. See “Example 4” on page 1-1197.

Note: `last` behaves differently at interactive level and in procedures. At interactive level, compound statements, such as `for`, `repeat`, and `while` loops and `if` and `case`

branching instructions, are stored in the history table as a whole. In procedures, the statements within a compound statement are stored in a separate history table of this procedure, but not the compound statement itself. See “Example 5” on page 1-1198.

Commands and their results are stored in the history table even if the output is suppressed by a colon. Thus the result of `last(n)` may differ from the `n`th previous output that is visible on the screen at interactive level. See “Example 1” on page 1-1195.

Commands appearing on the same input line lead to separate entries in the history table if they are separated by a colon or a semicolon. In contrast, an expression sequence is regarded as a single command. See “Example 2” on page 1-1196.

Commands that are read from a file via `fread` or `read` are stored in the history table *before* the `fread` or `read` command itself. If the option `Plain` is used, then a separate history table is valid within the file, and the commands from the file do not appear in the history table of the enclosing context. See the help page of `history` for examples.

Using `last` in procedures is generally considered bad programming style and is therefore deprecated. Future MuPAD releases may no longer support the use of `last` within procedures.

If the abbreviated syntax `%n` is used, then `n` must be a positive integer literally. If this is not the case, but `n` evaluates to a positive integer, use the equivalent functional notation `last(n)` (see “Example 3” on page 1-1197).

Examples

Example 1

Here are some examples for using `last` at interactive level. Note that `last(n)` refers to the `n`th previously computed result, whether it was displayed or not:

```
a := 42;
last(1), %, %1
```

$$42$$

$$42,\ 42,\ 42$$

```
a := 34: b := 56: last(2) = %2
```

```
34 = 34
```

Example 2

Commands appearing on one input line lead to separate entries in the history table:

```
"First command"; 11: 22; 33:
```

```
"First command"
```

```
22
```

```
last(1), last(2);
```

```
33, 22
```

If a sequence of commands is bracketed, it is regarded as a single command:

```
"First command"; (11: 22; 33:)
```

```
"First command"
```

```
33
```

```
last(1), last(2);
```

```
33, "First command"
```

An expression sequence is also regarded as a single command:

```
"First command"; 11, 22, 33;
```

```
"First command"
```

```
     11, 22, 33
```

```
last(1), last(2);
```

```
     11, 22, 33, "First command"
```

Example 3

Due to the fact that the MuPAD parser expects a number after the `%` sign, there is a difference between the use of `%` and `last`. `last` can be called with an expression that evaluates to a positive integer:

```
n := 2: a := 35: b := 56: last(n)
```

```
     35
```

If you try the same with `%`, an error occurs:

```
n := 2: a := 35: b := 56: %n
```

```
Error: Unexpected 'identifier'. [line 1, col 28]
```

Example 4

The result of `last` is not evaluated again:

```
delete a, b:
c := a + b + a: a:= b: %2
```

```
     2 a + b
```

Use `eval` to enforce the evaluation:

```
eval(%)
```

```
     3 b
```

Example 5

We demonstrate the difference between the use of `last` at interactive level and in procedures:

```
1: for i from 1 to 3 do i: print(%): end_for:
```

```
    1

    1

    1
```

Here `last(1)` refers to the most recent entry in the history table, which is the `1` executed before the `for` loop. We can also verify this by inspecting the history table after these commands. The command `history` returns a list with two elements. The first entry is a previously entered MuPAD command, and the second entry is the result of this command returned by MuPAD. You see that the history table contains the whole `for` loop as a single command:

```
history(history() - 1), history(history())
```

```
[1, 1], [(for i from 1 to 3 do
  i;
  print(%)
end_for), null()]
```

However, if the `for` loop defined above is executed inside a procedure, then we obtain a different result. In the following example, `last(1)` refers to the last evaluated expression, namely the `i` inside the loop:

```
f := proc()
begin
  1: for i from 1 to 3 do i: print(last(1)): end_for
end_proc:

f():
```

```
    1
```

$$2$$

$$3$$

The command `history` refers only to the interactive inputs and their results:

```
history(history())
```

$$[f(), \text{null}()]$$

Parameters

n

A positive integer

Return Values

MuPAD object.

See Also

MuPAD Functions

`HISTORY` | `history`

More About

- “History Mechanism”

lasterror

Reproduce the last error

Syntax

```
lasterror()
```

Description

`lasterror()` reproduces the last error that occurred in the current MuPAD session.

Typically, `lasterror` is used to reproduce errors that were caught by `traperror`. Cf. “Example 2” on page 1-1201.

Examples

Example 1

We produce an error:

```
x := 0: y := 1/x
```

```
Error: Division by zero. [_invert]
```

This error may be reproduced by `lasterror`:

```
lasterror()
```

```
Error: Division by zero. [_invert]
```

A further error is produced:

```
error("my error")
```

```
Error: my error
```

```
lasterror()
```

```
Error: my error
```

```
delete x, y:
```

Example 2

The following procedure `myln` computes the ln function of its argument. In case of an error produced by the system function `ln`, it prints information on the argument and reproduces the error:

```
myln := proc(x)
  local result;
begin
  if traperror((result := ln(x))) = 0 then
     return(result)
  else
     print(Unquoted, "the following error occurred " .
                     "when calling ln(".expr2text(x)."):");
     lasterror()
  end_if:
end:
```

Indeed, the ln has a singularity at 0 and produces:

```
myln(0)
```

```
the following error occurred when calling ln(0):
```

```
Error: Singularity. [ln]
  Evaluating: myln
```

```
delete myln:
```

See Also

MuPAD Functions

`error` | `getlasterror` | `traperror`

_lazy_and

"lazy and" of Boolean expressions

Syntax

```
_lazy_and(b1, b2, …)
```

Description

`_lazy_and(b1, b2, ...)` evaluates the Boolean expression `b1 and b2 and ...` by "lazy evaluation".

`_lazy_and(b1, b2, ...)` produces the same result as `bool(b1 and b2 and ...)`, provided the latter call does not produce an error. The difference between these calls is as follows:

`bool(b1 and b2 and ...)` evaluates *all* Boolean expressions before combining them logically via 'and'.

Note that the result is `FALSE` if one of `b1`, `b2` etc. evaluates to `FALSE`. "Lazy evaluation" is based on this fact: `_lazy_and(b1, b2, ...)` evaluates the arguments from left to right. The evaluation is stopped immediately if one argument evaluates to `FALSE`. In this case, `_lazy_and` returns `FALSE`*without* evaluating the remaining Boolean expressions. If none of the expressions `b1`, `b2` etc. evaluates to`FALSE`, then all arguments are evaluated and the corresponding result `TRUE` or `UNKNOWN` is returned.

`_lazy_and` is also called "conditional `and`".

If any of the considered Boolean expressions `b1`, `b2` etc. cannot be evaluated to `TRUE`, `FALSE`, or `UNKNOWN`, then `_lazy_and` produces errors.

`_lazy_and` is used internally by the `if`, `repeat`, and `while` statements. For example, the statement '`if b1 and b2 then ...`' is equivalent to '`if _lazy_and(b1, b2) then ...`'.

`_lazy_and()` returns `TRUE`.

Examples

Example 1

This example demonstrates the difference between lazy evaluation and complete evaluation of Boolean conditions. For $x = 0$, the evaluation of $\sin\left(\frac{1}{x}\right)$ leads to an error:

```
x := 0: bool(x <> 0 and sin(1/x) = 0)
```

```
Error: Division by zero. [_invert]
```

With “lazy evaluation”, the expression $\sin\left(\frac{1}{x}\right) = 0$ is not evaluated. This avoids the previous error:

```
_lazy_and(x <> 0, sin(1/x) = 0)
```

```
    FALSE
```

```
bool(x = 0 or sin(1/x) = 0)
```

```
Error: Division by zero. [_invert]
```

```
_lazy_or(x = 0, sin(1/x) = 0)
```

```
    TRUE
```

```
delete x:
```

Example 2

The following statements do no produce an error, because `if` uses lazy evaluation internally:

```
for x in [0, PI, 1/PI] do
  if x = 0 or sin(1/x) = 0 then
     print(x)
  end_if;
```

```
end_for:
```

$$0$$

$$\frac{1}{\pi}$$

```
delete x:
```

Example 3

Both functions can be called without parameters:

```
_lazy_and(), _lazy_or()
```

$$\text{TRUE, FALSE}$$

Parameters

`b1, b2, …`

Boolean expressions

Return Values

TRUE, FALSE, or UNKNOWN.

Overloaded By

b1, b2

See Also

MuPAD Functions

`_lazy_or` | `and` | `bool` | `FALSE` | `if` | `is` | `or` | `repeat` | `TRUE` | `UNKNOWN` | `while`

_lazy_or

"lazy or" of Boolean expressions

Syntax

```
_lazy_or(b1, b2, …)
```

Description

`_lazy_or(b1, b2, ...)` evaluates the Boolean expression `b1 or b2 or ...` by "lazy evaluation".

`_lazy_or(b1, b2, ...)` produces the same result as `bool(b1 or b2 or ...)`, provided the latter call does not produce an error. The difference between these calls is as follows:

`bool(b1 or b2 or ...)` evaluates *all* Boolean expressions before combining them logically via 'or'.

Note that the result is `TRUE` if one of `b1`, `b2` etc. evaluates to `TRUE`. "Lazy evaluation" is based on this fact: `_lazy_or(b1, b2, ...)` evaluates the arguments from left to right. The evaluation is stopped immediately if one argument evaluates to `TRUE`. In this case, `_lazy_or` returns `TRUE`*without* evaluating the remaining Boolean expressions. If none of the expressions `b1`, `b2` etc. evaluates to `TRUE`, then all arguments are evaluated and the corresponding result `FALSE` or `UNKNOWN` is returned.

`_lazy_or` is also called "conditional `or`".

If any of the considered Boolean expressions `b1`, `b2` etc. cannot be evaluated to `TRUE`, `FALSE`, or `UNKNOWN`, then `_lazy_or` produces errors.

`_lazy_or` is used internally by the `if`, `repeat`, and `while` statements.

`_lazy_or()` returns `FALSE`.

Examples

Example 1

This example demonstrates the difference between lazy evaluation and complete evaluation of Boolean conditions. For $x = 0$, the evaluation of $\sin\left(\frac{1}{x}\right)$ leads to an error:

```
x := 0: bool(x <> 0 and sin(1/x) = 0)
```

```
Error: Division by zero. [_invert]
```

With “lazy evaluation”, the expression $\sin\left(\frac{1}{x}\right) = 0$ is not evaluated. This avoids the previous error:

```
_lazy_and(x <> 0, sin(1/x) = 0)
```

```
    FALSE
```

```
bool(x = 0 or sin(1/x) = 0)
```

```
Error: Division by zero. [_invert]
```

```
_lazy_or(x = 0, sin(1/x) = 0)
```

```
    TRUE
```

```
delete x:
```

Example 2

The following statements do no produce an error, because `if` uses lazy evaluation internally:

```
for x in [0, PI, 1/PI] do
  if x = 0 or sin(1/x) = 0 then
     print(x)
  end_if;
```

```
end_for:
```

$$0$$

$$\frac{1}{\pi}$$

```
delete x:
```

Example 3

Both functions can be called without parameters:

```
_lazy_and(), _lazy_or()
```

$$\text{TRUE, FALSE}$$

Parameters

`b1, b2, …`

Boolean expressions

Return Values

TRUE, FALSE, or UNKNOWN.

Overloaded By

b1, b2

See Also

MuPAD Functions

`_lazy_and` | `and` | `bool` | `FALSE` | `if` | `is` | `or` | `repeat` | `TRUE` | `UNKNOWN` | `while`

lcm

Least common multiple of polynomials

Syntax

```
lcm(p, q, …)
```

```
lcm(f, g, …)
```

Description

`lcm(p, q, ...)` calculates the least common multiple of any number of polynomials. The coefficient ring of the polynomials may either be the integers or the rational numbers, `Expr`, a residue class ring `IntMod(n)` with a prime number `n`, or a domain.

All polynomials must have the same indeterminates and the same coefficient ring.

Polynomial expressions are converted to polynomials. See `poly` for details. `FAIL` is returned if an argument cannot be converted to a polynomial.

The return value is of the same type as the input polynomials, i.e., either a polynomial of type `DOM_POLY` or a polynomial expression.

`lcm` returns 1 if all arguments are 1 or - 1, or if no argument is given. If at least one of the arguments is 0, then `lcm` returns 0.

Use `ilcm` if all arguments are known to be integers, since it is much faster than `lcm`.

Examples

Example 1

The least common multiple of two polynomial expressions can be computed as follows:

```
lcm(x^3 - y^3, x^2 - y^2);
```

$$(x+y)\,(x^3-y^3)$$

One may also choose polynomials as arguments:

```
p := poly(x^2 - y^2, [x, y], IntMod(17)):
q := poly(x^2 - 2*x*y + y^2, [x, y], IntMod(17)):
lcm(p, q)
```

$$\mathrm{poly}(x^3-x^2\,y-x\,y^2+y^3,\,[x,\,y],\,\mathrm{IntMod}(17))$$

```
delete f, g, p, q:
```

Parameters

`pq, …`

polynomials of type `DOM_POLY`

`fg, …`

polynomial expressions

Return Values

Polynomial, a polynomial expression, or the value `FAIL`.

Overloaded By

f, g, p, q

See Also

MuPAD Functions

`content` | `factor` | `gcd` | `gcdex` | `icontent` | `ifactor` | `igcd` | `igcdex` | `ilcm` | `poly`

lcoeff

Leading coefficient of a polynomial

Syntax

```
lcoeff(p, <order>)
```

```
lcoeff(f, <vars>, <order>)
```

Description

`lcoeff(p)` returns the leading coefficient of the polynomial `p`.

The returned coefficient is "leading" with respect to the lexicographical ordering, unless a different ordering is specified via the argument `order`. Cf. "Example 1" on page 1-1210.

A polynomial expression `f` is first converted to a polynomial with the variables given by `vars`. If no variables are given, they are searched for in `f`. See `poly` about details of the conversion. The result is returned as polynomial expression. `FAIL` is returned if `f` cannot be converted to a polynomial. Cf. "Example 3" on page 1-1211.

The result of `lcoeff` is not fully evaluated. Evaluation can be enforced by the function `eval`. Cf. "Example 2" on page 1-1211.

Examples

Example 1

We demonstrate how various orderings influence the result:

```
p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x, y, z]):
lcoeff(p), lcoeff(p, DegreeOrder), lcoeff(p, DegInvLexOrder)
```

```
5, 4, 3
```

The following call uses the reverse lexicographical order on 3 indeterminates:

```
lcoeff(p, Dom::MonomOrdering(RevLex(3)))
```

$$3$$

```
delete p:
```

Example 2

The result of `lcoeff` is not fully evaluated:

```
p := poly(a*x^2 + 27*x, [x]): a := 5:
lcoeff(p), eval(lcoeff(p))
```

$$a, 5$$

```
delete p, a:
```

Example 3

The expression `1/x` may not be regarded as polynomial:

```
lcoeff(1/x)
```

$$\text{FAIL}$$

Parameters

`p`

A polynomial of type `DOM_POLY`

`f`

A polynomial expression

vars

A list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

order

The term ordering: either `LexOrder`, or `DegreeOrder`, or `DegInvLexOrder`, or a user-defined term ordering of type `Dom::MonomOrdering`. The default is the lexicographical ordering `LexOrder`.

Return Values

Element of the coefficient domain of the polynomial or `FAIL`.

Overloaded By

p

See Also

MuPAD Functions

`coeff` | `collect` | `degree` | `degreevec` | `ground` | `ldegree` | `lmonomial` | `lterm` | `monomials` | `nterms` | `nthcoeff` | `nthmonomial` | `nthterm` | `poly` | `poly2list` | `tcoeff`

ldegree

Lowest degree of the terms in a polynomial

Syntax

```
ldegree(p)

ldegree(p, x)

ldegree(f, <vars>)

ldegree(f, <vars>, x)
```

Description

`ldegree(p)` returns the lowest total degree of the terms of the polynomial `p`.

`ldegree(p, x)` returns the lowest degree of the terms in `p` with respect to the variable `x`.

If the first argument `f` is not element of a polynomial domain, then `ldegree` converts the expression to a polynomial via `poly(f)`. If a list of indeterminates is specified, then the polynomial `poly(f, vars)` is considered.

`ldegree(f, vars, x)` returns 0 if `x` is not an element of `vars`.

The low degree of the zero polynomial is defined as 0.

Examples

Example 1

The lowest total degree of the terms in the following polynomial is computed:

```
ldegree(x^3 + x^2*y^2)
```

```
3
```

The next call regards the expression as a polynomial in `x` with a parameter `y`:

```
ldegree(x^3 + x^2*y^2, x)
```

```
2
```

The next expression is regarded as a bi-variate polynomial in `x` and `z` with coefficients containing the parameter `y`. The total degree with respect to `x` and `z` is computed:

```
ldegree(x^3*z^2 + x^2*y^2*z, [x, z])
```

```
3
```

We compute the low degree with respect to `x`:

```
ldegree(x^3*z^2 + x^2*y^2*z, [x, z], x)
```

```
2
```

A polynomial in `x` and `z` is regarded constant with respect to any other variable, i.e., its corresponding degree is 0:

```
ldegree(poly(x^3*z^2 + x^2*y^2*z, [x, z]), y)
```

```
0
```

Parameters

`p`

A polynomial of type `DOM_POLY`

`f`

A polynomial expression

vars

A list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

x

An indeterminate

Return Values

Nonnegative number. FAIL is returned if the input cannot be converted to a polynomial.

Overloaded By

f, p

See Also

MuPAD Functions

coeff | degree | degreevec | ground | lcoeff | lmonomial | lterm | monomials | nterms | nthcoeff | nthmonomial | nthterm | poly | poly2list | tcoeff

length

The “length” of a object (heuristic complexity)

Syntax

```
length(object)
```

Description

`length(object)` returns an integer indicating the complexity of the object.

The (heuristic) complexity of an object may be useful in algorithms that need to predict the complexity and time for manipulating objects. E.g., a symbolic Gaussian algorithm for solving linear equations prefers Pivot elements of small complexity.

The length of an object is determined as follows:

- Objects of domain type `DOM_BOOL`, `DOM_DOMAIN`, `DOM_EXEC`, `DOM_FAIL`, `DOM_FLOAT`, `DOM_FUNC_ENV`, `DOM_IDENT`, `DOM_NIL`, `DOM_VAR`, and `DOM_PROC_ENV` are regarded as “atomic”. They have length 1. In particular, the length of identifiers and real floating-point numbers is 1.
- The length of an integer is (a close approximation of) the number of decimal digits, including the sign.
- The length of a string is the number of its characters.
- The length of composite objects such as complex numbers, rational numbers, arithmetical expressions, lists, sets, arrays, hfarrays, tables etc. is the sum of the lengths of the operands plus 1.

`length()` yields 0.

Note: `length` does *not* return the number of elements or entries in sets, lists or tables. Use `nops` instead!

Examples

Example 1

Intuitively, the length measures the complexity of an object:

```
length(1 + x) < length(x^3 + exp(a - b)/ln(45 - t) - 1234*I)
```

```
3 < 30
```

Example 2

We compute the lengths of some simple objects:

```
length(1.2), length(-1234.5), length(123456), length(-123456)
```

```
1, 1, 6, 7
```

```
length(17), length(123), length(17/123)
```

```
2, 3, 6
```

```
length(12), length(123), length(12 + 123*I)
```

```
2, 3, 6
```

```
length(x), length(x^2), length(x^12345)
```

```
1, 3, 7
```

```
length("123"), length("")
```

```
3, 0
```

```
length(x), length(a_long_name)
```

```
1, 1
```

Example 3

The length of an array is the sum of the lengths of all its elements plus 1:

```
A := array(1..2, [x, y]): length(A) = length(x) + length(y) + 1
```

```
3 = 3
```

```
A[1] := 12345: length(A) = length(12345) + length(y) + 1
```

```
7 = 7
```

```
A := hfarray(1..10, [1.0 $ 10]):
length(A) = 10*length(1.0) + 1
```

```
11 = 11
```

```
A := hfarray(1..10, [1.0 + 2.0*I $ 10]):
length(A) = 10*length(1.0 + 2.0*I) + 1
```

```
31 = 31
```

Beware: If only one complex number is contained in an hfarray, then *all* entries are regarded as complex numbers, even if they are real:

```
A := hfarray(1..10, [2.0 $ 9, 2.0 + 3.0*I]):
length(A) = 10*length(2.0 + 3.0*I) + 1
```

```
31 = 31
```

```
delete A:
```

Example 4

The operands of a table are the equations associating indices and entries. The length of each operand is the length of the index plus the length of the corresponding entry plus 1:

```
T[1] := 45: T
```

```
length(T) = length(1 = 45) + 1
```

```
delete T:
```

Parameters

`object`

An arbitrary MuPAD object

Return Values

Nonnegative integer.

See Also

MuPAD Functions
nops | op

LEVEL

Substitution depth of identifiers

Description

The environment variable LEVEL determines the maximal substitution depth of identifiers.

Possible values: a positive integer smaller than 2^{31}.

When a MuPAD object is evaluated, identifiers occurring in it are replaced by their values. This happens recursively, i.e., if the values themselves contain identifiers, then these are replaced as well. LEVEL determines the maximal recursion depth of this process.

Technically, evaluation of a MuPAD object works as follows. For a compound object, usually first the operands are evaluated recursively, and then the object itself is evaluated. E.g., if the object is a function call with arguments,the arguments are evaluated first, and then the function is executed with the evaluated arguments.

With respect to the evaluation of identifiers, the *current substitution depth* is recorded internally. Initially, this value is zero. If an identifier is encountered during the recursive evaluation process as described above and the current substitution depth is smaller than LEVEL, then the identifier is replaced by its value, the current substitution depth is increased by one, and evaluation proceeds recursively with the value of the identifier. After the identifier has been evaluated, the current substitution depth is reset to its previous value. If the current substitution depth equals LEVEL, however, then the recursion stops and the identifier remains unevaluated.

Note: The default value of LEVEL at interactive level is 100. However, the default value of LEVEL within a procedure is 1. Then an identifier is only replaced by its value, which is not evaluated recursively.

The value of LEVEL may be changed within a procedure, but it is reset to 1 each time a new procedure is entered. After the procedure returns, LEVEL is reset to its previous value. See “Example 3” on page 1-1223.

Note: The evaluation of local variables and formal parameters of procedures, of type DOM_VAR, is not affected by LEVEL: they are always evaluated with substitution depth 1. This means that a local variable or a formal parameter is replaced by its value when evaluated, but the value is not evaluated further.

See "Example 3" on page 1-1223.

Note: LEVEL does not affect the evaluation of arrays, tables and polynomials.

See "Example 4" on page 1-1224.

The function eval evaluates its argument with substitution depth given by LEVEL, and then evaluates the result again with the same substitution depth.

The call level(object, n) evaluates its argument with substitution depth n, independent of the value of LEVEL.

If, during evaluation, the substitution depth MAXLEVEL, is reached, then the evaluation is terminated with an error. This is a heuristic for recognizing recursive definitions, as in the example delete a; a := a + 1; a. Here, a would be replaced by a + 1 infinitely often. Note that this has no effect if MAXLEVEL is greater than LEVEL. The default value of MAXLEVEL is 100, i.e., it is equal to the default value of LEVEL at interactive level. However, unlike LEVEL, MAXLEVEL is not changed within a procedure, and hence recursive definitions are usually not recognized within procedures. See the help page of MAXLEVEL for examples.

The default value of LEVEL is 100 at interactive level; LEVEL has this value after starting or resetting the system via reset. Within a procedure, the default value is 1. The command delete LEVEL restores the default value.

Examples

Example 1

We demonstrate the effect of various values of LEVEL at interactive level:

```
delete a0, a1, a2, a3, a4, b: b := b + 1:
```

```
a0 := a1: a1 := a2 + 2: a2 := a3 + a4: a3 := a4^2: a4 := 5:

LEVEL := 1: a0, a0 + a2, b;
LEVEL := 2: a0, a0 + a2, b;
LEVEL := 3: a0, a0 + a2, b;
LEVEL := 4: a0, a0 + a2, b;
LEVEL := 5: a0, a0 + a2, b;
LEVEL := 6: a0, a0 + a2, b;
delete LEVEL:
```

$$a1,\ a1 + a3 + a4,\ b + 1$$

$$a2 + 2,\ a4^2 + a2 + 7,\ b + 2$$

$$a3 + a4 + 2,\ a3 + a4 + 32,\ b + 3$$

$$a4^2 + 7,\ a4^2 + 37,\ b + 4$$

$$32,\ 62,\ b + 5$$

$$32,\ 62,\ b + 6$$

Example 2

In the following calls, the identifier `a` is fully evaluated:

```
delete a, b, c:
a := b: b := c: c := 7: a
```

$$7$$

After assigning the value `2` to `LEVEL`, `a` is evaluated only with depth two:

```
LEVEL := 2: a;
delete LEVEL:
```

$$c$$

If we set MAXLEVEL to 2 as well, evaluation of a produces an error, although there is no recursive definition involved:

```
LEVEL := 2: MAXLEVEL := 2: a
```

```
Error: Recursive definition, the maximal evaluation level is reached.
```

```
delete LEVEL, MAXLEVEL:
```

Example 3

This example shows the difference between the evaluation of identifiers and local variables. By default, the value of LEVEL is 1 within a procedure, i.e., a global identifier is replaced by its value when evaluated, but there is no further recursive evaluation. This changes when LEVEL is assigned a bigger value inside the procedure:

```
delete a0, a1, a2, a3:
a0 := a1 + a2:  a1 := a2 + a3:  a2 := a3^2 - 1:  a3 := 5:
p := proc()
       save LEVEL;
     begin
       print(a0, eval(a0)):
       LEVEL := 2:
       print(a0, eval(a0)):
     end_proc:

p()
```

$$a1 + a2,\ a3^2 + a3 + a2 - 1$$

$$a3^2 + a3 + a2 - 1,\ 53$$

In contrast, evaluation of a local variable replaces it by its value, without further evaluation. When eval is applied to an object containing a local variable, then the effect is an evaluation of the value of the local variable with substitution depth LEVEL:

```
q := proc()
       save LEVEL;
       local x;
     begin
```

```
      x := a0:
      print(x, eval(x)):
      LEVEL := 2:
      print(x, eval(x)):
    end_proc:
q()
```

$$a1 + a2,\ a3^2 + a3 + a2 - 1$$

$$a1 + a2,\ a3^2 + 28$$

The command `x:=a0` assigns the value of the identifier `a0`, namely the unevaluated expression `a1+a2`, to the local variable `x`, and `x` is replaced by this value every time it is evaluated, independent of the value of `LEVEL`.

Example 4

`LEVEL` does not affect on evaluation of `polynomials`:

```
delete a, x:  p := poly(a*x, [x]):  a := 2:  x := 3:
p, eval(p);
LEVEL := 1: p, eval(p);
delete LEVEL:
```

$$\mathrm{poly}(a\,x, [x]),\ \mathrm{poly}(a\,x, [x])$$

$$\mathrm{poly}(a\,x, [x]),\ \mathrm{poly}(a\,x, [x])$$

The same is true for `arrays` and `tables`:

```
delete a, b:
A := array(1..2, [a, b]):  T := table(a = b):
a := 1:   b := 2:
A, eval(A), T, eval(T);
LEVEL := 1: A, eval(A), T, eval(T);
delete LEVEL:
```

$$(a\ b),\ (a\ b),\ \overline{a|b},\ \overline{a|b}$$

$$(a\ b), (a\ b), \overline{a|b}, \overline{a|b}$$

See Also

MuPAD Functions

`context` | `eval` | `hold` | `level` | `MAXDEPTH` | `MAXLEVEL` | `val`

More About

- "Level of Evaluation"

level

Evaluate an object with a specified substitution depth

Syntax

```
level(object)
```

```
level(object, n)
```

Description

`level(object, n)` evaluates `object` with substitution depth `n`.

When a MuPAD object is evaluated, identifiers occurring in it are replaced by their values. This happens recursively, i.e., if the values themselves contain identifiers, then these are replaced as well. `level` serves to evaluate an object with a specified recursion depth for this substitution process.

With `level(object, 0)`, `object` is evaluated without replacing any identifier occurring in it by its value. In most cases, but not always, this equivalent to `hold(object)`, and `object` is returned unevaluated. See “Example 3” on page 1-1230.

With `level(object, 1)`, all identifiers occurring in `object` are replaced by their values, but not recursively, and then all function calls in the result of the substitution are executed. This is how objects are evaluated within a procedure by default.

The call `level(object)` is equivalent to `level(object, MAXLEVEL)`, i.e., identifiers occurring in `object` are recursively replaced by their values up to substitution depth `MAXLEVEL - 1`, and an error occurs if the substitution depth `MAXLEVEL` is reached. Usually, this leads to a complete evaluation of `object`. See “Example 1” on page 1-1228.

You can use `level` without a second argument to request the complete evaluation of an object not containing local variables or formal parameters within a procedure. This may be necessary since by default, objects are evaluated with substitution depth `1` within procedures. See “Example 2” on page 1-1229.

Otherwise, it should never be necessary to use `level`.

Note: `level` does not affect the evaluation of local variables and formal parameters, of type `DOM_VAR`, in procedures. When such a local variable occurs in `object`, then it is always replaced by its value, independent of the value of `n`, and the value is not further recursively evaluated. See "Example 2" on page 1-1229.

`level` works by temporarily setting the value of `LEVEL` to `n`, or to 2^{31} - 1 if `n` is not given. However, the value of `MAXLEVEL` remains unchanged. If the substitution depth `MAXLEVEL` is reached, then an error message is returned. See `LEVEL` and `MAXLEVEL` for more information on these environment variables.

In contrast to most other functions, `level` does not flatten its first argument if it is an expression sequence. See "Example 5" on page 1-1231.

`level` does not recursively descend into `arrays`, `tables`, `matrices` or `polynomials`. Use the call `map(object, eval)` to evaluate the entries of an array, a table, a matrix or `mapcoeffs(object, eval)` to evaluate the coefficients of a polynomial. See "Example 4" on page 1-1231 and "Example 6" on page 1-1231.

Further information concerning the evaluation of arrays, tables, matrices or polynomials can be found on the `eval` help page.

The maximal substitution depth of `level` depends on the environment variable `MAXLEVEL`, while the maximum evaluation depth of the function `eval` depends on the environment variable `LEVEL`. See "Example 7" on page 1-1232.

Because `eval` evaluates the result again there is a difference between evaluating an expression with depth *n* by `level` in comparison with `eval`. See "Example 7" on page 1-1232.

As mentioned `level` does not affect the evaluation of local variables and formal parameters, of type `DOM_VAR`, in procedures. Here `eval` behaves different. See "Example 7" on page 1-1232 and the `eval` help page for more information.

The result of `level(hold(x))` is always `x`, because a full evaluation of `hold(x)` leads to `x`. The same does not hold for `eval(hold(x))`, because `eval` first evaluates its argument and then evaluates the result again.

The evaluation of elements of a user-defined domain depends on the implementation of the domain. Usually domain elements remain unevaluated by `level`. If the domain has a slot `"evaluate"`, the corresponding slot routine is called with the domain element

as argument at each evaluation, and hence it is called once when `level` is invoked. Cf. “Example 8” on page 1-1233.

Examples

Example 1

We demonstrate the effect of `level` for various values of the second parameter:

```
delete a0, a1, a2, a3, a4, b: b := b + 1:
a0 := a1: a1 := a2 + 2: a2 := a3 + a4: a3 := a4^2: a4 := 5:

hold(a0), hold(a0 + a2), hold(b);
level(a0, 0), level(a0 + a2, 0), level(b, 0);
level(a0, 1), level(a0 + a2, 1), level(b, 1);
level(a0, 2), level(a0 + a2, 2), level(b, 2);
level(a0, 3), level(a0 + a2, 3), level(b, 3);
level(a0, 4), level(a0 + a2, 4), level(b, 4);
level(a0, 5), level(a0 + a2, 5), level(b, 5);
level(a0, 6), level(a0 + a2, 6), level(b, 6);
```

$$a0,\ a0 + a2,\ b$$

$$a0,\ a0 + a2,\ b$$

$$a1,\ a1 + a3 + a4,\ b + 1$$

$$a2 + 2,\ a4^2 + a2 + 7,\ b + 2$$

$$a3 + a4 + 2,\ a3 + a4 + 32,\ b + 3$$

$$a4^2 + 7,\ a4^2 + 37,\ b + 4$$

$$32,\ 62,\ b + 5$$

$$32, 62, b + 6$$

Evaluating `object` by just typing `object` at the command prompt is equivalent to `level(object, LEVEL)`:

```
LEVEL := 2: MAXLEVEL := 4: a0, a2, b;
level(a0, LEVEL), level(a2, LEVEL), level(b, LEVEL)
```

$$a2 + 2, a4^2 + 5, b + 2$$

$$a2 + 2, a4^2 + 5, b + 2$$

If the second argument is omitted, then this corresponds to a complete evaluation up to substitution depth `MAXLEVEL - 1`:

```
level(a0)
```

```
Error: Recursive definition, the maximal evaluation level is reached.
```

```
level(a2)
```

$$30$$

```
level(b)
```

```
Error: Recursive definition, the maximal evaluation level is reached.
```

```
delete LEVEL, MAXLEVEL:
```

Example 2

We demonstrate the behavior of `level` in procedures:

```
delete a, b, c: a := b: b := c: c := 42:
p := proc()
  local x;
begin
```

```
  x := a:
  print(level(x, 0), x, level(x, 2), level(x)):
  print(level(a, 0), a, level(a, 2), level(a)):
end_proc:
p()
```

b, b, b, b

$a, b, c, 42$

Since `a` is evaluated with the default substitution depth `1`, the assignment `x:=a` sets the value of the local variable `x` to the unevaluated identifier `b`. You can see that any evaluation of `x`, whether `level` is used or not, simply replaces `x` by its value `b`, but no further recursive evaluation happens. In contrast, evaluation of the identifier `a` takes place with the default substitution depth `1`, and `level(a, 2)` evaluates it with substitution depth `2`.

Thus `level` without a second argument can be used to request the complete evaluation of an object not containing any local variables or formal parameters.

Example 3

There are some rare cases where `level(object, 0)` and `hold(object)` behaves different. This is the case if `object` is not an identifier, e.g., a nameless function, because `level` influences only the evaluation of identifiers:

```
level((x -> x^2)(2),0), hold((x -> x^2)(2))
```

$4, (x \to x^2)(2)$

For the same reason `level(object, 0)` and `hold(object)` behave differently if `object` is a local variable of a procedure:

```
f:=proc() local x; begin
  x := 42;
  hold(x), level(x, 0);
end_proc:
f();
delete f:
```

DOM_VAR(0, 2), 42

Example 4

In contrast to lists and sets, evaluation of an array does not evaluate its entries. Thus `level` has no effect for arrays either. The same holds for tables and matrices. Use `map` to evaluate all entries of an array. On the `eval` help page further examples can be found:

```
delete a, b:
L := [a, b]:  A := array(1..2, L):  a := 1:   b := 2:
L, A, level(A), map(A, level), map(A, eval)
```

$[1, 2], (a\ b), (a\ b), (a\ b), (1\ 2)$

Example 5

The first argument of `level` may be an expression sequence, which is not flattened. However, it must be enclosed in parentheses:

```
delete a, b: a := b: b := 3:
level((a, b), 1);
level(a, b, 1)
```

$b, 3$

```
Error: The number of arguments is incorrect. [level]
```

Example 6

`Polynomials` are inert when evaluated, and so `level` has no effect:

```
delete a, x: p := poly(a*x, [x]): a := 2: x := 3:
p, level(p)
```

$\mathrm{poly}(a\,x, [x]), \mathrm{poly}(a\,x, [x])$

Use `mapcoeffs` and the function `eval` to evaluate all coefficients:

```
mapcoeffs(p, eval)
```

$$\mathrm{poly}(2\,x, [x])$$

If you want to substitute a value for the indeterminate `x`, use `evalp`:

```
delete x: evalp(p, x = 3)
```

$$3\,a$$

As you can see, the result of an `evalp` call may contain unevaluated identifiers, and you can evaluate them by an application of `eval`. It is necessary to use `eval` instead of `level` because `level` does not evaluate its result:

```
eval(evalp(p, x = 3))
```

$$6$$

Example 7

The subtle difference between `level` and `eval` is shown. The evaluation depth of `eval` is limited by the environment variable `LEVEL`. `level` pays no attention to `LEVEL`, but rather continues evaluating its argument either as many times as the second argument implies or until it has been evaluated completely:

```
delete a0, a1, a2, a3:
a0 := a1 + a2: a1 := a2 + a3: a2 := a3^2 - 1: a3 := 5:
LEVEL := 1:
eval(a0), level(a0);
```

$$a3^2 + a3 + a2 - 1,\ 53$$

If the evaluation depth exceeds the value of `MAXLEVEL`, an error is raised in both cases:

```
delete LEVEL:
MAXLEVEL := 3:
level(a0);
```

```
Error: Recursive definition, the maximal evaluation level is reached.
```

```
delete LEVEL:
MAXLEVEL := 3:
eval(a0);
delete MAXLEVEL:
```

```
Error: Recursive definition, the maximal evaluation level is reached.
```

It is not the same evaluating an expression `ex` with `eval` and an evaluation depth *n* and by `level((ex, n))`, because `eval` evaluates its result:

```
LEVEL := 2: eval(a0), level(a0, 2);
delete LEVEL:
```

$$53,\ a3^2 + a3 + a2 - 1$$

`level` does not affect the evaluation of local variables of type `DOM_VAR` while `eval` evaluates them with evaluation depth `LEVEL`, which is one in a procedure:

```
p := proc()
  local x;
begin
  x := a0:
  print(eval(x), level(x)):
end_proc:
p()
```

$$a3^2 + a3 + a2 - 1,\ a1 + a2$$

Example 8

The evaluation of an element of a user-defined domain depends on the implementation of the domain. Usually it is not further evaluated:

```
delete a: T := newDomain("T"):
e := new(T, a): a := 1:
e, level(e), map(e, level), val(e)
```

$$\mathrm{new}(T, a),\ \mathrm{new}(T, a),\ \mathrm{new}(T, a),\ \mathrm{new}(T, a)$$

If the slot `"evaluate"` exists, the corresponding slot routine is called for a domain element each time it is evaluated. We implement the routine `T::evaluate`, which simply evaluates all internal operands of its argument, for our domain `T`. The unevaluated domain element can still be accessed via `val`:

```
T::evaluate := x -> new(T, eval(extop(x))):
e, level(e), map(e, level), val(e);
```

$$\mathrm{new}(T, 1), \mathrm{new}(T, 1), \mathrm{new}(T, 1), \mathrm{new}(T, a)$$

```
delete e, T:
```

Parameters

`object`

Any MuPAD object

`n`

A nonnegative integer less than 2^{31}

Return Values

Evaluated object.

See Also

MuPAD Functions

`context` | `eval` | `hold` | `indexval` | `LEVEL` | `MAXLEVEL` | `val`

More About

- “Level of Evaluation”

lhs

Left hand side of equations, inequalities, relations, intervals, ranges and tables

Syntax

```
lhs(f)
```

Description

`lhs(f)` returns the left hand side of `f`.

The call `lhs(f)` is equivalent to the direct call `op(f, 2)`, of the operand function `op`, if `f` is not a table.

If `t` is a table, the call `lhs(t)` returns the list of keys of the table (left hand side). Note that the i-th value in `rhs(t)` corresponds to the i-th key in `lhs(t)`.

Examples

Example 1

We extract the left and right hand sides of various objects:

```
lhs(x = sin(2)), lhs(3.14 <> PI), lhs(x + 3 < 2*y),
rhs(a <= b), rhs(m-1..n+1)
```

$$x,\ 3.14,\ x+3,\ b,\ n+1$$

The operands of an expression depend on its internal representation. In particular, a "greater" relation is always converted to the corresponding "less" relation:

```
y > -infinity; lhs(y > -infinity)
```

$$-\infty < y$$

$$-\infty$$

```
y >= 4; rhs(y >= 4)
```

$$4 \le y$$

$$y$$

Example 2

We extract the left and right hand sides of the solution of the following system:

```
s := solve({x + y = 1, 2*x - 3*y = 2})
```

$$\{[x = 1, y = 0]\}$$

```
map(op(s), lhs) =  map(op(s), rhs)
```

$$[x, y] = [1, 0]$$

Calls to `lhs` and `rhs` may be easier to read than the equivalent calls to the operand function `op`:

```
map(op(s), op, 1) = map(op(s), op, 2)
```

$$[x, y] = [1, 0]$$

However, direct calls to `op` should be preferred inside procedures for higher efficiency.

```
delete s:
```

Example 3

We extract the keys (left hand side) and values (right hand side) from a table:

```
t := table(1=2, 4=PI, 5=5.6, 19=1/2):
l := lhs(t);
```

$$[1, 4, 5, 19]$$

```
r := rhs(t);
```

$$\left[2, \pi, 5.6, \frac{1}{2}\right]$$

Note that the i-th value corresponds to the i-th key:

```
bool(r = map(lhs(t), e->t[e]))
```

$$\text{TRUE}$$

```
delete t,l,r:
```

Parameters

`f`

An equation `x = y`, an inequality `x <> y`, a relation `x < y`, a relation `x <= y`, an "is element of"-relation `x in y`, an interval `x...y`, a range `x..y` or a table `table(x=y,...)`

Return Values

arithmetical expression.

Overloaded By

f

See Also

MuPAD Functions

`isolate` | `op` | `rhs`

rhs

Right hand side of equations, inequalities, relations, intervals, ranges and tables

Syntax

```
rhs(f)
```

Description

`rhs(f)` returns the right hand side of `f`.

The call `rhs(f)` is equivalent to the direct call `op(f, 2)`, of the operand function `op`, if `f` is not a table.

If `t` is a table, the call `rhs(t)` returns the list of values of the table (right hand side). Note that the i-th value in `rhs(t)` corresponds to the i-th key in `lhs(t)`.

Examples

Example 1

We extract the left and right hand sides of various objects:

```
lhs(x = sin(2)), lhs(3.14 <> PI), lhs(x + 3 < 2*y),
rhs(a <= b), rhs(m-1..n+1)
```

$$x,\ 3.14,\ x+3,\ b,\ n+1$$

The operands of an expression depend on its internal representation. In particular, a “greater” relation is always converted to the corresponding “less” relation:

```
y > -infinity; lhs(y > -infinity)
```

$$-\infty < y$$

$$-\infty$$

```
y >= 4; rhs(y >= 4)
```

$$4 \le y$$

$$y$$

Example 2

We extract the left and right hand sides of the solution of the following system:

```
s := solve({x + y = 1, 2*x - 3*y = 2})
```

$$\{[x = 1, y = 0]\}$$

```
map(op(s), lhs) =  map(op(s), rhs)
```

$$[x, y] = [1, 0]$$

Calls to `lhs` and `rhs` may be easier to read than the equivalent calls to the operand function `op`:

```
map(op(s), op, 1) = map(op(s), op, 2)
```

$$[x, y] = [1, 0]$$

However, direct calls to `op` should be preferred inside procedures for higher efficiency.

```
delete s:
```

Example 3

We extract the keys (left hand side) and values (right hand side) from a table:

```
t := table(1=2, 4=PI, 5=5.6, 19=1/2):
l := lhs(t);
```

$$[1, 4, 5, 19]$$

```
r := rhs(t);
```

$$\left[2, \pi, 5.6, \frac{1}{2}\right]$$

Note that the i-th value corresponds to the i-th key:

```
bool(r = map(lhs(t), e->t[e]))
```

$$\text{TRUE}$$

```
delete t,l,r:
```

Parameters

`f`

An equation `x = y`, an inequality `x <> y`, a relation `x < y`, a relation `x <= y`, an "is element of"-relation `x in y`, an interval `x...y`, a range `x..y` or a table `table(x=y,...)`

Return Values

arithmetical expression.

Overloaded By

f

See Also

MuPAD Functions

`isolate` | `lhs` | `op`

Li

Integral logarithm

Syntax

```
Li(x)
```

Description

`Li(x)` represents the integral logarithm $\int_0^x \frac{1}{\ln(t)}\,dt$.

Note that in some places in the literature, the notation `li` is used while `Li` is reserved for the offset logarithmic integral $\int_2^x \frac{1}{\ln(t)}\,dt$. The latter may be obtained by entering `Li(x) - Li(2)`.

Further, do not confuse the integral logarithm `Li` with the polylogarithms `polylog` which are displayed on the screen as Li_n (with an index).

If `x` is a floating-point number, then `Li(x)` returns the numerical value of the integral logarithm. The special values $Li(0) = 1$ and $Li(1) = -\infty$ are implemented. For all other arguments, `Li` returns a symbolic function call.

For all complex numbers `z`, the identity `Li(z) = Ei(ln(z))` holds.

The continuation of `Li` to the complex plane is chosen such that the resulting function is analytic with a singularity at 1 and a branch cut on the real axis left to that singularity; such that `conjugate(Li(z)) = Li(conjugate(z))` holds for non-real `z`; and such that `Li` is continuous from above on the negative real axis. Between 0 and 1, `Li` is real and thus neither continuous from above nor from below.

Environment Interactions

When called with a floating-point argument, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

For symbolic arguments, `Li` returns a symbolic function call in most cases:

```
Li(I), Li(0), Li(2), Li(x)
```

$$\mathrm{Li}(\mathrm{i}),\ 0,\ \mathrm{Li}(2),\ \mathrm{Li}(x)$$

Example 2

The integral logarithm of a large real number approximately equals the number of primes below that number:

```
numlib::pi(123456789), Li(123456789.0)
```

$$7027260,\ 7028122.595$$

Riemann suggested to use the approximation $\sum_{i=1}^{\infty} \mathrm{Li}\left(x^{1/i}\right)\ \mathrm{numlib::moebius}(i)$. This often gives a slightly better result, but it suffices to sum this series up to `i=2`:

```
R:= (x, n) -> _plus(float(Li(x^(1/i))) * numlib::moebius(i) $i=1..n):
for j from 1 to 9 do
  print(numlib::pi(10^j), float(Li(10^j)), R(10^j, 2), R(10^j, 50))
end_for:
delete j, R:
```

$$4,\ 6.165599505,\ 3.857761291,\ 5.065725444$$

$$25,\ 30.12614158,\ 23.96054208,\ 21.04138706$$

$$168,\ 177.609658,\ 164.1135957,\ 156.8772935$$

$$1229,\ 1246.137216,\ 1216.011074,\ 1202.985411$$

9592, 9629.809001, 9558.64347, 9536.769885

78498, 78627.54916, 78449.9395, 78413.40907

664579, 664918.405, 664455.4442, 664393.3058

5761455, 5762209.375, 5760963.238, 5760854.674

50847534, 50849234.96, 50845801.05, 50845606.04

Parameters

x

An arithmetical expression

Return Values

Arithmetical expression.

Overloaded By

x

See Also

MuPAD Functions
`Ci` | `Ei` | `int` | `ln` | `Shi` | `Si` | `Ssi`

READPATH

Search path for the command ‘ Read ’

Description

READPATH determines the directories, where the function `read` searches for files.

Possible values: String or a sequence of strings.

The variable READPATH can represent more than one search directory. This variable can be assigned a sequence of strings: each element of the sequence represents a directory in which files are search for.

Note: When concatenated with a file name, the directories given by the path variables must produce valid path names.

Path names are slightly system dependent. You can separate subdirectories with a `/` on all systems. On Windows systems, you may alternatively use a backslash character (\).

Note that in MuPAD, a single backslash inside a character string is created by typing two backslashes. E.g., the MuPAD string representing the path “`C:\Programs\MuPAD`” must be defined by `"C:\\Programs\\MuPAD"`.

The function `pathname` allows to create path names independent of the current operating system.

Examples

Example 1

This example shows how to define a READPATH. More than one path may be given. `read` will look for files to be opened in the directories given by READPATH. The following produces a valid READPATH for UNIX and Linux® systems only, since the path separators are hard coded in the strings:

```
READPATH := "math/lib/", "math/local/"
```

```
"math/lib/", "math/local/"
```

It is good programming style to use platform independent path strings. This can be achieved with the function `pathname`:

```
READPATH := pathname("math", "lib"),
            pathname("math", "local")
```

```
"math/lib/", "math/local/"
```

All path variables can be set to their default values by deleting them:

```
delete READPATH:
```

Example 2

The path variable `WRITEPATH` only accepts one path string:

```
WRITEPATH := "math/lib/", "math/local/"
```

```
Error: The argument is invalid. [WRITEPATH]
```

See Also

MuPAD Functions

`fclose` | `FILEPATH` | `finput` | `fopen` | `fprint` | `fread` | `ftextinput` | `NOTEBOOKFILE` | `NOTEBOOKPATH` | `pathname` | `print` | `protocol` | `read` | `write` | `WRITEPATH`

WRITEPATH

Search path for the command ‘ Write ’ et al.

Description

Possible values: String or a sequence of strings.

`WRITEPATH` determines the directory, into which the functions `fopen`, `fprint`, `write`, and `protocol` write files which are not specified with a full (absolute) pathname. If `WRITEPATH` is not defined, then the files are written into the “working directory”.

Note that the “working directory” depends on the operating system. On Windows systems, it is the folder where MuPAD is installed. On UNIX or Linux systems, the “working directory” is the directory where MuPAD was started.

Note: When concatenated with a file name, the directories given by the path variables must produce valid path names.

Path names are slightly system dependent. You can separate subdirectories with a `/` on all systems. On Windows systems, you may alternatively use a backslash character (\).

Note that in MuPAD, a single backslash inside a character string is created by typing two backslashes. E.g., the MuPAD string representing the path `"math\lib\"` must be defined by `"math\\lib\\"`.

The function `pathname` allows to create path names independent of the current operating system.

Examples

Example 1

This example shows how to define a `READPATH`. More than one path may be given. `read` will look for files to be opened in the directories given by `READPATH`. The following

produces a valid READPATH for UNIX and Linux systems only, since the path separators are hard coded in the strings:

```
READPATH := "math/lib/", "math/local/"
```

```
"math/lib/", "math/local/"
```

It is good programming style to use platform independent path strings. This can be achieved with the function pathname:

```
READPATH := pathname("math", "lib"),
            pathname("math", "local")
```

```
"math/lib/", "math/local/"
```

All path variables can be set to their default values by deleting them:

```
delete READPATH:
```

Example 2

The path variable WRITEPATH only accepts one path string:

```
WRITEPATH := "math/lib/", "math/local/"
```

```
Error: The argument is invalid. [WRITEPATH]
```

See Also

MuPAD Functions

fclose | FILEPATH | finput | fopen | fprint | fread | ftextinput | NOTEBOOKFILE | NOTEBOOKPATH | pathname | print | protocol | read | READPATH | write

limit

Compute a limit

Syntax

```
limit(f, x, <Left | Right | Real>, <Intervals>, <NoWarning>)
limit(f, x = x0, <Left | Right | Real>, <Intervals>, <NoWarning>)
```

Description

`limit(f, x = x0, Real)` computes the bidirectional limit $\lim_{x \to x_0} f(x)$, $x - x_0 \in \mathbb{R} \setminus \{0\}$.

`limit(f, x = x0, Left | Right)` computes the one-sided limit $\lim_{x \to x_0^-} f(x)$, $\lim_{x \to x_0^+} f(x)$ respectively.

`limit(f, x = x0, Intervals)` computes a set containing all accumulation points of $\lim_{x \to x_0} f(x)$, $x - x_0 \in \mathbb{R} \setminus \{0\}$.

`limit(f, x = x0, <Real>)` computes the bidirectional limit of `f` when `x` tends to `x0` on the real axis. The limit point `x0` may be omitted, in which case `limit` assumes `x0 = 0`.

If the limit point `x0` is *infinity* or - ∞, then the limit is taken from the left to *infinity* or from the right to - ∞, respectively.

If provably no limit exists, then `undefined` is returned. See “Example 2” on page 1-1250.

`limit(f, x = x0, Left)` returns the limit when `x` tends to `x0` from the left.
`limit(f, x = x0, Right)` returns the limit when `x` tends to `x0` from the right. See “Example 2” on page 1-1250.

If it cannot be determined whether a limit exist, or cannot determine its value, then a symbolic to `limit` is returned. See "Example 3" on page 1-1250. The same holds, in case the option `Intervals` is given, if no information on the set of accumulation points could be obtained.

If `f` contains parameters, then `limit` reacts to properties of those parameters set by `assume`. See "Example 5" on page 1-1251. It may also return a case analysis (`piecewise`) depending on these parameters.

You can compute the limit of a piecewise function. The conditions you use to define a piecewise function can depend on the limit variable. See "Example 6" on page 1-1252.

Internally, `limit` tries to determine the limit from a series expansion of `f` around `x = x0` computed via `series`. It may be necessary to increase the value of the environment variable `ORDER` in order to find the limit.

Note: `limit` works on a symbolic level and should not be called with arguments containing floating point arguments.

Environment Interactions

The function is sensitive to the environment variable `ORDER`, which determines the default number of terms in series computations (see `series`).

Properties of identifiers set by `assume` are taken into account.

Examples

Example 1

The following command computes $\lim_{x \to 0} \frac{1-\cos(x)}{x^2}$:

```
limit((1 - cos(x))/x^2, x)
```

$$\frac{1}{2}$$

A possible definition of e is given by the limit of the sequence $\left(1+\frac{1}{n}\right)^n$ for $n \to \infty$:

```
limit((1 + 1/n)^n, n = infinity)
```

$$e$$

Here is a more complex example:

```
limit(
  (exp(x*exp(-x)/(exp(-x) + exp(-2*x^2/(x+1)))) - exp(x))/x,
  x = infinity
)
```

$$-e^2$$

Example 2

The bidirectional limit of $f(x) = \frac{1}{x}$ for $x \to 0$ does not exist:

```
limit(1/x, x = 0)
```

$$\text{undefined}$$

You can compute the one-sided limits from the left and from the right by passing the options `Left` and `Right`, respectively:

```
limit(1/x, x = 0, Left),
limit(1/x, x = 0, Right)
```

$$-\infty, \infty$$

Example 3

If `limit` is not able to compute the limit, then a symbolic `limit` call is returned:

```
delete f: limit(f(x), x = infinity)
```

$$\lim_{x \to \infty} f(x)$$

Example 4

The function *sin*(*x*) oscillates for $x \to \infty$ between - 1 and 1; no accumulation points outside that interval exist:

```
limit(sin(x), x = infinity, Intervals)
```

$$[-1, 1]$$

In fact, all elements of the interval returned are accumulation points. This need not be the case in general. In the following example, the limit inferior and the limit superior are in fact $-\sqrt{2}$ and $\sqrt{2}$, respectively:

```
limit(sin(1/x) + cos(1/x), x = 0, Intervals)
```

$$[-2, 2]$$

Example 5

`limit` is not able to compute the limit of x^n for $x \to \infty$ without additional information about the parameter n:

```
assume(n in R_):
limit(x^n, x = infinity)
```

$$\begin{cases} 1 & \text{if } n = 0 \\ \infty & \text{if } 0 < n \\ 0 & \text{if } n < 0 \end{cases}$$

We can also `assume` immediately that $n > 0$ and get no case analysis then:

```
assume(n > 0): limit(x^n, x = infinity)
```

$$\infty$$

Similarly, we can assume that $n < 0$:

```
assume(n < 0): limit(x^n, x = infinity)
```

```
    0
```

```
delete n:
```

Example 6

Compute limit of the piecewise function:

```
limit(piecewise([x^3 > 10000*x, 1/x],
                [x^3 <= 10000*x, 10]),
                            x = infinity)
```

```
    0
```

Example 7

Compute limits of the incomplete Gamma function:

```
limit(igamma(z, t), t = infinity);
limit(igamma(z, t), t = 0)
```

```
    0
```

```
    Γ(z)
```

Parameters

f

An arithmetical expression representing a function in x

x

An identifier

x_0

The limit point: an arithmetical expression, possibly `infinity` or `-infinity`

Options

`Left`, `Real`, `Right`

This controls the direction of the limit computation. The option `Real` is the default case and means the bidirectional limit (i.e., there is no need to specify this option).

`Intervals`

Either `TRUE` or `FALSE`, by default `FALSE`. If this option is set to `TRUE`, then a superset of the set of all accumulation points is returned. If the result contains only one element, that element is the limit; on the other hand, if it contains more elements, not all of them are necessarily accumulation points, such that the limit may nevertheless exist.

`NoWarning`

If this option is set to `TRUE`, no warning messages are printed on the screen. Default is `FALSE`.

Return Values

arithmetical expression. If the option `Intervals` was given, the result is a (finite or infinite) set.

Overloaded By

f

Algorithms

`limit` uses an algorithm based on the thesis of Dominik Gruntz: “On Computing Limits in a Symbolic Manipulation System”, Swiss Federal Institute of Technology, Zurich,

Switzerland, 1995. If this fails, it tries to proceed recursively; finally, it attempts a series expansion.

See Also

MuPAD Functions

`asympt` | `diff` | `discont` | `int` | `O` | `series` | `taylor`

More About

- "If Limits Do Not Exist"

linsolve

Solve a system of linear equations

Syntax

```
linsolve(eqs, options)
```

```
linsolve(eqs, vars, options)
```

Description

`linsolve(eqs, vars)` solves a system of linear equations with respect to the unknowns `vars`.

`linsolve(eqs, < vars , < ShowAssumptions >>)` solves the linear system `eqs` with respect to the unknowns `vars`. If no unknowns are specified, then `linsolve` solves for all indeterminates in `eqs`; the unknowns are determined internally by `indets(eqs,PolyExpr)`.

`linsolve(eqs, vars, Domain = R)` solves the system over the domain `R`, which must be a field, i.e., a domain of category `Cat::Field`.

Note: Note that the return format does not allow to return kernel elements if elements of the domain `R` cannot be multiplied with the symbolic unknowns that span the kernel. In such a case, `linsolve` issues a warning and returns only a special solution. The kernel can be computed via `linalg::matlinsolve` for any field `R`.

Each element of `eqs` must be either an equation or an arithmetical expression `f`, which is considered to be equivalent to the equation `f = 0`.

The unknowns in `vars` need not be identifiers or indexed identifiers; expressions such as `sin(x)`, `f(x)`, or `y^(1/3)` are allowed as well. More generally, any expression accepted as indeterminate by `poly` is a valid unknown.

If the option `ShowAssumptions` is not given and the system is solvable, then the return value is a list of equations of the form `var = value`, where `var` is one of the

unknowns in `vars` and `value` is an arithmetical expression that does not involve any of the unknowns on the left side of a returned equation. Note that if the solution manifold has dimension greater than zero, then some of the unknowns in `vars` will occur on the right side of some returned equations, representing the degrees of freedom. See "Example 2" on page 1-1257.

If `vars` is a list, then the solved equations are returned in the the same order as the unknowns in `vars`.

The function `linsolve` can only solve systems of linear equations. Use `solve` for nonlinear equations.

`linsolve` is an interface function to the procedures `numeric::linsolve` and `linalg::matlinsolve`. For more details see the `numeric::linsolve`, `linalg::matlinsolve` help pages and the background section of this help page.

The system `eqs` is checked for linearity. Since such a test can be expensive, it is recommended to use `numeric::linsolve` or `linalg::matlinsolve` directly when you know that the system is linear.

Note: `linsolve` does *not* react to properties of identifiers set by `assume`.

Examples

Example 1

Equations and variables may be entered as sets or lists:

```
linsolve({x + y = 1, 2*x + y = 3}, {x, y}),
linsolve({x + y = 1, 2*x + y = 3}, [x, y]),
linsolve([x + y = 1, 2*x + y = 3], {x, y}),
linsolve([x + y = 1, 2*x + y = 3], [x, y])
```

$$[x = 2, y = -1], [x = 2, y = -1], [x = 2, y = -1], [x = 2, y = -1]$$

Also expressions may be used as variables:

```
linsolve({cos(x) + sin(x) = 1, cos(x) - sin(x) = 0},
```

```
          {cos(x), sin(x)})
```

$$\left[\cos(x) = \frac{1}{2}, \sin(x) = \frac{1}{2}\right]$$

Furthermore, indexed identifiers are valid, too:

```
linsolve({2*a[1] + 3*a[2] = 5, 7*a[2] + 11*a[3] = 13,
          17*a[3] + 19*a[1] = 23}, {a[1], a[2], a[3]})
```

$$\left[a_1 = \frac{691}{865}, a_2 = \frac{981}{865}, a_3 = \frac{398}{865}\right]$$

Next, we demonstrate the use of option `Domain` and solve a system over the field $\mathbb{Z}_{23}$ with it:

```
linsolve([2*x + y = 1, -x - y = 0],
         Domain = Dom::IntegerMod(23))
```

$$[x = 1 \bmod 23, y = 22 \bmod 23]$$

The following system does not have a solution:

```
linsolve({x + y = 1, 2*x + 2*y = 3}, {x, y})
```

FAIL

Example 2

If the solution of the linear system is not unique, then some of the unknowns are used as "free parameters" spanning the solution space. In the following example the unknown `z` is such a parameter. It does not appear on the left side of the solved equations:

```
eqs := [x + y = z, x + 2*y = 0, 2*x - z = -3*y, y + z = 0]:

vars := [w, x, y, z]:

linsolve(eqs, vars)
```

$$[x = 2\,z,\ y = -z]$$

Example 3

If you use the `Normal` option, `linsolve` calls the `normal` function for final results. This call ensures that `linsolve` returns results in normalized form:

```
linsolve([x + a*y = a + 1, b*x - y = b - 1], {x, y})
```

$$[x = 1,\ y = 1]$$

If you specify `Normal = FALSE`, `linsolve` does not call `normal` for the final result:

```
linsolve([x + a*y = a + 1, b*x - y = b - 1], {x, y}, Normal = FALSE)
```

$$\left[x = a - \frac{a\,(b\,(a+1) - b + 1)}{a\,b + 1} + 1,\ y = \frac{b\,(a+1) - b + 1}{a\,b + 1}\right]$$

Example 4

Solve this system:

```
eqs := [x + a*y = b, x + A*y = b]:
```

```
linsolve(eqs, [x, y])
```

$$[x = b,\ y = 0]$$

Note that more solutions exist for `a = A`. `linsolve` omits these solutions because it makes some additional assumptions on symbolic parameters of this system. To see the assumptions that `linsolve` made while solving this system, use the `ShowAssumptions` option:

```
linsolve(eqs, [x, y], ShowAssumptions)
```

$$[[x = b,\ y = 0],\ [],\ [A - a \neq 0]]$$

```
delete eqs:
```

Parameters

`eqs`

A list or a set of linear equations or arithmetical expressions

`vars`

A list or a set of unknowns to solve for: typically identifiers or indexed identifiers

Options

`Domain`

Option, specified as `Domain = R`

Solve the system over the field `R`, which must be a domain of category `Cat::Field`.

`Normal`

Option, specified as `Normal = b`

Return normalized results. The value `b` must be `TRUE` or `FALSE`. By default, `Normal = TRUE`, meaning that `linsolve` guarantees normalization of the returned results. Normalizing results can be computationally expensive.

By default, `linsolve` calls `normal` before returning results. This option affects the output only if the solution contains variables or exact expressions, such as `sqrt(5)` or `sin(PI/7)`.

To avoid this additional call, specify `Normal = FALSE`. In this case, `linsolve` also can return normalized results, but does not guarantee such normalization. See “Example 3” on page 1-1258.

`ShowAssumptions`

Return information about internal assumptions that `linsolve` made on symbolic parameters in `eqs`.

With this option, `linsolve` returns a list `[Solution, Constraints, Pivots]`. `Solution` is a list of solved equations representing the complete solution manifold of `eqs`, as described above. The lists `Constraints` and `Pivots` contain equations and

inequalities involving symbolic parameters in `eqs`. Internally, these were assumed to hold true when solving the system. See "Example 4" on page 1-1258.

When Gaussian elimination produces an equation `0 = c` with nonzero `c`, `linsolve` without `ShowAssumptions` returns `FAIL`. If `c` involves symbolic parameters, try using `linsolve` with `ShowAssumptions` to solve such systems. If the system is solvable, you will get the solution. In this case, an equation `0 = c` is returned in the `Constraints` list. If the system is not solvable, `linsolve` with `ShowAssumptions` returns `[FAIL, [], []]`.

Return Values

Without the `ShowAssumptions` option, a list of simplified equations is returned. It represents the general solution of the system `eqs`. `FAIL` is returned if the system is not solvable.

With `ShowAssumptions`, a list `[Solution, Constraints, Pivots]` is returned. `Solution` is a list of simplified equations representing the general solution of `eqs`. The lists `Constraints` and `Pivots` contain equations and inequalities involving symbolic parameters in `eqs`. Internally, these were assumed to hold true when solving the system.

Algorithms

If the option `Domain` is not present, the system is solved by calling `numeric::linsolve` with the option `Symbolic`.

If the option `Domain = R` is given and `R` is either `Dom::ExpressionField()` or `Dom::Float`, then `numeric::linsolve` is used to compute the solution of the system. This function uses a sparse representation of the equations.

Otherwise, `eqs` is first converted into a matrix and then solved by `linalg::matlinsolve`. A possibly sparse structure of the input system is not taken into account.

See Also

MuPAD Functions

`isolate` | `linalg::matlinsolve` | `numeric::linsolve` | `solve`

lllint

Compute an LLL-reduced basis of a lattice

Syntax

```
lllint(A)
```

Description

`lllint(A)` applies the LLL algorithm to the list of integer vectors `A`.

`lllint` applies the LLL algorithm to the entries of the list `A`. The entries of `A` must be lists of integers, all of the same length; the number of lists need not equal that length.

The return value of `lllint` has the same form.

The computations are done entirely with integers and are both accurate and quite fast.

Examples

Example 1

We apply the LLL algorithm to a list of two vectors of length three:

```
A := [[1, 2, 3], [4, 5, 6]]:
lllint(A)
```

$$[[2, 1, 0], [-1, 1, 3]]$$

The result is to be interpreted as follows: the two vectors in the output form an LLL-reduced basis of the lattice genberated by the two vectors in the input.

Example 2

If the input vectors are not linearly independent, `FAIL` is returned:

```
lllint([[1, 2], [2, 4]])
```

```
    FAIL
```

Parameters

A

A list of vectors, each being a list of integers

Return Values

list of lists is returned whose entries form an LLL-reduced basis of the lattice spanned by the entries of `A`. If the entries of `A` are not linearly independent, `FAIL` is returned.

References

A. K. Lenstra, H. W. Lenstra Jr., and L. Lovasz, Factoring polynomials with rational coefficients. Math. Ann. 261, 1982, pp. 515–534.

Joachim von zur Gathen and Jürgen Gerhard, Modern Computer Algebra. Cambridge University Press, 1999, Chapter 16.

George L. Nemhauser and Laurence A. Wolsey, Integer and Combinatorial Optimization. New York, Wiley, 1988.

A. Schrijver, Theory of Linear and Integer Programming. New York, Wiley, 1986.

See Also

MuPAD Functions

`linalg::basis` | `linalg::factorLU` | `linalg::factorQR` | `linalg::gaussElim` | `linalg::hermiteForm` | `linalg::orthog`

lmonomial

Leading monomial of a polynomial

Syntax

```
lmonomial(p, <order>, <Rem>)

lmonomial(f, <vars>, <order>, <Rem>)
```

Description

`lmonomial(p)` returns the leading monomial of the polynomial `p`.

The returned monomial is "leading" with respect to the lexicographical ordering, unless a different ordering is specified via the argument `order`. Cf. "Example 1" on page 1-1263.

The leading monomial of the zero polynomial is the zero polynomial.

A polynomial expression `f` is first converted to a polynomial with the variables given by `vars`. If no variables are given, they are searched for in `f`. See `poly` about details of the conversion. The result is returned as polynomial expression. `FAIL` is returned if `f` cannot be converted to a polynomial. Cf. "Example 4" on page 1-1265.

The result of `lmonomial` is not fully evaluated. It can be evaluated by the functions `mapcoeffs` and `eval`. Cf. "Example 3" on page 1-1264.

Examples

Example 1

We demonstrate how various orderings influence the result:

```
p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x, y, z]):
lmonomial(p), lmonomial(p, DegreeOrder),
lmonomial(p, DegInvLexOrder)
```

$$\text{poly}(5\,x^4, [x, y, z]),\ \text{poly}(4\,x^3\,y\,z^2, [x, y, z]),\ \text{poly}(3\,x^2\,y^3\,z, [x, y, z])$$

The following call uses the reverse lexicographical order on 3 indeterminates:

```
lmonomial(p, Dom::MonomOrdering(RevLex(3)))
```

$$\text{poly}(3\,x^2\,y^3\,z, [x, y, z])$$

```
delete p:
```

Example 2

We compute the reductum of a polynomial:

```
p := poly(2*x^2*y + 3*x*y^2 + 6, [x, y]):
q := lmonomial(p, Rem)
```

$$[\text{poly}(2\,x^2\,y, [x, y]),\ \text{poly}(3\,x\,y^2 + 6, [x, y])]$$

The leading monomial and the reductum add up to the polynomial `p`:

```
p = q[1] + q[2]
```

$$\text{poly}(2\,x^2\,y + 3\,x\,y^2 + 6, [x, y]) = \text{poly}(2\,x^2\,y + 3\,x\,y^2 + 6, [x, y])$$

```
delete p, q:
```

Example 3

We demonstrate the evaluation strategy of `lmonomial`:

```
p := poly(6*x^6*y^2 + x^2 + 2, [x]): y := 4: lmonomial(p)
```

$$\text{poly}((6\,y^2)\,x^6, [x])$$

Evaluation is enforced by `eval`:

```
mapcoeffs(%, eval)
```

$$\mathrm{poly}(96\,x^6, [x])$$

```
delete p, y:
```

Example 4

The expression `1/x` may not be regarded as polynomial:

```
lmonomial(1/x)
```

$$\mathrm{FAIL}$$

Parameters

`p`

A polynomial of type `DOM_POLY`

`f`

A polynomial expression

`vars`

A list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

`order`

The term ordering: either `LexOrder` or `DegreeOrder` or `DegInvLexOrder` or a user-defined term ordering of type `Dom::MonomOrdering`. The default is the lexicographical ordering `LexOrder`.

Options

`Rem`

Makes `lmonomial` return a list with two polynomials: the leading monomial and the reductum. The reductum of a polynomial `p` is `p - lmonomial(p)`.

Return Values

Polynomial of the same type as `p`. An expression is returned if an expression is given as input. `FAIL` is returned if the input cannot be converted to a polynomial. With `Rem`, a list of two polynomials is returned.

Overloaded By

p

See Also

MuPAD Functions

`coeff` | `degree` | `degreevec` | `ground` | `lcoeff` | `ldegree` | `lterm` | `monomials` | `nterms` | `nthcoeff` | `nthmonomial` | `nthterm` | `poly` | `poly2list` | `tcoeff`

ln

Natural logarithm

Syntax

```
ln(x)
```

Description

`ln(x)` represents the natural logarithm of `x`.

Natural logarithm is defined for all complex arguments $x \neq 0$.

`ln` applies the following simplification rules to its arguments:

- If `x` is of the type `Type::Numeric`, then $\ln(e^x) = x + k\,i\,2\,\pi$. Here `k` is an integer, such that the imaginary part of the result lies in the interval $(-\pi, \pi]$. Similar simplifications occur for $(e^y)^a$.
- If `x` is a negative integer or a negative rational, then $ln(x) = i\,\pi + ln(-x)$.
- If `x` is an integer, then $\ln\left(\frac{1}{x}\right) = -\ln(x)$.
- `ln` uses the following special values:

  $ln(1) = 0$, $ln(-1) = i\,\pi$, $\ln(i) = \frac{i\,\pi}{2}$, $\ln(-i) = -\frac{i\,\pi}{2}$, $ln(\infty) = \infty$, $ln(-\infty) = i\,\pi + \infty$.

For exact numeric and symbolic arguments, `ln` typically returns unresolved function calls.

If an argument is a floating-point value, `ln` returns a floating-point result. The imaginary part of the result takes values in the interval $(-\pi, \pi]$. The negative real axis is a branch cut; the imaginary part of the result jumps when crossing the cut. On the negative real axis, the imaginary part is π according to $ln(x) = i\,\pi + ln(-x)$, $x < 0$. See “Example 3” on page 1-1269.

If an argument is a floating-point interval of type DOM_INTERVAL, ln returns the results of type DOM_INTERVAL, properly rounded outwards. This implies that the result contains only real numbers. See “Example 4” on page 1-1269.

Arithmetical rules such as $ln(x\,y) = ln(x) + ln(y)$ are not valid throughout the complex plane. Use properties to mark identifiers as real and apply functions such as expand, combine or simplify to manipulate expressions involving ln. See “Example 5” on page 1-1270.

Environment Interactions

When called with a floating-point argument, the function is sensitive to the environment variable DIGITS which determines the numerical working precision.

Examples

Example 1

Compute the natural logarithms of these numeric and symbolic values:

```
ln(2), ln(-3), ln(1/4), ln(1 + I), ln(x^2)
```

$$\ln(2),\ \ln(3) + \pi\,i,\ -\ln(4),\ \ln(1+i),\ \ln(x^2)$$

For floating-point arguments, ln returns floating-point results:

```
ln(123.4), ln(5.6 + 7.8*I), ln(1.0/10^20)
```

$$4.815431111,\ 2.261980065 + 0.948125538\,i,\ -46.05170186$$

ln applies special simplification rules to its arguments:

```
ln(1), ln(-1), ln(exp(-5)), ln(exp(5 + 27/4*I))
```

$$0,\ \pi\,i,\ -5,\ 5 - 2\,\pi\,i + \frac{27\,i}{4}$$

Example 2

`diff`, `float`, `limit`, `series` and similar functions handle expressions involving `ln`:

```
diff(ln(x^2), x)
```

$$\frac{2}{x}$$

```
float(ln(PI + I))
```

$$1.192985153 + 0.3081690711\,i$$

```
limit(ln(x)/x, x = infinity)
```

$$0$$

```
series(x*ln(sin(x)), x = 0, 10)
```

$$x\ln(x) - \frac{x^3}{6} - \frac{x^5}{180} - \frac{x^7}{2835} - \frac{x^9}{37800} + O(x^{11})$$

Example 3

The negative real axis is a branch cut. The imaginary part of the values returned by `ln` jump when crossing this cut:

```
ln(-2.0), ln(-2.0 + I/10^1000), ln(-2.0 - I/10^1000)
```

$$0.6931471806 + 3.141592654\,i,\ 0.6931471806 + 3.141592654\,i,\ 0.6931471806 - 3.141592654\,i$$

Example 4

The natural logarithm of an interval is the image set of the logarithm function over the set represented by the interval:

```
ln(1 ... 2)
```

```
0.0 ... 0.6931471806
```

```
ln(-1 ... 1)
```

```
RD_NINF ... 0.0 ∪ RD_NINF ... 0.0 + 3.141592653 ... 3.141592654 i
```

This definition extends to unions of intervals:

```
ln(1 ... 2 union 3 ... 4)
```

```
0.0 ... 0.6931471806 ∪ 1.098612288 ... 1.386294362
```

Example 5

`expand`, `combine`, and `simplify` react to properties set via `assume`. The following call does not produce an expanded result, because the arithmetical rule $ln(x\,y) = ln(x) + ln(y)$ does not hold for arbitrary complex x, y:

```
expand(ln(x*y))
```

$$\ln(x\,y)$$

If one of the factors is real and positive, the rule is valid:

```
assume(x > 0): expand(ln(x*y))
```

$$\ln(x) + \ln(y)$$

```
combine(%, ln)
```

$$\ln(x\,y)$$

```
simplify(ln(x^3*y) - ln(x))
```

$$\ln(x^2\,y)$$

For further computations, clear the assumption:

```
unassume(x):
```

Parameters

x

An arithmetical expression

Return Values

Arithmetical expression

Overloaded By

x

See Also

MuPAD Functions

`dilog` | `log` | `log10` | `log2` | `polylog`

log

Logarithm to arbitrary base

Syntax

```
log(b, x)
```

```
log(x)
```

Description

Note: The output of syntax `log(b, x)` has been changed and is rewritten in terms of natural logarithms as `ln(x)/ln(b)`.

`log(b, x)` represents the logarithm of `x` to the base `b`.

`log(x)` is an alias for the natural logarithm `ln(x)`.

Mathematically, $\log_b(x)$ coincides with `ln(x)/ln(b)`. When you call `log`, the result is rewritten in terms of the natural logarithms.

The logarithm is defined for all complex arguments $x \neq 0$. The base b, however, is assumed to be real, positive and not equal to 1.

Note: For symbolic `b`, MuPAD applies simplifications based on these assumptions.

`log` applies the following simplification rules to its arguments:

- $\log_b(b^x) = x$ in the following cases:
  - `b` is a symbolic (indexed) identifier and `x` is of type `Type::Real`
  - `b` is numerical and `x` is integer or rational.

  Mathematically, this rule is valid for any real value `x`.

- If `x` is a negative integer or a negative rational, then $\log_b(x) = \frac{i\pi}{\ln(b)} + \frac{\ln(-x)}{\ln(b)}$.
- If x is an integer, then $\log_b\left(\frac{1}{x}\right) = -\frac{\ln(x)}{\ln(b)}$.
- `log` uses the following special values:

$$\log_b(1) = 0,\ \log_b(-1) = \frac{i\pi}{\ln(b)},\ \log_b(i) = \frac{i\pi}{2\ln(b)},\ \log_b(-i) = -\frac{i\pi}{2\ln(b)}.$$

For exact numeric and symbolic arguments, `log` rewrites the function call in terms of the natural logarithm.

If both arguments are numerical and at least one of them is a floating-point number, `log` returns a floating-point result. The imaginary part of the result takes values in the interval $\left(-\frac{\pi}{\ln(b)}, \frac{\pi}{\ln(b)}\right]$ if $b > 1$ and in the interval $\left[\frac{\pi}{\ln(b)}, -\frac{\pi}{\ln(b)}\right)$ if $b < 1$. The negative real axis is a branch cut, the imaginary part of the result jumps when crossing the cut. On the negative real axis, the imaginary part is $\frac{\pi}{\ln(b)}$ according to $\log_b(x) = \frac{i\pi}{\ln(b)} + \frac{\ln(-x)}{\ln(b)}$, $x < 0$. See “Example 3” on page 1-1275.

Arithmetical rules such as $\log_b(x\,y) = \frac{\ln(x)}{\ln(b)} + \frac{\ln(y)}{\ln(b)}$ are not valid throughout the complex plane. Use properties to mark identifiers as real and apply functions such as `expand` or `simplify` to manipulate expressions involving `log`. See “Example 4” on page 1-1275.

Environment Interactions

When called with a floating-point argument, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

When computing a logarithm to an arbitrary base, use identifiers, indexed identifiers, or numbers of type `Type::Positive` to specify the base of a logarithm:

```
log(b, 2), log(b[1], 3), log(2, 5), log(2/3, 4/9), log(0.3, x)
```

$$\frac{\ln(2)}{\ln(b)}, \frac{\ln(3)}{\ln(b_1)}, \frac{\ln(5)}{\ln(2)}, 2, -0.8305835451\,\ln(x)$$

Do not use general arithmetical expressions to specify the base:

```
log(-PI^2, 2)
```

```
Error: The base must be an identifier, an indexed identifier, or a positive real numbe
```

For floating-point arguments, `log` returns floating-point results:

```
log(2, 123.4), log(2.0, 5.6 + 7.8*I), log(10.0, 2/10^20)
```

$$6.947198584,\ 3.263347423 + 1.367856012\,i,\ -19.69897$$

`log` applies special simplification rules to its arguments:

```
log(b, 1), log(b, -1), log(2/3, (4/9)^10), log(b, b^(-5))
```

$$0, \frac{\pi\,i}{\ln(b)}, 20, -5$$

Example 2

`diff`, `float`, `limit`, `series` and similar functions handle expressions involving `log`:

```
diff(log(b, x^2), x)
```

$$\frac{2}{x\,\ln(b)}$$

```
float(log(10, PI + I))
```

$$0.5181068691 + 0.1338361271\,i$$

```
limit(log(10, x)/x, x = infinity)
```

$$0$$

```
series(x*log(x, sin(x)), x = 0)
```

$$x - \frac{x^3}{6\ln(x)} - \frac{x^5}{180\ln(x)} + O(x^7)$$

Example 3

The negative real axis is a branch cut. The imaginary part of the values returned by `log` jump when crossing this cut:

```
log(10, -2.0),
log(10, -2.0 + I/10^1000),
log(10, -2.0 - I/10^1000)
```

$$0.3010299957 + 1.364376354\,i,\ 0.3010299957 + 1.364376354\,i,\ 0.3010299957 - 1.364376354\,i$$

Example 4

`expand` and `simplify` react to properties set via `assume`. The following call does not produce an expanded result, because the arithmetical rule $\log_b(x\,y) = \log_b(x) + \log_b(y)$ does not hold for arbitrary complex x, y:

```
expand(log(10, x*y))
```

$$\frac{\ln(x\,y)}{\ln(10)}$$

If one of the factors is real and positive, the rule is valid:

```
assume(x > 0): expand(log(b, x*y))
```

$$\frac{\ln(x)}{\ln(b)} + \frac{\ln(y)}{\ln(b)}$$

```
simplify(log(b, x^3*y) - log(b, x))
```

$$\frac{\ln(x^2\,y)}{\ln(b)}$$

For further computations, clear the assumption:

```
unassume(x):
```

Parameters

b

An identifier of domain type `DOM_IDENT`, indexed identifier, real numerical value of type `Type::Positive`, or the expression `exp(1)` that leads to the natural logarithm: `log(exp(1), x)` = `ln(x)`.

x

An arithmetical expression

Return Values

Arithmetical expression

Overloaded By

x

See Also

MuPAD Functions

`dilog` | `ln` | `log10` | `log2` | `polylog`

log10

Logarithm to base 10

Syntax

```
log10(x)
```

Description

`log10(x)` represents the logarithm of `x` to the base 10.

Mathematically, `log10(x)` is equivalent to `log(10, x)`. See "Example 1" on page 1-1277.

The logarithm to the base 10 is defined for all complex arguments $x \neq 0$.

`log10(x)` rewrites logarithms to the base 10 in terms of the natural logarithm: `log10(x) = ln(x)/ln(10)`. See "Example 2" on page 1-1278.

See the `ln` help page for details.

Environment Interactions

When called with a floating-point argument, this function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Compute these logarithms using `log10`:

```
log10(10), log10(1000), log10(1)
```

```
    1, 3, 0
```

Compute the same logarithms using `log` with 10 as the first argument:

```
log(10, 10), log(10, 1000), log(10, 1)
```

$$1, 3, 0$$

Example 2

`log10` rewrites logarithms in terms of `ln`:

```
log10(x), log10(x^2 - 1)
```

$$\frac{\ln(x)}{\ln(10)}, \frac{\ln(x^2 - 1)}{\ln(10)}$$

Example 3

For floating-point values, `log10` returns floating-point results:

```
log10(123.4), log10(5.6 + 7.8*I), log10(-15.45)
```

$$2.09131516,\ 0.9823654605 + 0.4117656893\ i,\ 1.188928484 + 1.364376354\ i$$

Example 4

For floating-point intervals, `log10` returns results as floating-point intervals:

```
log10(2.0...10.15)
```

$$0.3010299956 \ldots 1.006466043$$

Parameters

x

An arithmetical expression

Return Values

Arithmetical expression

Overloaded By

x

See Also

MuPAD Functions
`dilog` | `ln` | `log` | `log2` | `polylog`

log2

Logarithm to base 2

Syntax

```
log2(x)
```

Description

`log2(x)` represents the logarithm of `x` to the base 2.

Mathematically, `log2(x)` is equivalent to `log(2, x)`. See "Example 1" on page 1-1280.

The logarithm to the base 2 is defined for all complex arguments $x \neq 0$.

`log2(x)` rewrites logarithms to the base 2 in terms of the natural logarithm: `log2(x) = ln(x)/ln(2)`. See "Example 2" on page 1-1281.

See the `ln` help page for details.

Environment Interactions

When called with a floating-point argument, this function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Compute these logarithms using `log2`:

```
log2(2), log2(8), log2(1)
```

```
     1, 3, 0
```

Compute the same logarithms using `log` with 2 as the first argument:

```
log(2, 2), log(2, 8), log(2, 1)
```

$$1,\ 3,\ 0$$

Example 2

`log2` rewrites logarithms in terms of `ln`:

```
log2(x), log2(x^2 - 1)
```

$$\frac{\ln(x)}{\ln(2)},\ \frac{\ln(x^2-1)}{\ln(2)}$$

Example 3

For floating-point values, `log2` returns floating-point results:

```
log2(123.4), log2(5.6 + 7.8*I), log2(-15.45)
```

$$6.947198584,\ 3.263347423 + 1.367856012\,i,\ 3.949534933 + 4.532360142\,i$$

Example 4

For floating-point intervals, `log2` returns results as floating-point intervals:

```
log2(2.0...10.15)
```

$$0.9999999999 \ldots 3.343407823$$

Parameters

x

An arithmetical expression

Return Values

Arithmetical expression

Overloaded By

x

See Also

MuPAD Functions
`dilog` | `ln` | `log` | `log10` | `polylog`

lterm

Leading term of a polynomial

Syntax

```
lterm(p, <order>)
```

```
lterm(f, <vars>, <order>)
```

Description

`lterm(p)` returns the leading term of the polynomial `p`.

The returned term is "leading" with respect to the lexicographical ordering, unless a different ordering is specified via the argument `order`. Cf. "Example 1" on page 1-1283.

The identity `lterm(p)*lcoeff(p) = lmonomial(p)` holds.

The leading term of the zero polynomial is the zero polynomial.

A polynomial expression `f` is first converted to a polynomial with the variables given by `vars`. If no variables are given, they are searched for in `f`. See `poly` about details of the conversion. The result is returned as polynomial expression. `FAIL` is returned if `f` cannot be converted to a polynomial. Cf. "Example 3" on page 1-1284.

Examples

Example 1

We demonstrate how various orderings influence the result:

```
p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x, y, z]):
lterm(p), lterm(p, DegreeOrder), lterm(p, DegInvLexOrder)
```

$$\mathrm{poly}(x^4, [x, y, z]), \mathrm{poly}(x^3\, y\, z^2, [x, y, z]), \mathrm{poly}(x^2\, y^3\, z, [x, y, z])$$

The following call uses the reverse lexicographical order on 3 indeterminates:

```
lterm(p, Dom::MonomOrdering(RevLex(3)))
```

$$\mathrm{poly}(x^2\, y^3\, z, [x, y, z])$$

```
delete p:
```

Example 2

The leading monomial is the product of the leading coefficient and the leading term:

```
p := poly(2*x^2*y + 3*x*y^2 + 6, [x, y]):
mapcoeffs(lterm(p),lcoeff(p)) = lmonomial(p)
```

$$\mathrm{poly}(2\, x^2\, y, [x, y]) = \mathrm{poly}(2\, x^2\, y, [x, y])$$

```
delete p:
```

Example 3

The expression `1/x` may not be regarded as polynomial:

```
lterm(1/x)
```

$$\mathrm{FAIL}$$

Parameters

`p`

A polynomial of type `DOM_POLY`

`f`

A polynomial expression

vars

A list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

order

The term ordering: either `LexOrder` or `DegreeOrder` or `DegInvLexOrder` or a user-defined term ordering of type `Dom::MonomOrdering`. The default is the lexicographical ordering `LexOrder`.

Return Values

Polynomial of the same type as `p`. An expression is returned if an expression is given as input. `FAIL` is returned if the input cannot be converted to a polynomial.

Overloaded By

p

See Also

MuPAD Functions

`coeff` | `degree` | `degreevec` | `ground` | `lcoeff` | `ldegree` | `lmonomial` | `monomials` | `nterms` | `nthcoeff` | `nthmonomial` | `nthterm` | `poly` | `poly2list` | `tcoeff`

match

Pattern matching

Syntax

```
match(expression, pattern, options)
```

Description

`match(expression, pattern)` checks whether the syntactical structure of `expression` matches `pattern`. If so, the call returns a set of replacement equations transforming `pattern` into `expression`.

`match` computes a set of replacement equations `S` for the identifiers occurring in `pattern`, such that `subs(pattern, S)` and `expression` coincide up to associativity, commutativity, and neutral elements.

Without additional options, a purely syntactical matching is performed; associativity, commutativity, or neutral elements are taken into account only for the builtin operators `+` and `*`, `and` and `or`, and `union` and `intersect`. In this case, `subs(pattern, S) = expression` holds for the set `S` of replacement equations returned by `match` if the matching was successful. Cf. “Example 1” on page 1-1287. You can declare these properties for operators via the options `Associative`, `Commutative`, and `Null` (see below). Then `subs(pattern, S)` and `expression` need no longer be equal in MuPAD, but they can be transformed into each other by application of the rules implied by the options.

Both `expression` and `pattern` may be arbitrary MuPAD expressions, i.e., both atomic expressions such as numbers, Boolean constants, and identifiers, and compositeexpressions.

Each identifier without a value that occurs in `pattern`, including the 0th operands, is regarded as a *pattern variable*, in the sense that it may be replaced by some expression in order to transform `pattern` into `expression`. Use the option `Const` (see below) to declare identifiers as non-replaceable.

With the exception of some automatic simplifications performed by the MuPAD kernel, distributivity is *not* taken into account. Cf. “Example 5” on page 1-1289.

Note: `match` evaluates its arguments, as usual. This evaluation usually encompasses a certain amount of simplification, which may change the syntactical structure of both `expression` and `pattern` in an unexpected way. Cf. “Example 6” on page 1-1289.

Even if there are several possible matches, `match` returns at most one of them. Cf. “Example 7” on page 1-1290.

If the structure of `expression` does not match `pattern`, `match` returns `FAIL`.

If `expression` and `pattern` are equal, the empty set is returned.

Otherwise, if a match is found and `expression` and `pattern` are different, then a set `S` of replacement equations is returned. For each pattern variable `x` occurring in `pattern` that is not declared constant via the option `Const`, `S` contains exactly one replacement equation of the form `x = y`, and `y` is the expression to be substituted for `x` in order to transform `pattern` into `expression`.

Examples

Example 1

All identifiers of the following pattern are pattern variables:

```
match(f(a, b), f(X, Y))
```

$$\{X = a, Y = b, f = f\}$$

The function `f` is declared non-replaceable:

```
match(f(a, b), f(X, Y), Const = {f})
```

$$\{X = a, Y = b\}$$

Example 2

The following call contains a condition for the pattern variable `X`:

```
match(f(a, b), f(X, Y), Const = {f}, Cond = {X -> not has(X, a)})
```

FAIL

If the function `f` is declared commutative, the expression matches the given pattern—in contrast to the preceding example:

```
match(f(a, b), f(X, Y), Const = {f}, Commutative = {f},
                        Cond = {X -> not has(X, a)})
```

$\{X = b, Y = a\}$

Example 3

The following expression cannot be matched since the number of arguments of the expression and the pattern are different:

```
match(f(a, b, c), f(X, Y), Const = {f})
```

FAIL

We declare the function `f` associative with the option `Associative`. In this case the pattern matches the given expression:

```
match(f(a, b, c), f(X, Y), Const = {f}, Associative = {f})
```

$\{X = a, Y = f(b, c)\}$

Example 4

If, however, the function call in the pattern has more arguments than the corresponding function call in the expression, no match is found:

```
match(f(a, b), f(X, Y, Z), Const = {f}, Associative = {f})
```

FAIL

If the neutral element with respect to the operator `f` is known, additional matches are possible by substituting it for some of the pattern variables:

```
match(f(a, b), f(X, Y, Z), Const = {f},
```

```
Associative = {f}, Null = {f = 0})
```

$$\{X = a, Y = b, Z = 0\}$$

Example 5

Distributivity is *not* taken into account in general:

```
match(a*x + a*y, a*(X + Y), Const = {a})
```

FAIL

The next call finds a match, but not the expected one:

```
match(a*(x + y), X + Y)
```

$$\{Y = 0, X = a\,(x + y)\}$$

The following declarations and conditions do not lead to the expected result, either:

```
match(a*(x + y), a*X + a*Y, Const = {a},
                            Cond = {X -> X <> 0, Y -> Y <> 0})
```

FAIL

Example 6

Automatic simplifications can “destroy” the structure of the given expression or pattern:

```
match(sin(-2), sin(X))
```

FAIL

The result is `FAIL`, because the first argument `sin(-2)` is evaluated and rewritten to `-sin(2)`:

```
sin(-2)
```

$$-\sin(2)$$

You can circumvent this problem by using `hold`:

```
match(hold(sin(-2)), sin(X))
```

$$\{X = -2\}$$

Example 7

`match` returns only one possible match:

```
match(a + b + c + 1, X + Y)
```

$$\{X = a,\ Y = b + c + 1\}$$

To obtain other solutions, use conditions to exclude the solutions that you already have:

```
match(a + b + c + 1, X + Y, Cond = {X <> a})
```

$$\{Y = a,\ X = b + c + 1\}$$

```
match(a + b + c + 1, X + Y, Cond = {X <> a and Y <> a})
```

$$\{X = b,\ Y = a + c + 1\}$$

```
match(a + b + c + 1, X + Y,
      Cond = {X <> a and X <> b and Y <> a})
```

$$\{X = c,\ Y = a + b + 1\}$$

Example 8

Every pattern variable can have at most one condition procedure. Simple conditions can be given by anonymous procedures (`->`):

```
match(a + b, X + Y, Cond = {X -> X <> a, Y -> Y <> b})
```

$$\{X = b,\ Y = a\}$$

Several conditions on a pattern variable can be combined in one procedure:

```
Xcond := proc(X) begin
  if domtype(X) = DOM_IDENT then
    X <> a and X <> b
  else
    X <> 0
  end_if
end_proc:

match(sin(a*b), sin(X*Y), Cond = {Xcond})
```

$$\{Y = 1, X = a\,b\}$$

```
match(sin(a*c), sin(X*Y), Cond = {Xcond})
```

$$\{Y = a, X = c\}$$

```
delete Xcond:
```

Parameters

`expression`

A MuPAD expression

`pattern`

The pattern: a MuPAD expression

`option1, option2, …`

Optional arguments as listed below

Options

`Associative`

Option, specified as `Associative = {f1, f2, …}`

It is assumed that identifiers `f1, f2, ...` represent associative operators and may take an arbitrary number of arguments, i.e., expressions such as `f1(f1(a, b), c)`, `f1(a, f1(b, c))`, and `f1(a, b, c)` are considered equal.

No special rules for associative operators with less than two arguments apply. In particular, `f1(a)` and `a` are *not* considered equal.

`Commutative`

Option, specified as `Commutative = {g1, g2, …}`

It is assumed that the identifiers `g1, g2, ...` represent commutative operators, i.e., expressions such as `g1(a, b)` and `g1(b, a)` are considered equal.

`Cond`

Option, specified as `Cond = {p1, p2, …}`

Only matches satisfying the conditions specified by the procedures `p1, p2, ...` are considered. Each procedure must take exactly one argument and represents a condition on exactly one pattern variable. The name of the procedure's formal argument must be equal to the name of a pattern variable occurring in `pattern` that is not declared constant via the option `Const`. Each condition procedure must return an expression that the function `bool` can evaluate to one of the Boolean values `TRUE` or `FALSE`.

Anonymous procedures created via `->` can be used to express simple conditions. Cf. “Example 8” on page 1-1290.

If a possible match is found, given by a set of replacement equations `S`, then `match` checks whether all specified conditions are satisfied by calling `bool(p1(y1) and p2(y2) and ...)`, where `y1` is the expression to be substituted for the pattern variable `x1` that agrees with the formal argument of the procedure `p1`, etc. If the return value of the call is `TRUE`, then `match` returns `S`. Otherwise, the next possible match is tried.

For example, if `p1` is a procedure with formal argument `x1`, where `x1` is a pattern variable occurring in `pattern`, then a match `S = {..., x1 = y1, ...}` is considered valid only if `bool(p1(y1))` returns `TRUE`.

There can be at most one condition procedure for each pattern variable. If necessary, use the logical operators `and` and `or` as well as the control structures `if` and `case` to combine several conditions for the same pattern variable in one condition procedure. Cf. “Example 8” on page 1-1290.

`Const`

Option, specified as `Const = {c1, c2, …}`

The identifiers `c1, c2, ...` are regarded as constants, i.e., they must match literally and must not be replaced in order to transform `pattern` into `expression`.

`Null`

Option, specified as `Null = {h1 = e1, h2 = e2, …}`

It is assumed that `e1, e2, ...` are the neutral elements with respect to the associative operations `h1, h2, ...` i.e., expressions such as `h1(a, e1)`, `h1(e1, a)`, and `h1(a)` are considered equal.

This declaration affects only operators that are declared associative via the option `Associative`. Moreover, the neutral elements are not implicitly assumed to be constants.

Return Values

Set of replacement equations, or `FAIL`.

See Also

MuPAD Functions

`simplify` | `subs` | `subsex` | `subsop`

map

Apply a function to all operands of an object

Syntax

```
map(object, f, <p1, p2, , …>)

map(object, f, <p1, p2, , …>, <Unsimplified>)
```

Description

`map(object, f)` returns a copy of `object` where each operand `x` has been replaced by `f(x)`. The object itself is not modified by `map` (see “Example 2” on page 1-1296).

The second argument `f` may be a procedure generated via `->` or `proc` (e.g., `x -> x^2 + 1`), a function environment (e.g., `sin`), or a functional expression (e.g., `sin@exp + 2*id`).

If optional arguments are present, then each operand `x` of `object` is replaced by `f(x, p1, p2, ...)` (see “Example 1” on page 1-1295).

It is possible to apply an operator, such as `+` or `*`, to all operands of `object`, by using its functional equivalent, such as `_plus` or `_mult`. See “Example 1” on page 1-1295.

In contrast to `op`, `map` does not decompose rational numbers and complex numbers further. Thus, if the argument is a rational number or a complex number, then `f` is applied to the number itself and not to the numerator and the denominator or the real part and the imaginary part, respectively (see “Example 3” on page 1-1296).

If `object` is a string, then `f` is applied to the string as a whole and not to the individual characters (see “Example 3” on page 1-1296).

If `object` is an expression, then `f` is applied to the operands of `f` as returned by `op` (see “Example 1” on page 1-1295).

If `object` is an expression sequence, then this sequence is not flattened by `map` (see “Example 4” on page 1-1297).

If `object` is a polynomial, then `f` is applied to the polynomial itself and not to all of its coefficients. Use `mapcoeffs` to achieve the latter (see “Example 3” on page 1-1296).

If `object` is a list, a set, an array, or an hfarray, then the function `f` is applied to all elements of the corresponding data structure.

Note: If `object` is a table, the function `f` is applied to all *entries* of the table, not to the indices (see “Example 9” on page 1-1300). The entries are the right sides of the operands of a table.

If `object` is an element of a library domain, then the slot `"map"` of the domain is called and the result is returned. This can be used to extend the functionality of `map` to user-defined domains. If no `"map"` slot exists, then `f` is applied to the object itself (see “Example 10” on page 1-1300).

`map` does not evaluate its result after the replacement; use `eval` to achieve this. Nevertheless, internal simplifications occur after the replacement, unless the option `Unsimplified` is given (see “Example 8” on page 1-1299).

`map` does not descend recursively into an object; the function `f` is only applied to the operands at first level. Use `misc::maprec` for a recursive version of `map` (see “Example 11” on page 1-1301).

The procedure `f` should be deterministic and should not have side effects (such as changing and using global variables). The user does not have any control over the ordering in which the function is applied to the operands of the object!

Examples

Example 1

`map` works for expressions:

```
map(a + b + 3, sin)
```

$$\sin(3) + \sin(a) + \sin(b)$$

The optional arguments of map are passed to the function being mapped:

```
map(a + b + 3, f, x, y)
```

$$f(3, x, y) + f(a, x, y) + f(b, x, y)$$

In the following example, we add 10 to each element of a list:

```
map([1, x, 2, y, 3, z], _plus, 10)
```

$$[11, x + 10, 12, y + 10, 13, z + 10]$$

Example 2

Like most other MuPAD functions, map does not modify its first argument, but returns a modified copy:

```
a := [0, PI/2, PI, 3*PI/2]:
map(a, sin)
```

$$[0, 1, 0, -1]$$

The list a still has its original value:

```
a
```

$$\left[0, \frac{\pi}{2}, \pi, \frac{3\,\pi}{2}\right]$$

Example 3

map does not decompose rational and complex numbers:

```
map(3/4, _plus, 1), map(3 + 4*I, _plus, 1)
```

$$\frac{7}{4}, 4 + 4\,i$$

map does not decompose strings:

```
map("MuPAD", text2expr)
```

MuPAD

`map` does not decompose polynomials:

```
map(poly(x^2 + x + 1), _plus, 1)
```

$$\mathrm{poly}(x^2 + x + 2, [x])$$

Use `mapcoeffs` to apply a function to all coefficients of a polynomial:

```
mapcoeffs(poly(x^2 + x + 1), _plus, 1)
```

$$\mathrm{poly}(2\,x^2 + 2\,x + 2, [x])$$

Example 4

The first argument is not flattened:

```
map((1, 2, 3), _plus, 2)
```

3, 4, 5

Example 5

Sometimes a MuPAD function returns a set or a list of big symbolic expressions containing mathematical constants etc. To get a better intuition about the result, you can map the function `float` to all elements, which often drastically reduces the size of the expressions:

```
solve(x^4 + x^2 + PI, x)
```

$$\left\{\sqrt{-\frac{\sqrt{1-4\,\pi}}{2}-\frac{1}{2}},\ \sqrt{\frac{\sqrt{1-4\,\pi}}{2}-\frac{1}{2}},\ -\sqrt{-\frac{\sqrt{1-4\,\pi}}{2}-\frac{1}{2}},\ -\sqrt{\frac{\sqrt{1-4\,\pi}}{2}-\frac{1}{2}}\right\}$$

```
map(%, float)
```

$$\{0.7976383425 - 1.065939457\,i,\ -0.7976383425 - 1.065939457\,i,\ 0.7976383425 + 1.065939457\,i,\ -0.7976383425 + 1.065939457\,i\}$$

Example 6

In the following example, we delete the values of all global identifiers in the current MuPAD session. The command `anames(All, User)` returns a set with the names of all user-defined global identifiers having a value. Mapping the function `_delete` to this set deletes the values of all these identifiers. Since the return value of `_delete` is the empty sequence `null()`, the result of the call is the empty set:

```
x := 3: y := 5: x + y
```

$$8$$

```
map(anames(All, User), _delete)
```

$$\varnothing$$

```
x + y
```

$$x + y$$

Example 7

It is possible to perform arbitrary actions with all elements of a data structure via a single `map` call. This works by passing an anonymous procedure as the second argument `f`. In the following example, we check that the fact "an integer $n \geq 2$ is prime if and only if $\varphi(n) = n - 1$", where φ denotes Euler's totient function, holds for all integer $2 \leq n < 10$. We do this by comparing the result of `isprime(n)` with the truth value of the equation $\varphi(n) = n - 1$ for all elements `n` of a list containing the integers between `2` and `9`:

```
map([2, 3, 4, 5, 6, 7, 8, 9],
    n -> bool(isprime(n) = bool(numlib::phi(n) = n - 1)))
```

$$[\mathrm{TRUE}, \mathrm{TRUE}, \mathrm{TRUE}, \mathrm{TRUE}, \mathrm{TRUE}, \mathrm{TRUE}, \mathrm{TRUE}, \mathrm{TRUE}]$$

Example 8

The result of `map` is not evaluated further. If desired, you must request evaluation explicitly by `eval`:

```
map(sin(5), float);
eval(%)
```

$$\sin(5.0)$$

$$-0.9589242747$$

```
delete a:
A := array(1..1, [a]);
a := 0:
map(A, sin);
map(A, eval@sin);
delete a:
```

$$(a)$$

$$(\sin(a))$$

$$(0)$$

Nevertheless, certain internal simplifications take place, such as the calculation of arithmetical operations with numerical arguments. The following call replaces `sqrt(2)` and `PI` by floating-point approximations, and the system automatically simplifies the resulting sum:

```
map(sin(5) + cos(5), float)
```

$$-0.6752620892$$

This internal simplification can be avoided by giving the option `Unsimplified`:

```
map(sin(5) + cos(5), float, Unsimplified)
```

$$0.2836621855 - 0.9589242747$$

Example 9

`map` applied to a table changes only the right sides (the entries) of each operand of the table. Assume the entries stand for net prices and the sales tax (16 percent in this case) must be added:

```
T := table(1 = 65, 2 = 28, 3 = 42):
map(T, _mult, 1.16)
```

1	75.4
2	32.48
3	48.72

Example 10

`map` can be overloaded for elements of library domains, if a slot`"map"` is defined. In this example `d` is a domain, its elements contains two integer numbers: an index and an entry (like a table). For nice input and printing elements of this domain the slots`"new"` and `"print"` are defined:

```
d := newDomain("d"):
d::new := () -> new(d, args()):
d::print := object -> _equal(extop(object)):
d(1, 65), d(2, 28), d(3, 42)
```

$$1 = 65,\ 2 = 28,\ 3 = 42$$

Without a slot `"map"` the function `f` will be applied to the domain element itself. Because the domain `d` has no slot `"_mult"`, the result is the symbolic `_mult` call:

```
map(d(1, 65), _mult, 1.16),
type(map(d(1, 65), _mult, 1.16))
```

$$1.16\ (1 = 65),\ \text{"\_mult"}$$

The slot `"map"` of this domain should map the given function only onto the second operand of a domain element. The domain `d` gets a slot `"map"` and `map` works properly (in the authors sense) with elements of this domain:

```
d::map := proc(obj, f)
```

```
          begin
            if args(0) > 2 then
              d(extop(obj, 1), f(extop(obj, 2), args(3..args(0))))
            else
              d(extop(obj, 1), f(extop(obj, 2)))
            end_if
          end_proc:
map(d(1, 65), _mult, 1.16),
map(d(2, 28), _mult, 1.16),
map(d(3, 42), _mult, 1.16)
```

$1 = 75.4,\ 2 = 32.48,\ 3 = 48.72$

Example 11

`map` does not work recursively. Suppose that we want to de-nest a nested list. We use `map` to apply the function `op`, which replaces a list by the sequence of its operands, to all entries of the list `l`. However, this only affects the entries at the first level:

```
l := [1, [2, [3]], [4, [5]]]:
map(l, op)
```

$[1, 2, [3], 4, [5]]$

Use `misc::maprec` to achieve the desired behavior:

```
[misc::maprec(l, {DOM_LIST} = op)]
```

$[1, 2, 3, 4, 5]$

Parameters

`object`

An arbitrary MuPAD object

`f`

A function

p_1, p_2, …

Any MuPAD objects accepted by `f` as additional parameters

Options

`Unsimplified`

The resulting expressions are not further simplified.

Return Values

Copy of `object` with `f` applied to all operands.

Overloaded By

object

See Also

MuPAD Functions

`eval` | `mapcoeffs` | `misc::maprec` | `op` | `select` | `split` | `subs` | `subsex` | `subsop` | `zip`

mapcoeffs

Apply a function to the coefficients of a polynomial

Syntax

```
mapcoeffs(p, F, <a1, a2, …>)
```

```
mapcoeffs(f, <vars>, F, <a1, a2, …>)
```

Description

`mapcoeffs(p, F, a1, a2, ...)` applies the function `F` to the polynomial `p` by replacing each coefficient `c` in `p` by `F(c, a1, a2, ...)`.

For a polynomial `p` of type `DOM_POLY` generated by `poly`, the function `F` must accept arguments from the coefficient ring of `p` and must produce corresponding results.

A polynomial expression `f` is first converted to a polynomial with the variables given by `vars`. If no variables are given, they are searched for in `f`. See `poly` about details of the conversion. `FAIL` is returned if `f` cannot be converted to a polynomial. After applying the function `F`, the result is converted to an expression.

`mapcoeffs` evaluates its arguments. Note, however, that polynomials of type `DOM_POLY` do not evaluate their coefficients for efficiency reasons. Cf. “Example 4” on page 1-1305.

Examples

Example 1

The function `sin` is mapped to the coefficients of a polynomial expression in the indeterminates `x` and `y`:

```
mapcoeffs(3*x^3 + x^2*y^2 + 2, sin)
```

$$\sin(3)\,x^3 + \sin(1)\,x^2\,y^2 + \sin(2)$$

The following call makes `mapcoeffs` regard this expression as a polynomial in `x`. Consequently, `y` is regarded as a parameter that becomes part of the coefficients:

```
mapcoeffs(3*x^3 + x^2*y^2 + 2, [x], sin)
```

$$\sin(3)\,x^3 + \sin(y^2)\,x^2 + \sin(2)$$

The system function `_plus` adds its arguments. In the following call, it is used to add 2 to all coefficients by providing this shift as an additional argument:

```
mapcoeffs(c1*x^3 + c2*x^2*y^2 + c3, [x, y], _plus, 2)
```

$$(c1+2)\,x^3 + (c2+2)\,x^2\,y^2 + c3 + 2$$

Example 2

The function `sin` is mapped to the coefficients of a polynomial in the indeterminates `x` and `y`:

```
mapcoeffs(poly(3*x^3 + x^2*y^2 + 2, [x, y]), sin)
```

$$\mathrm{poly}(\sin(3)\,x^3 + \sin(1)\,x^2\,y^2 + \sin(2), [x, y])$$

In the following call, the polynomial has the indeterminate `x`. Consequently, `y` is regarded as a parameter that becomes part of the coefficients:

```
mapcoeffs(poly(3*x^3 + x^2*y^2 + 2, [x]), sin)
```

$$\mathrm{poly}(\sin(3)\,x^3 + \sin(y^2)\,x^2 + \sin(2), [x])$$

A user-defined function is mapped to a polynomial:

```
F := (c, a1, a2) -> exp(c + a1 + a2):
mapcoeffs(poly(x^3 + c*x, [x]), F, a1, a2)
```

$$\mathrm{poly}\left(e^{a1+a2+1}\,x^3 + e^{a1+a2+c}\,x, [x]\right)$$

```
delete F:
```

Example 3

We consider a polynomial over the integers modulo 7:

```
p := poly(x^3 + 2*x*y, [x, y], Dom::IntegerMod(7)):
```

A function to be applied to the coefficients must produce values in the coefficient ring of the polynomial:

```
mapcoeffs(p, c -> c^2)
```

$$\text{poly}(x^3 + 4\,x\,y, [x, y], \text{Dom::IntegerMod}(7))$$

The following call maps a function which converts its argument to an integer modulo 3. Such a return value is not a valid coefficient of `p`:

```
mapcoeffs(p, c -> Dom::IntegerMod(3)(expr(c)))
```

FAIL

```
delete p:
```

Example 4

Note that polynomials of type `DOM_POLY` do not evaluate their arguments:

```
delete a, x: p := poly(a*x, [x]): a := PI: p
```

$$\text{poly}(a\,x, [x])$$

Evaluation can be enforced by the function `eval`:

```
mapcoeffs(p, eval)
```

$$\text{poly}(\pi\,x, [x])$$

We map the sine function to the coefficients of `p`. The polynomial does not evaluate its coefficient `sin(a)` to `0`:

```
mapcoeffs(p, sin)
```

$$\mathrm{poly}(\sin(a)\, x, [x])$$

The composition of `sin` and `eval` is mapped to the coefficients of the polynomial:

```
mapcoeffs(p, eval@sin)
```

$$\mathrm{poly}(0, [x])$$

```
delete p, a:
```

Parameters

`p`

A polynomial of type `DOM_POLY`

`F`

A procedure

`a1, a2, …`

Additional parameters for the function `F`

`f`

A polynomial expression

`vars`

A list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

Return Values

Polynomial of type `DOM_POLY`, or a polynomial expression, or `FAIL`.

Overloaded By

f, p

See Also

MuPAD Functions

`coeff` | `degree` | `degreevec` | `lcoeff` | `ldegree` | `lterm` | `map` | `monomials` | `nterms` | `nthcoeff` | `nthmonomial` | `nthterm` | `poly` | `tcoeff`

maprat

Apply a function to a rationalized expression

Syntax

```
maprat(object, f, options)
```

Description

As a first step, `maprat(object, f, options)` calls `rationalize(object, options)`, which generates a rational expression. The `maprat` function uses the expression returned by `rationalize` as an input to the function `f`. As a second step, `maprat` replaces all variables generated by `rationalize` with the original subexpressions in `object`.

See the `rationalize` help page for details.

Examples

Example 1

Find the greatest common divisor (the `gcd` function) for the following two rationalized expressions. The first argument of `maprat` is a sequence of the two expressions `p`, `q`, which `gcd` takes as two parameters. Note the brackets around the sequence `p, q`:

```
p := (x - sqrt(2))*(x^2 + sqrt(3)*x - 1):
q := (x - sqrt(2))*(x - sqrt(3)):
maprat((p, q), gcd)
```

$$\sqrt{2} - x$$

Example 2

The `maprat` function accepts the same options as the `rationalize` function. For example, find the least common multiple (the `lcm` function) for the following two

rationalized expressions. Use the `FindRelations` option to detect trigonometric relations:

```
p := tan(x)^2 + 1/cos(x)^2:
q := 1/sin(x)^4 + cot(x)^4:
maprat((p, q), lcm, FindRelations = ["sin"])
```

$$\left(\tan\left(\frac{x}{2}\right)^4 + 6\tan\left(\frac{x}{2}\right)^2 + 1\right)\left(\tan\left(\frac{x}{2}\right)^8 + 6\tan\left(\frac{x}{2}\right)^4 + 1\right)$$

Without this option, the result is:

```
p := tan(x)^2 + 1/cos(x)^2:
q := 1/sin(x)^4 + cot(x)^4:
maprat((p, q), lcm)
```

$$(\cot(x)^4\sin(x)^4 + 1)(\cos(x)^2\tan(x)^2 + 1)$$

Free the variables for further calculations:

```
delete p, q:
```

Parameters

`object`

An arithmetical expression, or a sequence, or a set, or a list of such expressions

`f`

A procedure or a functional expression

Options

`ApproximateFloats`

Approximate floating-point numbers by rational numbers.

`FindRelations`

Detect algebraic dependencies for subexpressions of specified types.

`DescendInto`

If the original expression contains subexpressions, rationalize the specified types of subexpressions.

`ReplaceHardToEval`

Replace all subexpressions with limits, sums, and integrals by variables.

`ReplaceTypes`

Replace all subexpressions of the specified types by variables.

`StopOn`

Do not rationalize specified types of subexpressions.

`StopOnConstants`

Do not rationalize numbers, strings, Boolean constants, `NIL`, `FAIL`, PI, EULER, and CATALAN in the set `Type::ConstantIdents`.

Return Values

Object returned by the function `f`.

See Also

MuPAD Functions
`map` | `rationalize`

matrix

Create a matrix or a vector

Syntax

```
matrix(Array)
matrix(List)
matrix(ListOfRows)
matrix(Matrix)
matrix(m, n)
matrix(m, n, Array)
matrix(m, n, List)
matrix(m, n, ListOfRows)
matrix(m, n, Table)
matrix(m, n, [(i1, j1) = value1, (i2, j2) = value2, …])
matrix(m, n, f)
matrix(m, n, List, Diagonal)
matrix(m, n, g, Diagonal)
matrix(m, n, List, Banded)
matrix(1, n, [j1 = value1, j2 = value2, …])
matrix(m, 1, [i1 = value1, i2 = value2, …])
```

Description

`matrix(m, n, [[a11, a12, ...], [a21, a22, ...], ...])` returns an $m \times n$ matrix of the domain type `Dom::Matrix()`.

`matrix(m, n, [a11, a12, ..., a21, a22, ..., a.m.n])` returns an $m \times n$ matrix of the domain type `Dom::Matrix()`.

`matrix(m, 1, [a1, a2, ...])` returns an $m \times 1$ column vector of the domain type `Dom::Matrix()`.

`matrix(1, n, [a1, a2, ...])` returns an $1 \times n$ row vector of the domain type `Dom::Matrix()`.

`matrix` is equivalent to `Dom::Matrix()`.

`matrix` creates matrices and vectors. A column vector is represented as an $m \times 1$ matrix. A row vector is represented as a $1 \times n$ matrix.

Matrix and vector components must be arithmetical expressions (numbers and/or symbolic expressions). If matrices over special component rings are desired, use the domain constructor `Dom::Matrix` with a suitable component ring.

Arithmetical operations with matrices can be performed by using the standard arithmetical operators of MuPAD.

E.g., if `A` and `B` are two matrices defined by `matrix`, then `A + B` computes the sum and `A * B` computes the product of the two matrices, provided that the dimensions are appropriate.

Similarly, `A^(-1)` or `1/A` computes the inverse of a square matrix `A` if it can be inverted. Otherwise, `FAIL` is returned.

Cf. “Example 1” on page 1-1315.

Many system functions accept matrices as input, such as `map`, `subs`, `has`, `zip`, `conjugate`, `norm` or `exp`. Cf. “Example 4” on page 1-1318.

Most of the functions in the MuPAD linear algebra package linalg work with matrices. For example, the command `linalg::gaussJordan(A)` performs Gauss-Jordan elimination on `A` to transform `A` to its reduced row echelon form.

For numerical matrix computations, the corresponding functions of the `numeric` package accept matrices.

Matrix components can be extracted by the usual index operator `[ ]`, which also works for lists, arrays, and tables. The call `A[i, j]` extracts the matrix component in the i-th row and the j-th column.

Assignments to matrix components are performed similarly. The call `A[i, j] := c` replaces the matrix component in the *i*-th row and the *j*-th column of *A* by `c`.

If one of the indices is not in its valid range, an error message is issued.

The index operator also extracts submatrices. The call `A[r1..r2, c1..c2]` creates the submatrix of *A* comprising the rows with the indices r_1, $r_1 + 1$, …, r_2 and the columns with the indices c_1, $c_1 + 1$, …, c_2 of *A*.

See “Example 3” on page 1-1317 and “Example 5” on page 1-1320.

`matrix(Array)` or `matrix(Matrix)` create a new matrix with the same dimension and the components of `Array` or `Matrix`, respectively. The array must not contain any uninitialized entries. If `Array` is one-dimensional, the result is a column vector. Cf. “Example 8” on page 1-1324.

`matrix(List)` creates an $m \times 1$ column vector with components taken from the non-empty list, where *m* is the number of entries of `List`. Cf. “Example 5” on page 1-1320.

`matrix(ListOfRows)` creates an $m \times n$ matrix with components taken from the nested list `ListOfRows`, where *m* is the number of inner lists of `ListOfRows`, and *n* is the maximal number of elements of an inner list. Each inner list corresponds to a row of the matrix. Both *m* and *n* must be non-zero.

If a row has less than *n* entries, the remaining entries in the corresponding row of the matrix are regarded as zero. Cf. “Example 7” on page 1-1322.

The call `matrix(m, n)` returns the $m \times n$ zero matrix.

The call `matrix(m, n, Array)` creates an $m \times n$ matrix with components taken from `Array`, which must be an array or an hfarray. `Array` must have *m n* operands. The first *m* operands define the first row, the next *m* operands define the second row, etc. The formatting of the array is irrelevant. E.g., any array with 6 elements can be used to create matrices of dimension 1×6, or 2×3, or 3×2, or 6×1.

`matrix(m, n, List)` creates an $m \times n$ matrix with components taken row after row from the non-empty list. The list must contain *m n* entries. Cf. “Example 7” on page 1-1322.

`matrix(m, n, ListOfRows)` creates an $m \times n$ matrix with components taken from the list `ListOfRows`.

If $m \geq 2$ and $n \geq 2$, then `ListOfRows` must consist of at most `m` inner lists, each having at most `n` entries. The inner lists correspond to the rows of the returned matrix.

If a row has less than `n` entries, the remaining components of the corresponding row of the matrix are regarded as zero. If there are less than `m` rows, the remaining lower rows of the matrix are filled with zeroes. Cf. "Example 7" on page 1-1322.

`matrix(m,n,Table)` creates an $m \times n$ matrix with components taken from the table `Table`. The table entries `Table[i,j]` with positive integer values of `i` and `j` define the corresponding entries of the matrix. Zero entries need not be specified in the table. This way, sparse table input can be used to create the matrix.

For large sparse matrices, the fastest way of creation is the generation of an empty table that is filled by indexed assignments and then passed to `matrix`. Alternatively, one may first create an empty sparse matrix via `matrix(m, n)` and then fill in the non-zero entries via indexed assignments. Note that the indexed assignment to a matrix is somewhat slower than the indexed assignment to a table.

`matrix(m, n, [(i1, j1) = value1, (i2, j2) = value2, ...])` is a further way to create a matrix specifying only the non-zero entries `A[i1, j1] = value1`, `A[i2, j2] = value2` etc. The ordering of the entries in the input list is irrelevant.

`matrix(m, n, f)` returns the matrix whose (i, j)-th component is the return value of the function call `f(i,j)`. The row index i runs from 1 to m and the column index j from 1 to n. Cf. "Example 9" on page 1-1325.

`matrix(m, 1, Array)` returns the $m \times 1$ column vector with components taken from `Array`. The `array` or `hfarray` `Array` must have `m` entries.

`matrix(m, 1, List)` returns the $m \times 1$ column vector with components taken from `List`. The list `List` must have no more than `m` entries. If there are fewer entries, the remaining vector components are regarded as zero. Cf. "Example 5" on page 1-1320.

`matrix(m, 1, Table)` returns the $m \times 1$ column vector with components taken from `Table`. The table `Table` must have no more than `m` entries. If there are fewer entries, the remaining vector components are regarded as zero. Cf. "Example 6" on page 1-1322.

`matrix(m, 1, [i1 = value1, i2 = value2, ...])` provides a way to create a sparse column vector specifying only the non-zero entries `A[i1] = value1`, `A[i2] = value2` etc. The ordering of the entries in the input list is irrelevant.

`matrix(1, n, Array)` returns the $1 \times n$ row vector with components taken from `Array`. The `array` or `hfarray` `Array` must have `n` entries.

`matrix(1, n, List)` returns the $1 \times n$ row vector with components taken from `List`. The list `List` must not have more than `n` entries. If there are fewer entries, the remaining vector components are regarded as zero. Cf. “Example 5” on page 1-1320.

`matrix(1, n, Table)` returns the $1 \times n$ row vector with components taken from `Table`. The table `Table` must not have more than `n` entries. If there are fewer entries, the remaining vector components are regarded as zero. Cf. “Example 6” on page 1-1322.

`matrix(1, n, [j1 = value1, j2 = value2, ...])` provides a way to create a sparse row vector specifying only the non-zero entries `A[j1] = value1`, `A[j2] = value2` etc. The ordering of the entries in the input list is irrelevant.

Note: The number of rows and columns, respectively, of a matrix must be less than 2^{31}.

Note: The components of a matrix are no longer evaluated after the creation of the matrix, i.e., if they contain free identifiers they will not be replaced by their values.

Examples

Example 1

We create a 2×2 matrix by passing a list of two rows to `matrix`, where each row is a list of two elements:

```
A := matrix([[1, 5], [2, 3]])
```

$$\begin{pmatrix} 1 & 5 \\ 2 & 3 \end{pmatrix}$$

In the same way, we generate the following 2 ×3 matrix:

```
B := matrix([[-1, 5/2, 3], [1/3, 0, 2/5]])
```

$$\begin{pmatrix} -1 & \frac{5}{2} & 3 \\ \frac{1}{3} & 0 & \frac{2}{5} \end{pmatrix}$$

We can do matrix arithmetic using the standard arithmetical operators of MuPAD. For example, the matrix product $A\,B$, the fourth power of A, and the scalar multiplication of A by $\frac{1}{3}$ are given by:

```
A * B, A^4, 1/3 * A
```

$$\begin{pmatrix} \frac{2}{3} & \frac{5}{2} & 5 \\ -1 & 5 & \frac{36}{5} \end{pmatrix} \cdot \begin{pmatrix} 281 & 600 \\ 240 & 521 \end{pmatrix} \cdot \begin{pmatrix} \frac{1}{3} & \frac{5}{3} \\ \frac{2}{3} & 1 \end{pmatrix}$$

Since the dimensions of the matrices A and B differ, the sum of A and B is not defined and MuPAD returns an error message:

```
A + B
```

```
Error: The dimensions do not match. [(Dom::Matrix(Dom::ExpressionField()))::_plus]
```

To compute the inverse of A, enter:

```
1/A
```

$$\begin{pmatrix} -\frac{3}{7} & \frac{5}{7} \\ \frac{2}{7} & -\frac{1}{7} \end{pmatrix}$$

If a matrix is not invertible, the result of this operation is `FAIL`:

```
C := matrix([[2, 0], [0, 0]])
```

$$\begin{pmatrix} 2 & 0 \\ 0 & 0 \end{pmatrix}$$

```
C^(-1)
```

FAIL

```
delete A, B, C:
```

Example 2

In addition to standard matrix arithmetic, the library linalg offers numerous functions handling matrices. For example, the function `linalg::rank` determines the rank of a matrix:

```
A := matrix([[1, 5], [2, 3]])
```

$$\begin{pmatrix} 1 & 5 \\ 2 & 3 \end{pmatrix}$$

```
linalg::rank(A)
```

$$2$$

The function `linalg::eigenvectors` computes the eigenvalues and the eigenvectors of A:

```
linalg::eigenvectors(A)
```

$$\left[\left[2-\sqrt{11},\ 1,\ \left[\begin{pmatrix} -\frac{\sqrt{11}}{2}-\frac{1}{2} \\ 1 \end{pmatrix}\right]\right],\ \left[\sqrt{11}+2,\ 1,\ \left[\begin{pmatrix} \frac{\sqrt{11}}{2}-\frac{1}{2} \\ 1 \end{pmatrix}\right]\right]\right]$$

To determine the dimension of a matrix, use the function `linalg::matdim`:

```
linalg::matdim(A)
```

$$[2,\ 2]$$

The result is a list of two positive integers, the row and column number of the matrix.

Use `info(linalg)` to obtain a list of available functions, or enter `?linalg` for details about this library.

```
delete A:
```

Example 3

Matrix entries can be accessed with the index operator `[ ]`:

```
A := matrix([[1, 2, 3, 4], [2, 0, 4, 1], [-1, 0, 5, 2]])
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 0 & 4 & 1 \\ -1 & 0 & 5 & 2 \end{pmatrix}$$

```
A[2, 1] * A[1, 2] - A[3, 1] * A[1, 3]
```

$$7$$

You can redefine a matrix entry by assigning a value to it:

```
A[1, 2] := a^2: A
```

$$\begin{pmatrix} 1 & a^2 & 3 & 4 \\ 2 & 0 & 4 & 1 \\ -1 & 0 & 5 & 2 \end{pmatrix}$$

The index operator can also be used to extract submatrices. The following call creates a copy of the submatrix of *A* comprising the second and the third row and the first three columns of *A*:

```
A[2..3, 1..3]
```

$$\begin{pmatrix} 2 & 0 & 4 \\ -1 & 0 & 5 \end{pmatrix}$$

The index operator does *not* allow to replace a submatrix of a given matrix by another matrix. Use `linalg::substitute` to achieve this.

```
delete A:
```

Example 4

Some system functions can be applied to matrices. For example, if you have a matrix with symbolic entries and want to have all entries in expanded form, simply apply the function `expand`:

```
delete a, b:
```

```
A := matrix([
  [(a - b)^2, a^2 + b^2],
  [a^2 + b^2, (a - b)*(a + b)]
])
```

$$\begin{pmatrix} (a-b)^2 & a^2+b^2 \\ a^2+b^2 & (a+b)\,(a-b) \end{pmatrix}$$

```
expand(A)
```

$$\begin{pmatrix} a^2-2\,a\,b+b^2 & a^2+b^2 \\ a^2+b^2 & a^2-b^2 \end{pmatrix}$$

You can differentiate all matrix components with respect to some indeterminate:

```
diff(A, a)
```

$$\begin{pmatrix} 2\,a-2\,b & 2\,a \\ 2\,a & 2\,a \end{pmatrix}$$

The following command evaluates all matrix components at a given point:

```
subs(A, a = 1, b = -1)
```

$$\begin{pmatrix} 4 & 2 \\ 2 & 0 \end{pmatrix}$$

Note that the function `subs` does not evaluate the result of the substitution. For example, we define the following matrix:

```
A := matrix([[sin(x), x], [x, cos(x)]])
```

$$\begin{pmatrix} \sin(x) & x \\ x & \cos(x) \end{pmatrix}$$

Then we substitute $x = 0$ in each matrix component:

```
B := subs(A, x = 0)
```

$$\begin{pmatrix} \sin(0) & 0 \\ 0 & \cos(0) \end{pmatrix}$$

You see that the matrix components are not evaluated completely. For example, if you enter `sin(0)` directly, it evaluates to zero.

The function `eval` can be used to evaluate the result of the function `subs`. However, `eval` does not operate on matrices directly, and you must use the function `map` to apply the function `eval` to each matrix component:

```
map(B, eval)
```

$$\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

The function `zip` can be applied to matrices. The following call combines two matrices A and B by dividing each component of A by the corresponding component of B:

```
A := matrix([[4, 2], [9, 3]]):
B := matrix([[2, 1], [3, -1]]):
A, B, zip(A, B, `/`)
```

$$\begin{pmatrix} 4 & 2 \\ 9 & 3 \end{pmatrix}, \begin{pmatrix} 2 & 1 \\ 3 & -1 \end{pmatrix}, \begin{pmatrix} 2 & 2 \\ 3 & -3 \end{pmatrix}$$

```
delete A, B:
```

Example 5

A vector is either an $m \times 1$ matrix (a column vector) or a $1 \times n$ matrix (a row vector). To create a vector with `matrix`, pass the dimension of the vector and a list of vector components as argument to `matrix`:

```
row_vector := matrix(1, 3, [1, 2, 3]);
column_vector := matrix(3, 1, [1, 2, 3])
```

$$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

If the only argument of `matrix` is a non-nested list or a one-dimensional array, the result is a column vector:

```
matrix([1, 2, 3])
```

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

For a row vector `r`, the calls `r[1, i]` and `r[i]` both return the *i*-th vector component of `r`. Similarly, for a column vector `c`, the calls `c[i, 1]` and `c[i]` both return the *i*-th vector component of `c`.

We extract the second component of the vectors defined above:

```
row_vector[2] = row_vector[1, 2],
column_vector[2] = column_vector[2, 1]
```

$$2 = 2,\ 2 = 2$$

Use the function `linalg::vecdim` to determine the number of components of a vector:

```
linalg::vecdim(row_vector), linalg::vecdim(column_vector)
```

$$3,\ 3$$

The number of components of a vector can also be determined directly by the call `nops(vector)`.

The dimension of a vector can be determined as described above in the case of matrices:

```
linalg::matdim(row_vector),
linalg::matdim(column_vector)
```

$$[1,\ 3],\ [3,\ 1]$$

See the linalg package for functions working with vectors, and the help page of `norm` for computing vector norms.

```
delete row_vector, column_vector:
```

Example 6

A vector is either an $m \times 1$ matrix (a column vector) or a $1 \times n$ matrix (a row vector). To create a vector with `matrix`, one may also pass the dimension of the vector and a table of vector components as argument to `matrix`:

```
delete v1, v2, t1, t2:
t1 := table():
t1[1,1] := 1:
t1[1,2] := 2:
t1[1,3] := 3:
v1 := matrix(1, 3, t1);
```

$$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

```
t2 := table():
t2[1,1] := 1:
t2[2,1] := 2:
t2[3,1] := 3:
v2 := matrix(3, 1, t2);
```

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

All functions applied to the vectors in the previous example (see above) can can also be used on these vectors.

```
delete t1, t2, v1, v2:
```

Example 7

In the following examples, we illustrate various calls of `matrix` as described above. We start by passing a nested list to `matrix`, where each inner list corresponds to a row of the matrix:

```
matrix([[1, 2], [2]])
```

$$\begin{pmatrix} 1 & 2 \\ 2 & 0 \end{pmatrix}$$

The number of rows of the created matrix is the number of inner lists, namely $m = 2$. The number of columns is determined by the maximal number of entries of an inner list. In the example above, the first list is the longest one, and hence $n = 2$. The second list has only one element and, therefore, the second entry in the second row of the returned matrix was set to zero.

In the following call, we use the same nested list, but in addition pass two dimension parameters to create a 4×4 matrix:

```
matrix(4, 4, [[1, 2], [2]])
```

$$\begin{pmatrix} 1 & 2 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

In this case, the dimension of the matrix is given by the dimension parameters. As before, missing entries in an inner list correspond to zero, and in addition missing rows are treated as zero rows.

If the dimension $m \times n$ of the matrix is stated explicitly, the entries may also be specified by a plain list with $m\,n$ elements. The matrix is filled with these elements row by row:

```
matrix(2, 3, [1, 2, 3, 4, 5, 6])
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

```
matrix(3, 2, [1, 2, 3, 4, 5, 6])
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

Example 8

A one- or two-dimensional array of arithmetical expressions, such as:

```
a := array(1..3, 2..4,
  [[1, 1/3, 0], [-2, 3/5, 1/2], [-3/2, 0, -1]]
)
```

$$\begin{pmatrix} 1 & \frac{1}{3} & 0 \\ -2 & \frac{3}{5} & \frac{1}{2} \\ -\frac{3}{2} & 0 & -1 \end{pmatrix}$$

can be converted into a matrix as follows:

```
A := matrix(a)
```

$$\begin{pmatrix} 1 & \frac{1}{3} & 0 \\ -2 & \frac{3}{5} & \frac{1}{2} \\ -\frac{3}{2} & 0 & -1 \end{pmatrix}$$

Arrays serve, for example, as an efficient structured data type for programming. However, arrays do not have any algebraic meaning, and no mathematical operations are defined for them. If you convert an array into a matrix, you can use the full functionality defined for matrices as described above. For example, let us compute the matrix $2A - A^2$ and the Frobenius norm of A:

```
2*A - A^2, norm(A, Frobenius)
```

$$\begin{pmatrix} \frac{5}{3} & \frac{2}{15} & -\frac{1}{6} \\ -\frac{1}{20} & \frac{113}{75} & \frac{6}{5} \\ -3 & \frac{1}{2} & -3 \end{pmatrix}, \frac{\sqrt{2}\,\sqrt{4037}}{30}$$

Note that an array may contain uninitialized entries:

```
b := array(1..4): b[1] := 2: b[4] := 0: b
```

$$(2\ \text{NIL}\ \text{NIL}\ 0)$$

`matrix` cannot handle arrays that have uninitialized entries, and responds with an error message:

```
matrix(b)
```

```
Error: Cannot define a matrix over 'Dom::ExpressionField()'. [(Dom::Matrix(Dom::Express
```

We initialize the remaining entries of the array `b` and convert it into a matrix, or more precisely, into a column vector:

```
b[2] := 0: b[3] := -1: matrix(b)
```

$$\begin{pmatrix} 2 \\ 0 \\ -1 \\ 0 \end{pmatrix}$$

```
delete a, A, b:
```

Example 9

We show how to create a matrix whose components are defined by a function of the row and the column index. The entry in the i-th row and the j-th column of a Hilbert matrix (see also `linalg::hilbert`) is $\frac{1}{(i+j-1)}$. Thus the following command creates a 2×2 Hilbert matrix:

```
matrix(2, 2, (i, j) -> 1/(i + j - 1))
```

$$\begin{pmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{3} \end{pmatrix}$$

The following two calls produce different results. In the first call, `x` is regarded as an unknown function, while it is a constant in the second call:

```
delete x:
matrix(2, 2, x), matrix(2, 2, (i, j) -> x)
```

$$\begin{pmatrix} x(1,1) & x(1,2) \\ x(2,1) & x(2,2) \end{pmatrix}, \begin{pmatrix} x & x \\ x & x \end{pmatrix}$$

Example 10

Diagonal matrices can be created by passing the option `Diagonal` and a list of diagonal entries:

```
matrix(3, 4, [1, 2, 3], Diagonal)
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \end{pmatrix}$$

One can generate the 3×3 identity matrix as follows:

```
matrix::identity(3)
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Here are alternative ways to create this matrix:

```
matrix(3, 3, [1 $ 3], Diagonal)
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Equivalently, you can use a function of one argument:

```
matrix(3, 3, i -> 1, Diagonal)
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Since the integer 1 also represents a constant function, the following shorter call creates the same matrix:

```
matrix(3, 3, 1, Diagonal)
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

To demonstrate the use of tables for creating (sparse) matrices we can also create the identity matrix above by the lines:

```
t := table(): t[1, 1] := 1: t[2, 2] := 1: t[3, 3] := 1:
matrix(3, 3, t)
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

```
delete t:
```

Example 11

Banded Toeplitz matrices can be created with the option `Banded`. The following command creates a tri-diagonal matrix with constant bands:

```
matrix(4, 4, [-1, 2, -1], Banded)
```

$$\begin{pmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{pmatrix}$$

Example 12

Matrices can also be created by using a `table`:

```
t := table():
```

```
t[1, 2] := 12:
t[3, 1] := 31:
t[3, 2] := 32:
t
```

1, 2	12
3, 1	31
3, 2	32

The missing table entries correspond to empty matrix entries:

```
A := matrix(4, 6, t)
```

$$\begin{pmatrix} 0 & 12 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 31 & 32 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

By using tables, one can easily create large (sparse) matrices without being forced to define all zero entries of the matrix. Note that this is a great advantage over using arrays where every component has to be initialized before.

```
delete t, A:
```

Example 13

The method `"doprint"` of `Dom::Matrix()` prints only the non-zero components of a sparse matrix:

```
A := matrix(4, 6):
A[1, 2]:= 12: A[3, 1]:= 31: A[3, 2]:= 32:
print(A::dom::doprint(A)):
```

$$\begin{pmatrix} 0 & 12 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 31 & 32 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

```
delete A:
```

Parameters

`Array`

A one- or two-dimensional array or hfarray

`List`

A list of arithmetical expressions

`ListOfRows`

A nested list of rows, each row being a list of arithmetical expressions

`Matrix`

A matrix, i.e., an object of a data type of category `Cat::Matrix`

`Table`

A table of matrix components

`m`

The number of rows: a positive integer

`n`

The number of columns: a positive integer

`f`

A function or a functional expression of two arguments

`g`

A function or a functional expression of one argument

i_1, i_2, …

Row indices: integers between 1 and m

j_1, j_2, …

Column indices: integers between 1 and m

value$_1$, value$_2$, …

Matrix entries: arithmetical expressions

Options

Diagonal

Create a diagonal matrix

With this option, diagonal matrices can be created with diagonal elements taken from a list, or computed by a function or a functional expression.

`matrix(m, n, List, Diagonal)` creates the $m \times n$ diagonal matrix whose diagonal elements are the entries of `List`. Cf. "Example 10" on page 1-1326.

`List` must have no more than $min(m, n)$ entries. If it has fewer elements, the remaining diagonal elements are regarded as zero.

`matrix(m, n, g, Diagonal)` returns the sparse matrix whose i-th diagonal element is `g(i, i)`, where the index i runs from 1 to $min(m, n)$. Cf. "Example 10" on page 1-1326.

Banded

Create a banded Toeplitz matrix

A *banded matrix* has zero entries outside the main diagonal and some of the adjacent sub- and superdiagonals.

`matrix(m, n, List, Banded)` creates an $m \times n$ banded Toeplitz matrix with the elements of `List` as entries. The number of entries of `List` must be odd, say $2h + 1$, where h must not exceed `n`. The bandwidth of the resulting matrix is at most h.

All elements of the main diagonal of the created matrix are initialized with the middle element of `List`. All elements of the i-th subdiagonal are initialized with the $(h + 1 - i)$-th element of `List`. All elements of the i-th superdiagonal are initialized with the $(h + 1 + i)$-th element of `List`. All entries on the remaining sub- and superdiagonals are regarded as zero.

Cf. "Example 11" on page 1-1327.

Return Values

Matrix of the domain type `Dom::Matrix()`.

See Also

MuPAD Domains
`Dom::DenseMatrix` | `Dom::Matrix` | `DOM_ARRAY` | `DOM_HFARRAY`

MuPAD Functions
`array` | `densematrix` | `hfarray`

max

Maximum of numbers

Syntax

```
max(x1, x2, , …)
max({x1, x2, …})
max([x1, x2, …])
max(A)
```

Description

`max(x1, x2, ...)` returns the maximum of the numbers x_1, x_2, ….

If the arguments of `max` are either integers, rational numbers, or floating-point numbers, then `max` returns the numerical maximum of these arguments.

Exact numerical expressions such as `PI + sqrt(2)` etc. are internally converted to floating-point intervals using the current value of `DIGITS`. After comparison, the exact expression is restored in the return value. If the current value of `DIGITS` does not suffice to determine the maximum of several expressions, a symbolic call of `max` is returned. Cf. “Example 2” on page 1-1334.

The call `max()` is illegal and leads to an error message. If there is only one argument `x1`, then `max` evaluates `x1` and returns it. Cf. “Example 3” on page 1-1335.

If one of the arguments is `infinity`, then `max` returns `infinity`. If an argument is `-infinity`, then it is removed from the argument list. Cf. “Example 4” on page 1-1335.

`max` returns an error when one of its arguments is a complex number or a floating point interval with non-zero imaginary part. Cf. “Example 3” on page 1-1335.

If one of the arguments is not a number, then a symbolic `max` call with the maximum of the numerical arguments and the remaining evaluated arguments may be returned. Cf. “Example 1” on page 1-1333.

Nested `max` calls with symbolic arguments are rewritten as a single `max` call, i.e., they are flattened. Cf. “Example 5” on page 1-1335.

`max` reacts to a very limited set of properties of identifiers set via `assume`. Use `simplify` to handle more general assumptions. Cf. “Example 5” on page 1-1335.

Environment Interactions

When called with exact numerical expressions such as `PI`, `sqrt(2)` etc., the function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

`max` computes the maximum of integers, rational numbers, and floating-point values:

```
max(-3/2, 7, 1.4)
```

$$7$$

Floating point intervals are interpreted as “any number within this range” and may thus cause symbolic `max` calls to be returned:

```
max(2...3 union 6...7, 4)
```

$$\max(2.0 \ldots 3.0 \cup 6.0 \ldots 7.0, 4)$$

```
max(2...3, 6...7, 4)
```

$$6.0 \ldots 7.0$$

```
max(2...3, PI)
```

$$\pi$$

If the argument list contains symbolic expressions, then a symbolic `max` call may be returned:

```
delete b:
max(-4, b + 2, 1, 3)
```

$$\max(b+2, 3)$$

In the following two examples, `max` is able to determine the maximum despite getting symbolic arguments (contrast this with <):

```
max(sqrt(2), 1)
```

$$\sqrt{2}$$

```
assume(x > 0):
max(exp(x), exp(-x))
```

$$e^x$$

Example 2

The following rational number `pi` approximates π to about 20 decimal places:

```
pi := 314159265358979323846/10^20:
```

With the default value `DIGITS = 10`, the function `max` cannot distinguish between `PI` and `pi` via floating-point approximations:

```
max(pi, PI)
```

$$\max\left(\frac{15707963267948966192}{5000000000000000000}, \pi\right)$$

With an increased value of `DIGITS`, the floating-point interval approximation of `PI` considered by `max` allows to decide that `PI` is larger than `pi`:

```
DIGITS := 20:
max(pi, PI)
```

π

```
delete pi, DIGITS:
```

Example 3

`max` with one argument returns the evaluated argument:

```
delete a:
max(a), max(sin(2*PI)), max(2)
```

$a, 0, 2$

Complex numbers lead to an error message:

```
max(0, 1, I)
```

```
Error: The argument is invalid. [max]
```

Example 4

`infinity` is always the maximum of arbitrary arguments:

```
delete x:
max(100000000000, infinity, x)
```

∞

`-infinity` is removed from the argument list:

```
max(100000000000, -infinity, x)
```

$\max(100000000000, x)$

Example 5

`max` reacts only to very few properties of identifiers set via `assume`:

```
delete a, b, c:
assume(a > 0 and b > a and c > b):
max(a, max(b, c), 0)
```

$$\max(a, b, c)$$

An application of `simplify` yields the desired result:

```
simplify(%)
```

$$c$$

Parameters

`x1, x2, …`

Arbitrary MuPAD objects

`A`

An array of domain type `DOM_HFARRAY` with real entries

Return Values

One of the arguments, a floating-point number, or a symbolic `max` call.

Overloaded By

See Also

MuPAD Functions

`_leequal` | `_less` | `min` | `sort` | `sysorder`

MAXDEPTH

Prevent infinite recursion during procedure calls

Description

The environment variable MAXDEPTH determines the maximal recursion depth of nested procedure calls. When this recursion depth is reached, an error occurs.

Possible values: Positive integer; the maximum value depends on the operating system, see below.

The purpose of MAXDEPTH is to provide a heuristic for recognizing infinite recursion with respect to procedure calls, like in `p := x -> p(x): p(0)`. If, in this example, the recursion depth would not be limited, then the procedure p would call itself recursively infinitely often, and the system would "hang".

If during the evaluation of an object the recursion depth MAXDEPTH is reached, then the computation is aborted with an error.

Similarly, the environment variable MAXLEVEL provides a heuristic for recognizing infinite recursion with respect to the substitution of values for identifiers; see the corresponding help page for details and examples.

The default value of MAXDEPTH is 500; MAXDEPTH has this value after starting or resetting the system via `reset`. Also the command `delete MAXDEPTH` restores the default value.

MAXDEPTH is a global variable. Use the statement `save MAXDEPTH` in a procedure to confine any changes to MAXDEPTH to this procedure.

The maximum value of MAXDEPTH depends on the operating system. Under Windows it is $2^{11} = 2048$. Under UNIX operating systems the maximum value depends on the maximum size of the C-stack. With a default stack size of `8 MB` the value is 2048, too; with a bigger stack size it can be bigger (in a bash the stack size can be set with `ulimit -s`).

Examples

Example 1

Evaluation of objects defined by an infinite recursion produces an error:

```
p := proc() begin p() end_proc: p()
```

```
Error: Recursive definition: the maximal depth for nested procedure calls is reached.
  Evaluating: p
```

This also works for mutually recursive definitions:

```
p := proc(x) begin q(x + 1)^2 end_proc:
q := proc(y) begin p(x) + 2 end_proc:
p(0)
```

```
Error: Recursive definition: the maximal depth for nested procedure calls is reached.
  Evaluating: p
```

Example 2

If the maximal recursion depth is reached, then this does not necessarily mean that infinite recursion is involved. The following recursive procedure computes the factorial of a nonnegative integer. If we set the maximal recursion depth to a smaller value than necessary to compute $5!$, then an error occurs:

```
factorial := proc(n) begin
  if n = 0 then 1
  else n*factorial(n - 1)
  end_if
end_proc:
MAXDEPTH := 4: factorial(5)
```

```
Error: Recursive definition: the maximal depth for nested procedure calls is reached.
  Evaluating: factorial
```

If we set `MAXDEPTH` to 5, then the recursion depth is big enough for computing $5!$. The command `delete MAXDEPTH` resets `MAXDEPTH` to its default value `500`:

```
MAXDEPTH := 5: factorial(5); delete MAXDEPTH:
```

```
120
```

See Also

MuPAD Functions
`eval` | `freeze` | `LEVEL` | `level` | `MAXLEVEL` | `proc`

MAXEFFORT

Maximum amount of work to spend on the computation

Description

The environment variable MAXEFFORT determines the amount of effort allowed for heuristical parts of a computation, measured in "working units". The default value is MAXEFFORT = 1000000.

Possible values: Non-negative floating-point number; or `infinity`.

MAXEFFORT determines the maximum number of "working units" that may be spent on internal heuristics.

One working unit roughly corresponds to 1000 evaluation steps done by an average kernel function.

Whatever MAXEFFORT is set to, every MuPAD function returns a correct though possibly unsimplified result; in particular, some functions may return unevaluated. MAXEFFORT determines the amount of additional time spent on obtaining a better or more simplified result; a value of `infinity` means that all built-in heuristics are really tried, a value of `0` means that all heuristics that might take considerable effort are left out.

A function whose result is uniquely specified has no way to react to MAXEFFORT.

Other functions carry out, in any event, all computations necessary to obtain some correct result; MAXEFFORT only determines the time available for improving that result. In case of functions that may return unevaluated immediately (e.g., `solve` or `int`), or may return their input immediately (as, e.g., `simplify`), or may answer a question by UNKNOWN immediately (as, e.g., `is`), all of their time consumption is counted to be spent on heuristics (purely heuristic functions).

Purely heuristic functions will usually return immediately if their input is quite complicated in relation to the effort allowed. This is also true if the user has provided that input on the interactive level. In order to pose a difficult problem where a longer running time is acceptable, MAXEFFORT should be increased.

A simplification achieved by heuristic methods may speed up the deterministic parts, such that a small value of MAXEFFORT does not necessarily decrease the total computing time.

The user may employ MAXEFFORT in his own functions as follows: any function may use the amount of effort given by MAXEFFORT partly for own overhead, and distribute the rest on the functions it calls. To do this, the caller has to save the variable MAXEFFORT and set it to whatever it wants to make available to the called function. Depending on whether the call is necessary to obtain a correct result at all and whether the called function is a heuristic one, there are the following cases to handle. If the call is necessary and the called function is deterministic, MAXEFFORT has no influence. If the call is not absolutely necessary and as far as the called function is deterministic, the caller has to subtract the necessary amount as own overhead from MAXEFFORT if enough is available; otherwise, such call must not take place. As far as the called function works heuristically (for whatever reason it was called), it has to limit its efforts to the amount given by MAXEFFORT.

In no event may the value of MAXEFFORT on entering a procedure be different from the value on leaving it, even not in case of an error. save must be used to ensure this.

No function may distribute and/or use more than the amount it has been given by its caller. The own overhead should be estimated; if it is supposedly small, MAXEFFORT may be ignored.

In order to avoid casual, not reproducible effects, e.g., by other programs running on the same computer, MAXEFFORT should not be used in connection with time measurement using time or rtime. For example, the running time saved in one recursive call according to time measurement must not be supplied to another recursive call.

Examples

Example 1

The decomposition of an integer into prime factors is unique; hence the result of ifactor is uniquely determined, such that ifactor does not react to MAXEFFORT:

```
MAXEFFORT:= 0:
ifactor(2^10 + 1)
```

$$5^2\ 41$$

Example 2

The function solve may return unevaluated. Hence it will do so if there is no effort left to spend on the computation:

```
MAXEFFORT:= 0: solve(ln(x) + x = 3, x)
```

$$\text{solve}(x + \ln(x) - 3 = 0, x)$$

See Also

MuPAD Functions

`prog::ntime` | `time` | `traperror`

MAXLEVEL

Prevent infinite recursion during evaluation

Description

The environment variable MAXLEVEL determines the maximal substitution depth of identifiers. When this substitution depth is reached, an error occurs.

Possible values: integer greater 2; the maximum value depends on the operationg system, see below.

When a MuPAD object is evaluated, identifiers occurring in it are replaced by their values. This happens recursively, i.e., if the values themselves contain identifiers, then these are replaced as well. MAXLEVEL determines the maximal recursion depth of this process. If the substitution depth MAXLEVEL is reached, then an error occurs.

The purpose of MAXLEVEL is to provide a heuristic for recognizing infinite recursion with respect to the replacement of identifiers by their values, like in `delete a: a := a + 1; a`. If, in this example, the substitution depth would not be limited, then `a + 1` would be substituted for `a` infinitely often, and the system would "hang".

Similarly, the environment variable MAXDEPTH provides a heuristic for recognizing infinite recursion with respect to function calls; see the corresponding help page for details.

There is a close connection between LEVEL and MAXLEVEL. If the substitution depth LEVEL is reached during the evaluation process, then the recursion stops and any remaining identifiers remain unevaluated, but no error occurs.

Thus, if `MAXLEVEL > LEVEL`, then MAXLEVEL has no effect. By default, LEVEL and MAXLEVEL have the same value 100 at interactive level. However, the default value of LEVEL within a procedure is 1, and thus usually MAXLEVEL has no effect within procedures.

There are some notable differences between LEVEL and MAXLEVEL. The value of LEVEL depends on the context, namely whether the evaluation happens at interactive level or in a procedure. Moreover, some system functions, such as `context` and `level`, do not respect the current value of LEVEL. In contrast, MAXLEVEL is a global bound. It works as a last resort when the control of the evaluation via LEVEL fails.

The default value of MAXLEVEL is 100; MAXLEVEL has this value after starting or resetting the system via `reset`. Also the command `delete MAXLEVEL` restores the default value.

MAXLEVEL is a global variable. Use the statement `save MAXLEVEL` in a procedure to confine any changes to MAXLEVEL to this procedure.

The maximum value of MAXLEVEL depends on the operating system. Under Windows it is $2^{13} = 8192$. Under UNIX operating systems the maximum value depends on the maximum size of the C-stack. With a default stack size of `8 MB` the value is 8192, too; with a bigger stack size it can be bigger (in a bash the stack size can be set with `ulimit -s`).

Examples

Example 1

Evaluation of objects defined by an infinite recursion produces an error:

```
delete a: a := a + 1: a
```

```
Error: Recursive definition, the maximal evaluation level is reached.
```

This also works for mutually recursive definitions:

```
delete a, b: a := b^2: b := a + 1: b
```

```
Error: Recursive definition, the maximal evaluation level is reached.
```

Example 2

If MAXLEVEL is smaller or equal to LEVEL, as is the default at interactive level, then objects are evaluated completely up to depth MAXLEVEL-1, and an error occurs if the substitution depth MAXLEVEL is reached, whether a recursive definition is involved or not:

```
delete a, b, c, d:
a := b: b := c: c := 7: d := d + 1:
```

```
MAXLEVEL := 2: LEVEL := 2: c
```

$$7$$

```
a
```

```
Error: Recursive definition, the maximal evaluation level is reached.
```

```
d
```

```
Error: Recursive definition, the maximal evaluation level is reached.
```

On the other hand, MAXLEVEL has no effect if it exceeds LEVEL. Then any object is evaluated up to depth at most LEVEL, and the "recursive definition" error does not occur:

```
MAXLEVEL := 3: a, b, c, d
```

$$c, 7, 7, d + 2$$

In particular, MAXLEVEL normally has no effect within procedures, where by default LEVEL has the value 1:

```
MAXLEVEL := 2:
p := proc() begin a, d end_proc:
p();
delete MAXLEVEL, LEVEL:
```

$$b, d + 1$$

See Also

MuPAD Functions

context | eval | hold | LEVEL | level | MAXDEPTH | val

meijerG

The Meijer G function

Syntax

```
meijerG([[a1, …, an], [an + 1, …, ap]], [[b1, …, bm], [bm + 1, …, bq]], z)
```

```
meijerG([a1, …, an], [an + 1, …, ap], [b1, …, bm], [bm + 1, …, bq], z)
```

```
meijerG(m, n, [a1, …, ap], [b1, …, bq], z)
```

Description

`meijerG( [[ a1, …, an], [ an + 1, …, ap]], [[ b1, …, bm], [ bm + 1, …, bq]] , z)` represents the Meijer G function.

The following calls are equivalent:

`meijerG( [ a1, …, an], [ an + 1, …, ap], [ b1, …, bm], [ bm + 1, …, bq] , z)`, and

`meijerG(m, n, [ a1, …, an, an + 1, …, ap], [ b1, …, bm, bm + 1, …, bq] , z)`.

`meijerG( [[ a1, …, an], [ an + 1, …, ap]], [[ b1, …, bm], [ bm + 1, …, bq]] , z)` represents the Meijer G function $G^{m,n}_{p,q}\left(\begin{matrix}a_1,\ldots,a_n,a_{n+1},\ldots,a_p\\ b_1,\ldots,b_m,b_{m+1},\ldots,b_q\end{matrix}\middle|\, z\right)$. The function is defined as

$$G^{m,n}_{p,q}\left(\begin{matrix}a_1,\ldots,a_p\\ b_1,\ldots,b_q\end{matrix}\middle|\, z\right)=\frac{1}{2\pi i}\int\frac{\left(\prod_{j=1}^{m}\Gamma\left(b_j-s\right)\right)\left(\prod_{j=1}^{n}\Gamma\left(1-a_j+s\right)\right)}{\left(\prod_{j=m+1}^{q}\Gamma\left(1-b_j+s\right)\right)\left(\prod_{j=n+1}^{p}\Gamma\left(a_j-s\right)\right)}z^s\,ds,$$

where $0 \le m \le q$ and $0 \le n \le p$. The parameters a_i, b_j and the argument z can be complex numbers. The integral represents an inverse Laplace transform or, more specifically, a Mellin-Barnes type of integral. See the Algorithms section for more details.

If $m = 0$, $m = q$, $n = 0$, $n = p$, $p = 0$, or $q = 0$, you can pass empty parameter lists to `meijerG`: $[a_1, \ldots, a_n] = []$, $[a_{n+1}, \ldots, a_p] = []$, $[b_1, \ldots, b_m] = []$, or $[b_{m+1}, \ldots, b_q] = []$.

No pair of parameters a_i - b_j, $i = 1, \ldots, n$. $j = 1, \ldots, m$, should differ by a positive integer. Thus, no pole of $\Gamma\left(b_j - s\right)$ coincides with any pole of $\Gamma\left(1 - a_i + s\right)$. Otherwise, `meijerG` returns an error.

Meijer G functions with different parameters can represent the same function:

- The Meijer G function is symmetric with respect to the parameters. Changing the order inside each of the following lists of parameters does not change the resulting Meijer G function: $[a_1, \ldots, a_n]$, $[a_{n+1}, \ldots, a_p]$, $[b_1, \ldots, b_m]$, $[b_{m+1}, \ldots, b_q]$.
- If z is not a negative real number, the function satisfies the following identity:
$$G^{m,n}_{p,q}\left(\begin{matrix}a_1,\ldots,a_p\\ b_1,\ldots,b_q\end{matrix}\middle|\, z\right) = G^{n,m}_{q,p}\left(\begin{matrix}1-b_1,\ldots,1-b_q\\ 1-a_1,\ldots,1-a_p\end{matrix}\middle|\, \frac{1}{z}\right).$$
- If $0 < n < p$ and $r = a_1 - a_p$ is an integer, the function satisfies the following identity:
$$G^{m,n}_{p,q}\left(\begin{matrix}a_1,a_2,\ldots,a_{p-1},a_p\\ b_1,b_2,\ldots,b_{q-1},b_q\end{matrix}\middle|\, z\right) = (-1)^r\, G^{m,n}_{p,q}\left(\begin{matrix}a_p,a_2,\ldots,a_{p-1},a_1\\ b_1,b_2,\ldots,b_{q-1},b_q\end{matrix}\middle|\, z\right).$$
- If $0 < m < q$ and $r = b_1 - b_q$ is an integer, the function satisfies the following identity:
$$G^{m,n}_{p,q}\left(\begin{matrix}a_1,a_2,\ldots,a_{p-1},a_p\\ b_1,b_2,\ldots,b_{q-1},b_q\end{matrix}\middle|\, z\right) = (-1)^r\, G^{m,n}_{p,q}\left(\begin{matrix}a_1,a_2,\ldots,a_{p-1},a_p\\ b_q,b_2,\ldots,b_{q-1},b_1\end{matrix}\middle|\, z\right).$$

According to these rules, the `meijerG` function call can return `meijerG` with modified input parameters.

If at least one of the arguments is a floating-point number and all other arguments can be converted to floating-point numbers, the function returns a floating-point value.

Particular choices of parameters can reduce the Meijer G function to simpler special or elementary functions. Most special functions can be derived from the Meijer G function. In many cases, you can rewrite results involving `meijerG` in terms of more elementary functions using `simplify` or `Simplify`. See “Example 3” on page 1-1349.

The call `meijerG([[], []], [[], []], x)` returns 0.

Environment Interactions

When called with floating-point arguments, this function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

For exact or symbolic arguments, the `meijerG` function returns `meijerG`:

```
meijerG([[1],[]], [[],[2]],x)
```

$$G^{0,1}_{1,1}\left(\begin{matrix}1\\2\end{matrix}\middle|\, x\right)$$

```
meijerG([[1], [1/2]], [[], [1/2]], PI + I)
```

$$G^{1,0}_{1,2}\left(\begin{matrix}\frac{1}{2}\\0, \frac{1}{2}\end{matrix}\middle|\, \frac{1}{\pi + i}\right)$$

For floating-point arguments, `meijerG` returns floating-point values:

```
meijerG([[1], []], [[1], [1/2]], 3.0),
meijerG([[PI], [2]], [[], [3]], 4.0),
meijerG([[I+1,2], []], [[1/(I+1), 1/2],[]], 0.5*I)
```

$$0.7115950253,\ 0.3712122423,\ 0.3868927363 + 1.005593841\ i$$

Example 2

The functions `diff` and `float` handle expressions involving the Meijer G function:

```
diff(meijerG([[a], [b]], [[c], [d]], x), x)
```

$$\frac{G^{1,1}_{2,2}\left(\begin{matrix}a-1, b\\ c, d\end{matrix}\middle| x\right)}{x} + \frac{(a-1)\,G^{1,1}_{2,2}\left(\begin{matrix}a, b\\ c, d\end{matrix}\middle| x\right)}{x}$$

Differentiating a Meijer G function with respect to one of its parameters $a_1, \ldots, b_q$ does not generally result in Meijer G functions. Such derivatives are not implemented:

```
diff(meijerG([a], [b], [c], [d], z), a)
```

$$\frac{\partial}{\partial a}\, G^{1,1}_{2,2}\left(\begin{matrix}a, b\\ c, d\end{matrix}\middle| z\right)$$

You can evaluate the expressions involving `meierG` numerically using `float`:

```
meijerG([[1], []], [[2], [sqrt(PI)]], 3) ~=
  float(meijerG([[1], []], [[2], [sqrt(PI)]], 3))
```

$$G^{1,1}_{1,2}\left(\begin{matrix}1\\ 2, \sqrt{\pi}\end{matrix}\middle| 3\right) \approx -0.6659717596$$

```
delete z:
```

Example 3

Particular choices of parameters can reduce the Meijer G function to simpler special or elementary functions. Use `simplify` or `Simplify` to obtain such a representation:

```
simplify(meijerG([[], []], [[0], []], z))
```

$$e^{-z}$$

```
simplify(meijerG([[1], []], [[1/2], [0]], z))
```

$$\sqrt{\pi}\,\mathrm{erf}(\sqrt{z})$$

```
simplify(meijerG([[], []], [[1/2, -1/2], []], z))
```

$$2\,K_1(2\sqrt{z})$$

You can verify these relations numerically:

```
z:= float(PI+I):
meijerG([[], []], [[0], []], z) = exp(-z);
```

$$0.02334857968 - 0.03636325836\,i = 0.02334857968 - 0.03636325836\,i$$

```
meijerG([[1], []], [[1/2], [0]],z) = float(sqrt(PI)*erf(sqrt(z)))
```

$$1.76330129 + 0.01917545012\,i = 1.76330129 + 0.01917545012\,i$$

```
meijerG([[], []], [[1/2, -1/2], []], z) = 2*besselK(1, 2*sqrt(z))
```

$$0.03176922109 - 0.02400073308\,i = 0.03176922109 - 0.02400073308\,i$$

Parameters

a_1, …, a_p

The 'first list of parameters': arithmetical expressions

b_1, …, b_q

The 'second list of parameters': arithmetical expressions

z

The 'argument': an arithmetical expression

m, n

Integers satisfying $0 \le m \le q$, $0 \le n \le p$ or symbolic expressions.

Return Values

Arithmetical expression.

Overloaded By

z

Algorithms

$$G^{m,n}_{p,q}\left(\begin{matrix}a_1,\ldots,a_p\\ b_1,\ldots,b_q\end{matrix}\middle|\, z\right)=\frac{1}{2\pi i}\int\frac{\left(\prod_{j=1}^{m}\Gamma\left(b_j-s\right)\right)\left(\prod_{j=1}^{n}\Gamma\left(1-a_j+s\right)\right)}{\left(\prod_{j=m+1}^{q}\Gamma\left(1-b_j+s\right)\right)\left(\prod_{j=n+1}^{p}\Gamma\left(a_j-s\right)\right)}z^s\,ds$$

involves a complex contour integral with one of the following types of integration paths:

- The contour goes from - i ∞ to i ∞ so that all poles of $\Gamma\left(b_j-s\right)$, j = 1, …, m, lie to the right of the path, and all poles of $\Gamma\left(1-a_k+s\right)$, k = 1, …, n, lie to the left of the path. The integral converges if $c=m+n-\frac{p+q}{2}>0$, $|arg(z)|<c\,\pi$. If $|arg(z)|=c\,\pi$, $c\geq 0$, the integral converges absolutely when $p=q$ and $\Re(\psi)<-1$, where $\psi=\left(\sum_{j=1}^{q}b_j\right)-\left(\sum_{i=1}^{p}a_i\right)$. When $p\neq q$, the integral converges if you choose the contour so that the contour points near i ∞ and - i ∞ have a real part σ satisfying $(q-p)\,\sigma>\Re(\psi)+1-\frac{q-p}{2}$.
- The contour is a loop beginning and ending at *infinity* and encircling all poles of $\Gamma\left(b_j-s\right)$, j = 1, …, m, moving in the negative direction, but none of the poles of $\Gamma\left(1-a_k+s\right)$, k = 1, …, n. The integral converges if $q\geq 1$ and either $p<q$ or $p=q$ and $|z|<1$.
- The contour is a loop beginning and ending at - ∞ and encircling all poles of $\Gamma\left(1-a_k+s\right)$, k = 1, …, n, moving in the positive direction, but none of the poles of $\Gamma\left(b_j+s\right)$, j = 1, …, m. The integral converges if $p\geq 1$ and either $p>q$ or $p=q$ and $|z|>1$.

For a given set of parameters, the contour chosen in the definition of the Meijer G function is the one for which the integral converges. To avoid confusion, if the integral converges for several contours, all contours lead to the same function.

The Meijer G function satisfies a differential equation of order $max(p, q)$ with respect to a variable z:

$$\left((-1)^{m+n-p}\, z \left(\prod_{i=1}^{p}\left(z\,\frac{d}{dz}-a_i-1\right)\right)-\prod_{j=1}^{q}\left(z\,\frac{d}{dz}-b_j\right)\right) G^{m,n}_{p,q}\left(\begin{matrix}a_1,\ldots,a_p\\ b_1,\ldots,b_q\end{matrix}\,\middle|\, z\right)=0 .$$

If $p < q$, this differential equation has a regular singularity at $z = 0$ and an irregular singularity at $z = \infty$. If $p = q$, the points $z = 0$ and $z = \infty$ are regular singularities, and there is an additional regular singularity at $z = (-1)^{m+n-p}$.

The Meijer G function represents an analytic continuation of the Hypergeometric Function (for details, see Luke in the references). For particular choices of parameters, you can express the Meijer G function through the hypergeometric function. For example, if no two of the b_h terms, $h = 1, \ldots, m$, differ by an integer or zero, all poles are simple, and

$$G^{m,n}_{p,q}\left(\begin{matrix}a_1,\ldots,a_p\\ b_1,\ldots,b_q\end{matrix}\,\middle|\, z\right)=\sum_{h=1}^{m}\frac{\left(\prod_{\substack{j=1..m\\ j\neq h}}\Gamma\left(b_j-b_h\right)\right)\left(\prod_{j=1}^{n}\Gamma\left(1+b_h-a_j\right)\right)}{\left(\prod_{j=m+1}^{q}\Gamma\left(1+b_h-b_j\right)\right)\left(\prod_{j=n+1}^{p}\Gamma\left(a_j-b_h\right)\right)}\, z^{b_h}\, {}_pF_{q-1}\left(A_h;\right.$$

$$\left. B_h;\,(-1)^{p-m-n}\, z\right),$$

where $p < q$ or $p = q$ and $|z| < 1$. The symbols A_h, B_h denote

$$A_h = 1+b_h-a_1,\ \ldots,\ 1+b_h-a_p$$

and

$$B_h = 1+b_h-b_1,\ \ldots,\ 1+b_h-b_{h-1},\ 1+b_h-b_{h+1},\ \ldots,\ 1+b_h-b_q .$$

References

- Y.L. Luke, “The Special Functions and Their Approximations”, Vol. 1, Academic Press, New York, 1969.

- A.P. Prudnikov, Yu.A. Brychkov and O.I. Marichev, “Integrals and Series”, Vol. 3: More Special Functions, Gordon and Breach, 1990.
- M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, Dover Publications, New York, 9th printing, 1970.

See Also

MuPAD Functions

`hypergeom`

min

Minimum of numbers

Syntax

```
min(x1, x2, , …)

min({x1, x2, …})

min([x1, x2, …])

min(A)
```

Description

`min(x1, x2, ...)` returns the minimum of the numbers x_1, x_2, ….

If the arguments of `min` are integers, rational numbers, or floating-point numbers, then `min` returns the numerical minimum of these arguments.

The call `min()` is illegal and leads to an error message. If there is only one argument x_1, then `min` evaluates x_1 and returns it. See “Example 2” on page 1-1356.

If one of the arguments is `-infinity`, then `min` returns `-infinity`. If an argument is `infinity`, then it is removed from the argument list (see “Example 3” on page 1-1356).

`min` returns an error when one of its arguments is a complex number or a floating point interval with on-zero imaginary part (see “Example 2” on page 1-1356).

If one of the arguments is not a number, then a symbolic `min` call with the minimum of the numerical arguments and the remaining evaluated arguments may be returned (see “Example 1” on page 1-1355).

Nested `min` calls with symbolic arguments are rewritten as a single `min` call, i.e., they are flattened; see “Example 4” on page 1-1356.

`min` reacts to a very limited set of properties of identifiers set via `assume`. Use `simplify` to handle more general assumptions (see “Example 4” on page 1-1356).

Examples

Example 1

`min` computes the minimum of integers, rational numbers, and floating-point values:

```
min(-3/2, 7, 1.4)
```

$$-\frac{3}{2}$$

If the argument list contains symbolic expressions, then a symbolic `min` call is returned:

```
delete b: min(-4, b + 2, 1, 3)
```

$$\min(-4, b+2)$$

In the following two examples, `min` is able to determine the minimum despite getting symbolic arguments:

```
min(sqrt(2), 1)
```

$$1$$

```
assume(x > 0): min(exp(x), exp(-x))
```

$$e^{-x}$$

Floating point intervals are interpreted as “any number within this range” and may thus cause symbolic `min` calls to be returned:

```
min(2...3 union 6...7, 4)
```

$$\min(2.0 \ldots 3.0 \cup 6.0 \ldots 7.0, 4)$$

```
min(2...3, 6...7, 4)
```

$$2.0 \ldots 3.0$$

```
min(6...7, 4)
```

$$4$$

Example 2

`min` with one argument returns the evaluated argument:

```
delete a:
min(a), min(sin(2*PI)), min(2)
```

$$a,\ 0,\ 2$$

Complex numbers lead to an error message:

```
min(0, 1, I)
```

```
Error: The argument is invalid. [min]
```

Example 3

`-infinity` is always the minimum of arbitrary arguments:

```
delete x:
min(-100000000000, -infinity, x)
```

$$-\infty$$

`infinity` is removed from the argument list:

```
min(-100000000000, infinity, x)
```

$$\min(-100000000000,\ x)$$

Example 4

`min` reacts only to very few properties of identifiers set via `assume`:

```
delete a, b, c:
assume(a > 0 and b > a and c > b):
min(a, min(b, c), 0)
```

$$\min(a, b, c, 0)$$

An application of `simplify` yields the desired result:

```
simplify(%)
```

$$0$$

Parameters

x_1, x_2, …

Arbitrary MuPAD objects

A

An array of domain type `DOM_HFARRAY` with real entries

Return Values

One of the arguments, a floating-point number, or a symbolic `min` call.

Overloaded By

See Also

MuPAD Functions

`_leequal` | `_less` | `max` | `sort` | `sysorder`

mod, _mod

Modulo operator

Syntax

```
x mod m
```

```
_mod(x, m)
```

Description

If `m <> 0`, then `mod(x, m)` returns the value `x - n*m` where `n = floor(x/m)`. If `x` and `m` have different signs, then `mod(x, m)` has the same sign as `m`. If `m = 0`, then `mod(x, m)` returns `x`. See “Example 1” on page 1-1359.

`_mod(x, m)` is the functional equivalent of the operator notation `x mod m`. See “Example 2” on page 1-1359.

By default, `x mod m` and `_mod(x, m)` are both equivalent to `modp(x, m)`. You can redefine the modulo operator `mod` and its functional form `_mod` by using `modp` and `mods`. For example, after the assignment `_mod:=mods`, both the operator `mod` and the equivalent function `_mod` return remainders of least absolute value. See “Example 3” on page 1-1360.

All functions return an error when one of the arguments is a floating-point number, a complex number, or not an arithmetical expression.

If one of the arguments is not a number, then a symbolic function call is returned. See “Example 4” on page 1-1360.

`_mod` and `modp` are kernel functions.

Environment Interactions

By default the operator `mod` and the function `_mod` are equivalent to `modp`. This can be changed by assigning a new value to `_mod`. See “Example 3” on page 1-1360.

Examples

Example 1

Find the modulus after division of these integers.

```
27 mod 4, 27 mod -4, -27 mod 4, -27 mod -4
```

$$3, -1, 1, -3$$

Find the modulus after division by zero.

```
9 mod 0, -9 mod 0, 0 mod 0
```

$$9, -9, 0$$

Find the modulus after division of these rational numbers.

```
22/3 mod 5, 22 mod 5/3, 22/3 mod 5/4
```

$$\frac{7}{3}, \frac{1}{3}, \frac{13}{12}$$

Example 2

Find the modulus after division of 23 by 5 using the modulo operator and its functional form. `_mod` and the operator `mod` are equivalent.

```
hold(_mod(23, 5))
```

$$23 \bmod 5$$

```
23 mod 5 = _mod(23, 5)
```

$$3 = 3$$

Example 3

By default the binary operator `mod` and the equivalent function `_mod` are both equivalent to `modp`. You can redefine `_mod`.

```
modp(11, 7), mods(11,7);
11 mod 7
```

$$4,\ -3$$

$$4$$

```
_mod := mods:
11 mod 7
```

$$-3$$

For further computations, define `_mod` as `modp`.

```
_mod := modp:
```

Example 4

If one of the arguments is not a number, then the modulo operator returns a symbolic function call.

```
delete x, m:
x mod m, x mod 2, 2 mod m
```

$$x \bmod m,\ x \bmod 2,\ 2 \bmod m$$

When called with nonnumeric arguments, the function currently associated with `_mod` is printed in the operator notation.

```
_mod := mods:
modp(x, m), mods(x, m)
```

$$x \bmod m,\ x \bmod m$$

```
_mod := modp:
modp(x, m), mods(x, m)
```

$$x \bmod m,\ \mathrm{mods}(x,\ m)$$

Parameters

x

An integer, a rational number, or an arithmetical expression

m

An integer or an arithmetical expression

Return Values

arithmetical expression.

Overloaded By

m, x

See Also

MuPAD Domains
`Dom::IntegerMod`

MuPAD Functions
`/` | `div` | `divide` | `frac` | `gcd` | `gcdex` | `igcd` | `igcdex` | `IntMod` | `modp` | `mods` | `powermod`

modp

Positive modulo function

Syntax

```
modp(x, m)
```

Description

If `m <> 0`, then `modp(x, m)` returns the value `x - n*m` where `n = floor(x/m)`. If `x` and `m` have different signs, then `modp(x, m)` has the same sign as `m`. If `m = 0`, then `modp(x, m)` returns `x`. See "Example 1" on page 1-1362.

By default, `x mod m` and `_mod(x, m)` are both equivalent to `modp(x, m)`. You can redefine the modulo operator `mod` and its functional form `_mod` by using `modp` and `mods`. For example, after the assignment `_mod:=mods`, both the operator `mod` and the equivalent function `_mod` return remainders of least absolute value. See "Example 2" on page 1-1363.

All functions return an error when one of the arguments is a floating-point number, a complex number, or not an arithmetical expression.

If one of the arguments is not a number, then a symbolic function call is returned. See "Example 3" on page 1-1363.

`_mod` and `modp` are kernel functions.

Examples

Example 1

Find the modulus after division of these integers.

```
modp(27, 4), modp(27, -4), modp(-27, 4), modp(-27, -4)
```

```
3, -1, 1, -3
```

Find the modulus after division by zero.

```
modp(9, 0), modp(-9, 0), modp(0, 0)
```

```
9, -9, 0
```

Find the modulus after division of these rational numbers.

```
modp(22/3, 5), modp(22, 5/3), modp(22/3, 5/4)
```

$$\frac{7}{3}, \frac{1}{3}, \frac{13}{12}$$

Example 2

By default the binary operator `mod` and the equivalent function `_mod` are both equivalent to `modp`. You can redefine `_mod`.

```
modp(11, 7), mods(11,7);
11 mod 7
```

```
4, -3
```

```
4
```

```
_mod := mods:
11 mod 7
```

```
-3
```

For further computations, define `_mod` as `modp`.

```
_mod := modp:
```

Example 3

If one of the arguments is not a number, then `modp` returns a symbolic function call.

```
delete x, m:
modp(x, m), modp(x, 2), modp(2, m)
```

$$x \bmod m,\ x \bmod 2,\ 2 \bmod m$$

When called with nonnumeric arguments, the function currently associated with `_mod` is printed in the operator notation.

```
_mod := mods:
modp(x, m), mods(x, m)
```

$$x \bmod m,\ x \bmod m$$

```
_mod := modp:
modp(x, m), mods(x, m)
```

$$x \bmod m,\ \mathrm{mods}(x,\ m)$$

Parameters

x

An integer, a rational number, or an arithmetical expression

m

An integer or an arithmetical expression

Return Values

arithmetical expression.

Overloaded By

m, x

See Also

MuPAD Domains
`Dom::IntegerMod`

MuPAD Functions
`/` | `div` | `divide` | `frac` | `gcd` | `gcdex` | `igcd` | `igcdex` | `IntMod` | `mod` | `mods` | `powermod`

mods

Symmetric modulo function

Syntax

```
mods(x, m)
```

Description

If `m <> 0`, then `mods(x, m)` returns the value `x + n*m` where `n = round(-x/m)`. If `m = 0`, then `mods(x, m)` returns `x`. See "Example 1" on page 1-1366.

By default, `x mod m` and `_mod(x, m)` are both equivalent to `modp(x, m)`. You can redefine the modulo operator `mod` and its functional form `_mod` by using `modp` and `mods`. For example, after the assignment `_mod:=mods`, both the operator `mod` and the equivalent function `_mod` return remainders of least absolute value. See "Example 2" on page 1-1367.

All functions return an error when one of the arguments is a floating-point number, a complex number, or not an arithmetical expression.

If one of the arguments is not a number, then a symbolic function call is returned. See "Example 3" on page 1-1367.

`mods` is a kernel function.

Examples

Example 1

Use the symmetric modulo function to find the modulus after division of these integers.

```
mods(27, 4), mods(27, -4), mods(-27, 4), mods(-27, -4)
```

$$-1, -1, 1, 1$$

Find the modulus after division by zero.

```
mods(9, 0), mods(-9, 0), mods(0, 0)
```

$$9, -9, 0$$

Use the symmetric modulo function to find the modulus after division of these rational numbers.

```
mods(22/3, 5), mods(22, 5/3), mods(22/3, 5/4)
```

$$\frac{7}{3}, \frac{1}{3}, -\frac{1}{6}$$

Example 2

By default the binary operator `mod` and the equivalent function `_mod` are both equivalent to `modp`. You can redefine `_mod`.

```
modp(11, 7), mods(11,7);
11 mod 7
```

$$4, -3$$

$$4$$

```
_mod := mods:
11 mod 7;
```

$$-3$$

For further computations, define `_mod` as `modp`.

```
_mod := modp:
```

Example 3

If one of the arguments is not a number, then the modulo operator returns a symbolic function call.

```
delete x, m:
x mod m, x mod 2, 2 mod m
```

$$x \bmod m,\ x \bmod 2,\ 2 \bmod m$$

When called with nonnumeric arguments, the function currently associated with `_mod` is printed in the operator notation.

```
_mod := mods: modp(x, m), mods(x, m)
```

$$x \bmod m,\ x \bmod m$$

```
_mod := modp: modp(x, m), mods(x, m)
```

$$x \bmod m,\ \mathrm{mods}(x,\ m)$$

Parameters

x

An integer, a rational number, or an arithmetical expression

m

An integer or an arithmetical expression

Return Values

arithmetical expression.

Overloaded By

m, x

See Also

MuPAD Domains
`Dom::IntegerMod`

MuPAD Functions
`/` | `div` | `divide` | `frac` | `gcd` | `gcdex` | `igcd` | `igcdex` | `IntMod` | `mod` | `modp` | `powermod`

monomials

Sorted list of monomials of a polynomial

Syntax

```
monomials(p, <order>)
```

```
monomials(f, <vars>, <order>)
```

Description

`monomials(p, order)` returns the list of non-zero monomials of the polynomial `p`. The list is sorted with respect to the term ordering `order`.

`monomials` returns a list of all non-trivial monomials of the polynomial given. The monomials are sorted according to the term ordering given. The list is empty if the polynomial is zero.

A polynomial expression `f` is first converted to a polynomial with the variables given by `vars`. If no variables are given, they are searched for in `f`. See `poly` about details of the conversion. The result is returned as list of polynomial expressions. `FAIL` is returned if `f` cannot be converted to a polynomial.

The result of `monomials` is not fully evaluated. It can be evaluated by the functions `mapcoeffs` and `eval`. Cf. “Example 4” on page 1-1372.

Examples

Example 1

We give some self explaining examples:

```
p := poly(100*x^100 + 49*x^49 + 7*x^7, [x]):
monomials(p)
```

$$[\mathrm{poly}(100\,x^{100}, [x]), \mathrm{poly}(49\,x^{49}, [x]), \mathrm{poly}(7\,x^{7}, [x])]$$

```
monomials(poly(0, [x]))
```

$$[]$$

```
delete p:
```

Example 2

We demonstrate the effect of various term orders:

```
p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x, y, z]):
monomials(p)
```

$$[\mathrm{poly}(5\,x^4, [x, y, z]), \mathrm{poly}(4\,x^3\,y\,z^2, [x, y, z]), \mathrm{poly}(3\,x^2\,y^3\,z, [x, y, z]), \mathrm{poly}(2, [x, y, z])]$$

```
monomials(p, DegreeOrder)
```

$$[\mathrm{poly}(4\,x^3\,y\,z^2, [x, y, z]), \mathrm{poly}(3\,x^2\,y^3\,z, [x, y, z]), \mathrm{poly}(5\,x^4, [x, y, z]), \mathrm{poly}(2, [x, y, z])]$$

```
monomials(p, DegInvLexOrder)
```

$$[\mathrm{poly}(3\,x^2\,y^3\,z, [x, y, z]), \mathrm{poly}(4\,x^3\,y\,z^2, [x, y, z]), \mathrm{poly}(5\,x^4, [x, y, z]), \mathrm{poly}(2, [x, y, z])]$$

```
delete p:
```

Example 3

This example features a user defined term ordering. Here we use the reverse lexicographical order on 3 indeterminates:

```
order := Dom::MonomOrdering(RevLex(3)):
p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x, y, z]):
monomials(p, order)
```

$$[\mathrm{poly}(3\,x^2\,y^3\,z, [x, y, z]), \mathrm{poly}(4\,x^3\,y\,z^2, [x, y, z]), \mathrm{poly}(5\,x^4, [x, y, z]), \mathrm{poly}(2, [x, y, z])]$$

```
delete order, p:
```

Example 4

We demonstrate the evaluation strategy of `monomials`:

```
p := poly(3*x^3 + 6*x^2*y^2 + 2, [x]): y := 4:
monomials(p)
```

$$[\mathrm{poly}(3\,x^3, [x]),\ \mathrm{poly}((6\,y^2)\,x^2, [x]),\ \mathrm{poly}(2, [x])]$$

Evaluation is enforced by `eval`:

```
map(%, mapcoeffs, eval)
```

$$[\mathrm{poly}(3\,x^3, [x]),\ \mathrm{poly}(96\,x^2, [x]),\ \mathrm{poly}(2, [x])]$$

```
delete p, y:
```

Parameters

`p`

A polynomial of type `DOM_POLY`

`f`

A polynomial expression

`vars`

A list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

`order`

The term ordering: `LexOrder`, or `DegreeOrder`, or `DegInvLexOrder`, or a user-defined term ordering of type `Dom::MonomOrdering`. The default is the lexicographical ordering `LexOrder`.

Return Values

List of polynomials of the same type as `p`. A list of expressions is returned if an expression is given. The list is empty if the polynomial is zero.

Overloaded By

p

See Also

MuPAD Functions

`coeff` | `degree` | `degreevec` | `ground` | `lcoeff` | `ldegree` | `lmonomial` | `lterm` | `nterms` | `nthcoeff` | `nthmonomial` | `nthterm` | `poly` | `poly2list` | `tcoeff`

mtaylor

Compute a multivariate Taylor series expansion

Syntax

```
mtaylor(f, x = x0, <order>, <mode>, <weights>, <Mapcoeffs = mc>)
mtaylor(f, x, <order>, <mode>, <weights>, <Mapcoeffs = mc>)
mtaylor(f, x = x0, AbsoluteOrder = order, <weights>, <Mapcoeffs = mc>)
mtaylor(f, x = x0, RelativeOrder = order, <weights>, <Mapcoeffs = mc>)
mtaylor(f, [x = x0, y = y0, …], <order>, <mode>, <weights>, <Mapcoeffs = mc>)
mtaylor(f, [x, y, …], <order>, <mode>, <weights>, <Mapcoeffs = mc>)
mtaylor(f, [x = x0, y = y0, …], <AbsoluteOrder = order>, <weights>, <Mapcoeffs = mc>)
mtaylor(f, [x = x0, y = y0, …], RelativeOrder = order, <weights>, <Mapcoeffs = mc>)
```

Description

`mtaylor(f, [x = x0, y = y0, ...])` computes the first terms of the multivariate Taylor series of `f` with respect to the variables `x, y` etc. around the points `x = x0, y = y0` etc.

With the default mode `RelativeOrder`, the number of requested terms for the expansion is determined by `order` if specified. If no order is specified, the value of the environment variable `ORDER` is used. You can change the default value 6 by assigning a new value to `ORDER`.

The terms are counted from the lowest total degree on for finite expansion points, and from the highest total degree term on for expansions around infinity.

If `AbsoluteOrder` is specified, `order` represents the truncation order of the series, i.e., no terms of total degree `order` or higher are computed.

For infinite expansion points, the absolute values of the exponents of the corresponding variables are used to compute the total degree.

For finite expansion points `x0, y0, ...`, the computed series with respect to the variables `x, y, ...` of weight `w1, w2, ...` is

```
taylor(f(x0 + t^w1*(x - x0), y0 + t^w2*(y - y0), dots), t = 0),
```

evaluated at the point $t = 1$.

Environment Interactions

The function is sensitive to the environment variable `ORDER`, which determines the default number of terms in series computations.

Examples

Example 1

We compute a Taylor series around the origin (default). The expansion contains all terms through total degree 3:

```
mtaylor(exp(x^2 - y), [x, y], 4)
```

$$-x^2 y + x^2 - \frac{y^3}{6} + \frac{y^2}{2} - y + 1$$

We request additional terms of higher order:

```
mtaylor(exp(x^2 - y), [x, y], 5)
```

$$\frac{x^4}{2} + \frac{x^2 y^2}{2} - x^2 y + x^2 + \frac{y^4}{24} - \frac{y^3}{6} + \frac{y^2}{2} - y + 1$$

In the example above, the leading term is of total degree 0. In the following example, the leading term is of total degree 2. Thus, the default mode `RelativeOrder` produces terms of total degree smaller than4 + 2 = 6:

```
mtaylor(x*y*exp(x^2 - y), [x, y], 4)
```

$$-x^3 y^2 + x^3 y - \frac{x y^4}{6} + \frac{x y^3}{2} - x y^2 + x y$$

We request an absolute truncation order of 4, so that only terms of total degree smaller than 4 are computed:

```
mtaylor(x*y*exp(x^2 - y), [x, y], AbsoluteOrder = 4)
```

$$x y - x y^2$$

Example 2

For infinite expansions points a series in the reciprocal of the variable is returned:

```
mtaylor(exp(z)/(x - y), [x = infinity, y = 0, z])
```

$$\frac{y^2}{x^3} + \frac{z^2}{2 x} + \frac{z^3}{6 x} + \frac{z^4}{24 x} + \frac{z^5}{120 x} + \frac{y}{x^2} + \frac{z}{x} + \frac{1}{x} + \frac{y z}{x^2} + \frac{y z^2}{2 x^2} + \frac{y z^3}{6 x^2} + \frac{y^2 z}{x^3}$$

We reduce the order in `z` by giving `z` a higher weight:

```
mtaylor(exp(z)/(x - y), [x = infinity, y = 0, z], [1, 1, 2])
```

$$\frac{y^2}{x^3} + \frac{z^2}{2 x} + \frac{y}{x^2} + \frac{z}{x} + \frac{1}{x} + \frac{y z}{x^2}$$

Example 3

If a Taylor series expansion does not exist, or if `mtaylor` cannot find a Taylor series expansion, then `mtaylor` throws an error.

Try to find the Taylor series expansion of $f(x) = \frac{1}{x y - 1}$ around $x = 1$, $y = 1$. The Taylor series expansion does not exist, and `mtaylor` throws an error:

```
mtaylor(1/(x*y - 1), [x = 1, y = 1])
```

```
Error: Cannot compute a Taylor expansion of '1/(x*y - 1)'. [mtaylor]
```

Example 4

This is an example of a directed Taylor expansion along the real axis around `x = infinity`:

```
mtaylor(sqrt(y)*sin(sqrt(y)/x), [x = infinity, y = 0])
```

$$\frac{y}{x} - \frac{y^2}{6\,x^3}$$

In fact, this is even an undirected expansion:

```
mtaylor(sqrt(y)*sin(sqrt(y)/x), [x = complexInfinity, y = 0])
```

$$\frac{y}{x} - \frac{y^2}{6\,x^3}$$

Example 5

A common problem in symbolic calculations is "expression swell:" Intermediate expressions which are not or cannot be simplified lead to unnecessarily complicated results. The following is an example of such behavior:

```
mtaylor((a+x)^n, x, 4)
```

$$\sigma_1 - x^2\,\sigma_1\left(\frac{n}{2\,a^2} - \frac{n^2}{2\,a^2}\right) - x^3\,\sigma_1\left(\frac{n^2}{4\,a^3} - \frac{n}{3\,a^3} + \frac{n\left(\frac{n}{4\,a^2} - \frac{n^2}{6\,a^2}\right)}{a}\right) + \frac{n\,x\,\sigma_1}{a}$$

where

$$\sigma_1 = e^{n\ln(a)}$$

In general, applying `simplify` or `Simplify` to complicated results is a strategy that often helps. In this case, however, it would destroy the format of the series:

```
simplify(%)
```

$$\frac{a^{n-3}\,(6\,a^3+6\,a^2\,n\,x+3\,a\,n^2\,x^2-3\,a\,n\,x^2+n^3\,x^3-3\,n^2\,x^3+2\,n\,x^3)}{6}$$

What is required is a way to map a function like `simplify` to the coefficients of the series only. Since `mtaylor` returns an ordinary expression, this must be done in the `mtaylor` call itself, using the `Mapcoeffs` option:

```
mtaylor((a+x)^n, x, 4, Mapcoeffs=simplify)
```

$$a^n+a^{n-1}\,n\,x+\frac{a^{n-2}\,n\,x^2\,(n-1)}{2}+\frac{a^{n-3}\,n\,x^3\,(n^2-3\,n+2)}{6}$$

Parameters

`f`

An arithmetical expression representing a function in `x, y, ...`

`x, y, …`

identifiers or indexed identifiers

`x0, y0, …`

The expansion points: arithmetical expressions. Also expressions involving `infinity` or `complexInfinity` are accepted.

If not specified, the default expansion point 0 is used.

`order`

The truncation order (in conjunction with `AbsoluteOrder`) or, in conjunction with `RelativeOrder`, the number of terms to be computed, respectively. A nonnegative integer; the default order is given by the environment variable `ORDER` (default value 6).

The order concept refers to the total degree in the variables (the sum of all exponents).

`mode`

One of the flags `AbsoluteOrder` or `RelativeOrder`. The default is `RelativeOrder`.

`weights`

A list of positive integers determining the number of terms of the computed series. A variable `x` with weight `w` contributes as `x^w` to the total degree of the terms in the series. Thus, using weight 2 for *x*, halves the order in `x` to which the series is computed.

By default, all variables have the weight 1.

Options

`AbsoluteOrder`

With this flag, the integer value `order` is the truncation order of the computed series, i.e., only terms of total degree less than `order` are present.

`RelativeOrder`

With this flag, the terms in the computed series range from some leading total degree `v` to the highest total degree `v + order - 1` (i.e., the truncation order w.r.t. the total degree is `v + order`).

`Mapcoeffs`

Option, specified as `Mapcoeffs = mc`

When building the resulting expression, for each coefficient `c`, insert `mc(c)` instead.

Return Values

Arithmetical expression.

Overloaded By

f

See Also

MuPAD Functions

`asympt` | `diff` | `limit` | `O` | `series` | `Series::Puiseux` | `taylor` | `Type::Series`

multcoeffs

Multiply the coefficients of a polynomial with a factor

Syntax

```
multcoeffs(p, c)
```

```
multcoeffs(f, <vars>, c)
```

Description

`multcoeffs(p, c)` multiplies all coefficients of the polynomial `p` with the factor `c`.

A polynomial expression `f` is first converted to a polynomial with the variables given by `vars`. If no variables are given, they are searched for in `f`. See `poly` about details of the conversion. `FAIL` is returned if `f` cannot be converted to a polynomial. After multiplication with `c`, the result is converted to an expression.

For a polynomial expression `f`, the factor `c` may be any arithmetical expression. For a polynomial `p` of type `DOM_POLY`, the factor `c` must be convertible to an element of the coefficient ring of `p`.

Examples

Example 1

Some simple examples:

```
multcoeffs(3*x^3 + x^2*y^2 + 2, 5)
```

$$15\,x^3+5\,x^2\,y^2+10$$

```
multcoeffs(3*x^3 + x^2*y^2 + 2, c)
```

$$3\,c\,x^3+c\,x^2\,y^2+2\,c$$

```
multcoeffs(poly(x^3 + 2, [x]), sin(y))
```

$$\mathrm{poly}(\sin(y)\, x^3 + 2\, \sin(y), [x])$$

Example 2

Mathematically, `multcoeffs(f, c)` is the same as `f*c`. However, `multcoeffs` produces an expanded form of the product which depends on the indeterminates:

```
f := 3*x^3 + x^2*y^2 + 2:
multcoeffs(f, [x], c), multcoeffs(f, [y], c),
multcoeffs(f, [z], c)
```

$$3\, c\, x^3 + c\, x^2\, y^2 + 2\, c,\ c\, (3\, x^3 + 2) + c\, x^2\, y^2,\ c\, (3\, x^3 + x^2\, y^2 + 2)$$

```
delete f:
```

Parameters

`p`

A polynomial of type `DOM_POLY`

`c`

An arithmetical expression or an element of the coefficient ring of `p`

`f`

A polynomial expression

`vars`

A list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

Return Values

Polynomial of type `DOM_POLY`, or a polynomial expression, or `FAIL`.

Overloaded By

f, p

See Also

MuPAD Functions

`coeff` | `degree` | `degreevec` | `lcoeff` | `ldegree` | `lterm` | `monomials` | `nterms` | `nthcoeff` | `nthmonomial` | `nthterm` | `poly` | `tcoeff`

new

Create a domain element

Syntax

```
new(T, object1, object2, …)
```

Description

Within a method of the domain type `T`, `new(T, object1, object2, ...)` creates a new element of the domain `T` with the internal representation `object1, object2, ...`.

`new` is a low-level function for creating elements of library domains.

The internal representation of a domain element comprises a reference to the corresponding domain and an arbitrary number of MuPAD objects, the internal operands of the domain element.

`new(T, object1, object2, ...)` creates a new element of the domain `T`, whose internal representation is the sequence of operands `object1, object2, ...`, and returns this element.

`new(T)` creates a new element of the domain `T`, whose internal representation is an empty sequence of operands.

Note: `new` is intended only for programmers implementing their own domains in MuPAD. You should never use `new` directly to generate elements of a predefined domain `T`; use the corresponding constructor `T(...)` instead, for the following reasons. The internal representation of the predefined MuPAD domains may be subject to changes more often than the interface provided by the constructor. Moreover, in contrast to `new`, the constructors usually perform argument checking. Thus using `new` directly may lead to invalid internal representations of MuPAD objects.

New domains can be created via `newDomain`.

You can access the operands of the internal representation of a domain element via `extop`, which, in contrast to `op`, cannot be overloaded for the domain. The function `op` is sometimes overloaded for a domain in order to hide the internal, technical representation of an object and to provide a more user friendly and intuitive interface.

Similarly, the function `extnops` returns the number of operands of a domain element in the internal representation, and `extsubsop` modifies an operand in the internal representation. These functions, in contrast to the related functions `nops` and `subsop`, cannot be overloaded for a domain.

You can write a constructor for your own domain `T` by providing a `"new"` method. This method is invoked whenever the user calls `T(arg1, arg2, ...)`. This is recommended since it provides a more elegant and intuitive user interface than `new`. The `"new"` method usually performs some argument checking and converts the arguments `arg1, arg2, ...` into the internal representation of the domain, using `new` (see “Example 1” on page 1-1385).

Examples

Example 1

We create a new domain `Time` for representing clock times. The internal representation of an object of this domain has two operands: the hour and the minutes. Then we create a new domain element for the time `12:45`:

```
Time := newDomain("Time"):
a := new(Time, 12, 45)
```

```
new(Time, 12, 45)
```

The domain type of `a` is `Time`, the number of operands is `2`, and the operands are `12` and `45`:

```
domtype(a), extnops(a)
```

```
Time, 2
```

```
extop(a)
```

```
12, 45
```

We now implement a `"new"` method for our new domain `Time`, permitting several input formats. It expects either two integers, the hour and the minutes, or only one integer that represents the minutes, or a rational number or a floating-point number, implying that the integral part is the hour and the fractional part represents a fraction of an hour corresponding to the minutes, or no arguments, representing midnight. Additionally, the procedure checks that the arguments are of the correct type:

```
Time::new := proc(HR = 0, MN = 0)
  local m;
begin
  if args(0) = 2 and domtype(HR) = DOM_INT
     and domtype(MN) = DOM_INT then
    m := HR*60 + MN
  elif args(0) = 1 and domtype(HR) = DOM_INT then
    m := HR
  elif args(0) = 1 and domtype(HR) = DOM_RAT then
    m := trunc(float(HR))*60 + frac(float(HR))*60
  elif args(0) = 1 and domtype(HR) = DOM_FLOAT then
    m := trunc(HR)*60 + frac(HR)*60
  elif args(0) = 0 then
    m := 0
  else
    error("wrong number or type of arguments")
  end_if;
  new(Time, trunc(m/60), trunc(m) mod 60)
end_proc:
```

Now we can use this method to create new objects of the domain `Time`, either by calling `Time::new` directly, or, preferably, by using the equivalent but shorter call `Time(...)`:

```
Time::new(12, 45), Time(12, 45), Time(12 + 3/4)
```

```
new(Time, 12, 45), new(Time, 12, 45), new(Time, 12, 45)
```

```
Time(), Time(8.25), Time(1/2)
```

```
new(Time, 0, 0), new(Time, 8, 15), new(Time, 0, 30)
```

In order to have a nicer output for objects of the domain `Time`, we also define a `"print"` method (see the help page for `print`):

```
Time::print := proc(TM)
begin
  expr2text(extop(TM, 1)) . ":" .
  stringlib::format(expr2text(extop(TM, 2)), 2, Right, "0")
end_proc:
```

```
Time::new(12, 45), Time(12, 45), Time(12 + 3/4)
```

```
     12:45, 12:45, 12:45
```

```
Time(), Time(8.25), Time(1/2)
```

```
     0:00, 8:15, 0:30
```

Parameters

T

A MuPAD domain

object1, object2, …

Arbitrary MuPAD objects

Return Values

Element of the domain T.

See Also

MuPAD Domains
DOM_DOMAIN

MuPAD Functions
extnops | extop | extsubsop | newDomain | op

newDomain

Create a new data type (domain)

Syntax

```
newDomain(k)
```

```
newDomain(k, T)
```

```
newDomain(k, t)
```

Description

`newDomain(k)` creates a new domain with key `k`.

`newDomain(k, T)` creates a copy of the domain `T` with new key `k`.

`newDomain(k, t)` creates a new domain with key `k` and slots from the table `t`.

Data types in MuPAD are called *domains*. `newDomain` is a low-level function for defining new data types. Cf. the corresponding entry in the Glossary for links to documentation about domains and more comfortable ways of defining new data types. The help page of `DOM_DOMAIN` contains a tutorial example for defining a new domain via `newDomain`.

Technically, a domain is something like a table. The entries of this table are called slots or *methods*. They serve for extending the functionality of standard MuPAD functions, such as the arithmetic operations `+` and `*`, the special mathematical functions `exp` and `sin`, or the symbolic manipulation functions `simplify` and `normal`, to objects of a domain in a modular, object-oriented way, without the need to modify the source code of the standard function. This is known as overloading.

The function `slot` and the equivalent operator `::` serve for defining and accessing a specific slot of a domain. The function `op` returns all slots of a domain.

Each domain has a distinguished slot `"key"`, which is its unique identification. There can be no two different domains with the same key. Typically, but not necessarily, the key is a string. However, the key serves mainly for internal and output purposes. Usually

a domain is assigned to an identifier immediately after its creation, and you access the domain via this identifier.

If a domain with the given key already exists, `newDomain(k)` returns that domain; both other forms of calling `newDomain` yield an error.

Examples

Example 1

We create new domain with key `"my-domain"`. This key is also used for output, but without quotes:

```
T := newDomain("my-domain")
```

```
my-domain
```

You can create elements of this domain with the function `new`:

```
e := new(T, 42);
domtype(e)
```

```
new(my-domain, 42)
```

```
my-domain
```

With the slot operator `::`, you can define a new slot or access an existing one:

```
op(T)
```

```
"key" = "my-domain"
```

```
T::key, T::myslot
```

```
"my-domain", FAIL
```

```
T::myslot := 42: op(T)
```

```
"key" = "my-domain", "myslot" = 42
```

```
T::myslot^2
```

```
1764
```

If a domain with key `k` already exists, then `newDomain(k)` does not create a new domain, but returns the existing domain instead:

```
T1 := newDomain("my-domain"):
op(T1)
```

```
"key" = "my-domain", "myslot" = 42
```

Note that you cannot delete a domain; the command `delete T` only deletes the value of the identifier `T`, but does not destroy the domain with the key `"my-domain"`:

```
delete T, T1:
T2 := newDomain("my-domain"):
op(T2);
delete T2:
```

```
"key" = "my-domain", "myslot" = 42
```

Example 2

There cannot exist different domains with the same key at the same time. Defining a slot for a domain implicitly changes all identifiers that have this domain as their value:

```
T := newDomain("1st"): T1 := T:
op(T);
op(T1);
```

```
"key" = "1st"
```

```
"key" = "1st"
```

```
T1::mySlot := 42:
op(T);
op(T1);
```

"key" = "1st", "mySlot" = 42

"key" = "1st", "mySlot" = 42

To avoid this, you can create a copy of a domain. You must reserve a new, unused key for that copy:

```
T2 := newDomain("2nd", T):
T2::anotherSlot := infinity:
op(T);
op(T2);
```

"key" = "1st", "mySlot" = 42

"key" = "2nd", "mySlot" = 42, "anotherSlot" = ∞

```
delete T, T1, T2:
```

Example 3

You can provide a domain with slots already when creating it:

```
T := newDomain("3rd",
  table("myslot" = 42, "anotherSlot" = infinity)):
op(T);
T::myslot, T::anotherSlot
```

"myslot" = 42, "anotherSlot" = ∞, "key" = "3rd"

42, ∞

```
delete T:
```

Parameters

k

An arbitrary object; typically a string

T

A domain

t

The slots of the domain: a table

Return Values

Object of type `DOM_DOMAIN`.

See Also

MuPAD Domains
`DOM_DOMAIN`

MuPAD Functions
`domtype` | `new` | `slot`

More About

- "Define Your Own Data Types"

next, _next

Skip a step in a loop

Syntax

```
next
```

```
_next()
```

Description

`next` interrupts the current step in `for`, `repeat`, and `while` loops. Execution proceeds with the next step of the loop.

The `next` statement is equivalent to the function call `_next()`. The return value is the void object of type `DOM_NULL`.

Inside `for`, `repeat`, and `while` loops, the `next` statement interrupts the current step of the loop. In `for` statements, the loop variable is incremented and execution continues at the beginning of the loop. Similarly, the control conditions at the beginning of a `while` loop and in the `until` clause of a `repeat` loop are verified, before execution continues at the beginning of the loop.

Outside `for`, `repeat`, and `while` loops, the `next` statement has no effect.

Examples

Example 1

In the following `for` loop, any step with even `i` is skipped:

```
for i from 1 to 5 do
  if testtype(i, Type::Even) then next end_if;
  print(i)
end_for:
```

```
    1

    3

    5
```

In the following `repeat` loop, all steps with odd `i` are skipped:

```
i := 0:
repeat
  i := i + 1;
  if testtype(i, Type::Odd) then next end_if;
  print(i)
until i >= 5 end_repeat:
```

```
    2

    4
```

```
delete i:
```

See Also

MuPAD Functions

`break` | `case` | `for` | `repeat` | `return` | `while`

nextprime

Next prime number

Syntax

```
nextprime(m)
```

Description

`nextprime(m)` returns the smallest prime number larger than or equal to `m`.

If the argument `m` is an integer, then `nextprime` returns the smallest prime number larger than or equal to `m`. A symbolic call of type `"nextprime"` is returned, if the argument is not of type `Type::Numeric`. An error occurs if the argument is a number that is not an integer.

The first prime number is 2.

Examples

Example 1

The first prime number is computed:

```
nextprime(-13)
```

```
    2
```

If the argument of `nextprime` is a prime number, this number is returned:

```
nextprime(11)
```

```
    11
```

We compute a large prime:

```
nextprime(56475767478567)
```

$$56475767478601$$

Symbolic arguments lead to a symbolic call:

```
nextprime(x)
```

$$\text{nextprime}(x)$$

Parameters

m

An arithmetical expression

Return Values

Prime number or a symbolic call to `nextprime`.

Algorithms

`nextprime` uses a fast probabilistic prime number test (Miller-Rabin test) to decide if the computed result is a prime number. The result returned by `nextprime` is either a prime number or a strong pseudo-prime for 10 randomly chosen bases.

References

Michael O. Rabin, Probabilistic algorithms, in J. F. Traub, ed., *Algorithms and Complexity*, Academic Press, New York, 1976, pp. 21-39.

See Also

MuPAD Functions

`ifactor` | `igcd` | `ilcm` | `isprime` | `ithprime` | `prevprime`

NIL

Singleton element of the domain DOM_NIL

Syntax

```
NIL
```

Description

`NIL` is a keyword of the MuPAD language which represents the singleton element of the domain `DOM_NIL`.

The kernel domain `DOM_NIL` has only one singleton element. `NIL` is a keyword of the MuPAD language which represents this element. `NIL` is not changed by evaluation, see `DOM_NIL`.

Most often, `NIL` is used to represent a “missing” or “void” operand in a data structure. The “void object” returned by `null` is not suitable for this, because it is removed from most containers (like lists, sets or expressions) during evaluation.

When a new array from the kernel domain `DOM_ARRAY` is created, its elements are initialized with the value `NIL`. The function `op` returns `NIL` for un-initialized array elements. Note, however, that an indexed access of an un-initialized array element returns the indexed expression instead of `NIL`.

Local variables of procedures defined by `proc` are initialized with `NIL`. Nevertheless, a warning is printed if one accesses a local variable without explicitly initializing its value.

In former versions of MuPAD, `NIL` was used to delete values of identifiers or entries of tables, by assigning `NIL` to the identifier or entry. This is no longer supported. One must use `delete` to delete values. `NIL` now is a valid value of an identifier and a valid entry of a table.

Examples

Example 1

Unlike the "void object" returned by `null`, `NIL` is not removed from lists and sets:

```
[1, NIL, 2, NIL], [1, null(), 2, null()],
{1, NIL, 2, NIL}, {1, null(), 2, null()}
```

$$[1, \mathrm{NIL}, 2, \mathrm{NIL}], [1, 2], \{1, 2, \mathrm{NIL}\}, \{1, 2\}$$

Example 2

`NIL` is used to represent "missing" entries of procedures. For example, the simplest procedure imaginable has the following operands:

```
op(proc() begin end)
```

$$\mathrm{NIL}, \mathrm{NIL}, \mathrm{NIL}, \mathrm{NIL}, \mathrm{NIL}, \mathrm{NIL}, \mathrm{NIL}, \mathrm{NIL}, \mathrm{NIL}, \mathrm{NIL}, \mathrm{NIL}, \mathrm{NIL}, \mathrm{NIL}, \mathrm{NIL}, \mathrm{NIL}, \mathrm{NIL}$$

The first `NIL`, for example, represents the empty argument list, the second the void list of local variables and the third the void set of procedure options.

Example 3

Array elements are initialized with `NIL` if not defined otherwise. Note, however, that the indexed access for such elements yields the indexed expression:

```
A := array(1..2): A[1], op(A,1)
```

$$A_1, \mathrm{NIL}$$

```
delete A:
```

Example 4

Local variables in procedures are implicitly initialized with `NIL`. Still, a warning is printed if one uses the variable without explicitly initializing it:

```
p := proc() local l; begin print(l) end:  p():
```

```
Warning: Uninitialized variable 'l' is used.
  Evaluating: p
```

```
    NIL
```

```
delete p:
```

Example 5

NIL may be assigned to an identifier or indexed identifier like any other value. Such an assignment no longer deletes the value of the identifier:

```
a := NIL:  b[1] := NIL:  a, b[1]
```

```
    NIL, NIL
```

```
delete a, b:
```

See Also

MuPAD Functions
delete | FAIL | null

nops

Number of operands

Syntax

```
nops(object)
```

Description

`nops(object)` returns the number of operands of the object.

See the help page of `op` for details on the MuPAD concept of “operands”.

For sets, lists, and tables, the function `nops` returns the number of elements or entries, respectively. Note that expressions of type `DOM_EXPR`, arrays and hfarrays have a 0-th operand which is *not counted* by `nops`. For arrays, also non-initialized elements are counted by `nops`.

The void object `null()` of type `DOM_NULL`, the empty list`[ ]`, the empty set`{ }`, and the empty table`table()` have no operands: `nops` returns `0`. Cf. “Example 1” on page 1-1400.

Integers of domain type `DOM_INT`, real floating-point numbers of domain type `DOM_FLOAT`, Boolean constants of domain type `DOM_BOOL`, identifiers of domain type `DOM_IDENT`, and strings of domain type `DOM_STRING` are `atomic' objects having only 1 operand: the object itself. Rational numbers of domain type `DOM_RAT` and complex numbers of domain type `DOM_COMPLEX` have 2 operands: the numerator and denominator and the real part and imaginary part, respectively. Cf. “Example 2” on page 1-1401.

In contrast to most other MuPAD functions, `nops` does not flattenexpression sequences. Cf. “Example 3” on page 1-1402.

Examples

Example 1

The following expression has the type "_plus" and the three operands `a*b`, `3*c`, and `d`:

```
nops(a*b + 3*c + d)
```

```
3
```

For sets and lists, `nops` returns the number of elements. Note that the sublist `[1, 2, 3]` and the subset `{1, 2}` each count as one operand in the following examples:

```
nops({a, 1, [1, 2, 3], {1, 2}})
```

```
4
```

```
nops([[1, 2, 3], 4, 5, {1, 2}])
```

```
4
```

Empty objects have no operands:

```
nops(null()), nops([ ]), nops({}), nops(table())
```

```
0, 0, 0, 0
```

The number of operands of a symbolic function call is the number of arguments:

```
nops(f(3*x, 4, y + 2)), nops(f())
```

```
3, 0
```

Example 2

Integers and real floating-point numbers only have one operand:

```
nops(12), nops(1.41)
```

```
1, 1
```

The same holds true for strings; use `length` to query the length of a string:

```
nops("MuPAD"), length("MuPAD")
```

```
1, 5
```

The number of operands of a rational number or a complex number is 2, even if the real part is zero:

```
nops(-3/2), nops(1 + I), nops(2*I)
```

```
2, 2, 2
```

A function environment has 3 and a procedure has 16 operands:

```
nops(sin), nops(op(sin, 1))
```

```
3, 16
```

Example 3

Expression sequences are not flattened by nops:

```
nops((1, 2, 3))
```

```
3
```

In contrast to the previous call, the following command calls nops with three arguments:

```
nops(1, 2, 3)
```

```
Error: The number of arguments is incorrect. [nops]
```

Parameters

object

An arbitrary MuPAD object

Return Values

Nonnegative integer.

Overloaded By

object

See Also

MuPAD Functions
`extnops` | `extop` | `extsubsop` | `length` | `op` | `subsop`

norm

Compute the norm of a matrix, a vector, or a polynomial

Syntax

```
norm(M, <1 | 2 | Frobenius | Infinity | Spectral>)
norm(v, <Frobenius | Infinity | kv>)
norm(p, <kp>)
norm(f, <vars>, <kp>)
```

Description

`norm(M, kM)` computes the norm of index `kM` of the matrix `M`.

`norm(v, kv)` computes the norm of index `kv` of the vector `v`.

`norm(p, kp)` computes the norm of index `kp` of the polynomial `p`.

In MuPAD, there is no difference between matrices and vectors: a vector is a matrix of dimension $1 \times n$ or $n \times 1$, respectively.

For an $m \times n$ matrix `M` = (M_{ij}) with $min(m, n) > 1$, only the 1-norm (maximum column sum)

$$\|M\|_1 = \max\left(\sum_{i=1}^{m} \left|M_{i,j}\right|\right),\ j = 1, \ldots, n$$

,

the Frobenius norm

$$\|M\| = \sqrt{\sum_{i=1}^{m}\left(\sum_{j=1}^{n} \left|M_{ij}\right|^2\right)}$$

,

the spectral norm

$$\|M\|_2 = \sqrt{\phi},$$

where ϕ is the largest eigenvalue of $A^H A$ and the *infinity*-norm (maximum row sum)

$$\|M\|_\infty = \max\left(\sum_{j=1}^{n} |M_{1,j}|, \sum_{j=1}^{n} |M_{2,j}|, \ldots, \sum_{j=1}^{n} |M_{m,j}|\right)$$

can be computed. The 1-norm and the `Infinity`-norm are operator norms with respect to the corresponding norms on the vector spaces the matrix is acting upon.

For vectors $\mathsf{v} = (v_i)$, represented by matrices of dimension $1 \times n$ or $n \times 1$, norms with arbitrary positive integer indices k as well as `Infinity` can be computed. For integers $k > 1$, the vector norms are given by

$$\|v\|_k = \left(\sum_{i=1}^{n} |v_i|^k\right)^{1/k}$$

for column vectors as well as for row vectors.

For indices 1, `Infinity`, and `Frobenius`, the vector norms are given by the corresponding matrix norms. For column vectors, the 1-norm is the sum norm

$$\|v\|_1 = \sum_{i=1}^{n} |v_i|,$$

the `Infinity`-norm is the maximum norm

$$\|v\|_\infty = \max\left(|v_1|, \ldots, |v_n|\right)$$

(this is the limit of the k-norms as k tends to infinity).

Note: For row vectors, the 1-norm is the maximum norm, whilst the `Infinity`-norm is the sum norm.

The Frobenius norm coincides with `norm(v, 2)` for both column and row vectors.

Cf. “Example 2” on page 1-1408.

Matrices and vectors may contain symbolic entries. No internal float conversion is applied.

For matrix and vector norms, also refer to the help page of `Dom::Matrix` (note that the function `matrix` generates matrices of type `Dom::Matrix()`).

For polynomials `p` with coefficients c_i, the norms are given by

$$\|p\|_\infty = \max\left(|c_i|\right),\ \|p\|_k = \left(\sum_{i=1}^{n} |c_i|^k\right)^{1/k}.$$

Also multivariate polynomials are accepted by `norm`. The coefficients with respect to all indeterminates are taken into account.

For polynomials, only numerical norms can be computed. The coefficients of the polynomial must not contain symbolic parameters that cannot be converted to floating-point numbers. Coefficients containing symbolic numerical expressions such as `PI+1`, `sqrt(2)` etc. are accepted. Internally, they are converted to floating-point numbers. Cf. “Example 3” on page 1-1408.

For indices `k> 1`, `norm(p, k)` always returns a floating-point number. The 1-norm produces an exact result if all coefficients are integers or rational numbers. The *infinity*-norm `norm(p)` produces an exact result, if the coefficient of largest magnitude is an integer or a rational number. In all other cases, also the 1-norm and the *infinity*-norm produce floating-point numbers. Cf. “Example 3” on page 1-1408.

For polynomials over the coefficient ring `IntMod(m)`, `norm` produces an error.

If the coefficient ring of the polynomial is a domain, it must implement the method `"norm"`. This method must return the norm of the coefficients as a number or as a numerical expression that can be converted to a floating-point number via `float`. With the coefficient norms $\|c_i\|$, `norm(p)` computes the maximum norm $\max\left(\|c_1\|, \ldots, \|c_n\|\right)$; `norm(p, k)` computes $\left(\sum_{i=1}^{n} \|c_i\|^k\right)^{1/k}$.

A polynomial expression `f` is internally converted to the polynomial `poly(f)`. If a list of indeterminates is specified, the norm of the polynomial `poly(f, vars)` is computed.

For polynomials and polynomial expressions, the norms are computed by a function of the system kernel.

Examples

Example 1

We compute various norms of a 2×3 matrix:

```
M := matrix([[2, 5, 8], [-2, 3, 5]]):
norm(M) = norm(M, Infinity),
norm(M, 1),
norm(M, Frobenius),
norm(M, Spectral)
```

$$15 = 15,\ 13,\ \sqrt{131},\ \sqrt{\frac{\sqrt{13429}}{2} + \frac{131}{2}}$$

For matrices, `norm` produces exact symbolic results:

```
M := matrix([[2/3, 63, PI],[x, y, z]]):
norm(M)
```

$$\max\left(\pi + \frac{191}{3},\ |x| + |y| + |z|\right)$$

```
norm(M, 1)
```

$$\max\left(|x| + \frac{2}{3},\ |y| + 63,\ \pi + |z|\right)$$

```
norm(M, Frobenius)
```

$$\sqrt{|x|^2 + |y|^2 + |z|^2 + \pi^2 + \frac{35725}{9}}$$

```
delete M:
```

Example 2

A column vector `col` and a row vector `row` are considered:

```
col := matrix([x1, PI]): row := matrix([[x1, PI]]): col, row
```

$$\begin{pmatrix} x1 \\ \pi \end{pmatrix}, \begin{pmatrix} x1 & \pi \end{pmatrix}$$

```
norm(col, 2) = norm(row, 2)
```

$$\sqrt{|x1|^2 + \pi^2} = \sqrt{|x1|^2 + \pi^2}$$

```
norm(col, 3) = norm(row, 3)
```

$$(|x1|^3 + \pi^3)^{1/3} = (|x1|^3 + \pi^3)^{1/3}$$

Note that the norms of index 1 and `Infinity` have exchanged meanings for column and row vectors:

```
norm(col, 1) = norm(row, Infinity)
```

$$\pi + |x1| = \pi + |x1|$$

```
norm(col, Infinity) = norm(row, 1)
```

$$\max(|x1|, \pi) = \max(|x1|, \pi)$$

```
delete col, row:
```

Example 3

The norms of some polynomials are computed:

```
p := poly(3*x^3 + 4*x, [x]): norm(p), norm(p, 1)
```

$$4, 7$$

If the coefficients are not integers or rational numbers, automatic conversion to floating-point numbers occurs:

```
p := poly(3*x^3 + sqrt(2)*x + PI, [x]): norm(p), norm(p, 1)
```

```
3.141592654, 7.555806216
```

Floating point numbers are always produced for indices greater than 1:

```
p := poly(3*x^3 + 4*x + 1, [x]):
norm(p, 1), norm(p, 2), norm(p, 5), norm(p, 10), norm(p)
```

```
8, 5.099019514, 4.174686339, 4.021974513, 4
```

```
delete p:
```

Example 4

The norms of some polynomial expressions are computed:

```
norm(x^3 + 1, 1), norm(x^3 + 1, 2), norm(x^3 + PI)
```

```
2, 1.414213562, 1
```

The following call yields an error, because the expression is regarded as a polynomial in `x`. Consequently, symbolic coefficients $6\,y$ and $9\,y^2$ are found which are not accepted:

```
f := 6*x*y + 9*y^2 + 2: norm(f, [x])
```

```
Error: The argument is invalid. [norm]
```

As a bivariate polynomial with the indeterminates `x` and `y`, the coefficients are 6, 9, and 2. Now, norms can be computed:

```
norm(f, [x, y], 1), norm(f, [x, y], 2), norm(f, [x, y])
```

```
17, 11.0, 9
```

```
delete f:
```

Parameters

`M`

A matrix of domain type `Dom::Matrix(...)`

`v`

A vector (a 1-dimensional matrix)

`kv`

A positive integer as index of the vector norm.

`p`

A polynomial generated by `poly`

`f`

A polynomial expression

`vars`

A list of identifiers or indexed identifiers, interpreted as the indeterminates of `f`

`kp`

The index of the norm of the polynomial: a real number greater or equal than 1. If no index is specified, the maximum norm (of index infinity) is computed.

Options

`Frobenius`

Computes the Frobenius norm for vectors and matrices.

`Infinity`

Computes the Infinity norm for vectors and matrices.

`Spectral`

Computes the Spectral norm for matrices.

Return Values

Arithmetical expression.

Overloaded By

f, p

See Also

MuPAD Functions
`coeff` | `float` | `matrix` | `poly`

normal

Normalize an expression

Syntax

```
normal(f, options)
```

```
normal(object)
```

Description

`normal(f)` returns a normal form of the rational expression `f`. MuPAD regards an expression as normalized when it is a fraction where both numerator and denominator are polynomials whose greatest common divisor is 1.

`normal(object)` replaces the operands of `object` with their normalized form.

`normal` and `simplifyFraction` are equivalent.

If argument `f` contains irrational subexpressions such as `sin(x)`, `x^(-1/3)` etc., then these are replaced by auxiliary variables before normalization. After normalization, these variables are replaced by the normalization of the original subexpressions. Algebraic dependencies of the subexpressions are not taken into account. The operands of the non-rational subexpressions are normalized recursively.

If argument `f` contains floating-point numbers, then these are replaced by rational approximants (see `numeric::rationalize`). In the end, `float` is applied to the result.

With the `Expand` option, the normal form is unique for rational expressions: it is the quotient of expanded polynomials whose greatest common divisor is 1. If `f` and `g` are rational expressions, the following statements are equivalent:

- `f` and `g` are mathematically equivalent.
- `normal(f, Expand) = normal(g, Expand)`
- `normal(f - g, Expand) = 0`

A normal form generated without the `Expand` option (which is equivalent to `Expand = FALSE`) is the quotient of products of powers of expanded polynomials, where all

factors of the numerator and the denominator are coprime. MuPAD regards factorized expressions, such as $x\,(x + 1)$, and equivalent expanded expressions, such as $x^2 + x$, as normalized. Therefore, if you do not use `Expand`, there is no unique normal form of a rational expression.

If `f` and `g` are rational expressions, these statements are equivalent:

- `f` and `g` are mathematically equivalent.
- `normal(f - g) = 0`

For special objects, `normal` is automatically mapped to its operands. In particular, if `object` is a polynomial of domain type `DOM_POLY`, then its coefficients are normalized. Further, if `object` is a set, list, table or array, respectively, then `normal` is applied to all entries. Further, the left and right sides of equations (type `"_equal"`), inequalities (type `"_unequal"`), and relations (type `"_less"` or `"_leequal"`) are normalized. Further, the operands of ranges (type `"_range"`) are normalized automatically.

Examples

Example 1

Compute the normal form of some rational expressions:

```
normal(x^2 - (x + 1)*(x - 1))
```

$$1$$

```
normal((x^2 - 1)/(x + 1))
```

$$x - 1$$

```
normal(1/(x + 1) + 1/(y - 1))
```

$$\frac{x + y}{(x + 1)\,(y - 1)}$$

The following expression must be regarded as a rational expression in the "indeterminates" `y` and `sin(x)`:

```
normal(1/sin(x)^2 + y/sin(x))
```

$$\frac{y \sin(x) + 1}{\sin(x)^2}$$

Example 2

Normalize the entries of this list:

```
[(x^2 - 1)/(x + 1), x^2 - (x + 1)*(x - 1)]
```

$$\left[\frac{x^2 - 1}{x + 1}, x^2 - (x - 1)\,(x + 1)\right]$$

```
normal(%)
```

$$[x - 1,\ 1]$$

Now, normalize the coefficients of polynomials:

```
poly((x^2-1)/(x+1)*Y^2 + (x^2-(x+1)*(x-1))*Y - 1, [Y])
```

$$\mathrm{poly}\left(\frac{x^2 - 1}{x + 1}\,Y^2 + (-(x - 1)\,(x + 1) + x^2)\,Y - 1, [Y]\right)$$

```
normal(%)
```

$$\mathrm{poly}((x - 1)\,Y^2 + Y - 1, [Y])$$

Example 3

If you use the `Expand` option, `normal` returns a fraction with the expanded numerator and denominator:

```
normal(x/(x^6 - 1) + x^2/(x^4 - 1), Expand)
```

$$-\frac{x^6+x^4+x^3+x^2+x}{-x^8-x^6+x^2+1}$$

Without `Expand`, a fraction returned by `normal` can contain factored expressions:

```
normal(x/(x^6 - 1) + x^2/(x^4 - 1))
```

$$\frac{x\,(x^5+x^3+x^2+x+1)}{(x^2-1)\,(x^2+1)\,(x^4+x^2+1)}$$

Example 4

If you use the `List` option, `normal` returns a list consisting of the numerator and denominator of the input:

```
normal((x^2-1)/(x^2+2*x+1), List)
```

$$[x-1,\ x+1]$$

Note that `normal(f, List)` is *not* the same as `[numer(f), denom(f)]`:

```
[numer, denom]((x^2-1)/(x^2+2*x+1))
```

$$[x^2-1,\ x^2+2\,x+1]$$

Example 5

To skip calculation of common divisors of the numerator and denominator of an expression, use the `NoGcd` option:

```
y := (x^4 - 1)/(x + 1) + 1:
normal(y);
normal(y, NoGcd)
```

$$x^3-x^2+x$$

$$\frac{x^4 + x}{x + 1}$$

Example 6

To specify common divisors that you want to cancel out, use the `ToCancel` option:

```
y := (x^4 - 1)/(x^2 - 1):
normal(y, ToCancel = {x - 1})
```

$$\frac{x^3 + x^2 + x + 1}{x + 1}$$

Example 7

By default, `normal` calls the `rationalize` function in attempt to rationalize the input expression. You might speed up computations by using `Rationalize = None` in conjunction with the `Expand` option. This combination of options lets you skip investigating algebraic dependencies and, therefore, saves some time:

```
n := exp(u):
a := (n^2 + n)/(n + 1) + 1:
normal(a, Expand, Rationalize = None)
```

$$\frac{e^{2u} + 2\,e^{u} + 1}{e^{u} + 1}$$

Without `Rationalize = None`, MuPAD analyzes algebraic dependencies and returns this result:

```
normal(a, Expand)
```

$$e^{u} + 1$$

Example 8

Disable recursive calls to `normal` for subexpressions by using `Recursive = FALSE`:

```
y := sqrt((x^2 + 2*x + 1)/(x + 1)):
normal(y, Recursive = FALSE)
```

$$\sqrt{\frac{x^2+2x+1}{x+1}}$$

Example 9

Solve this equation, and sum up the fifth powers of the solutions:

```
solutions := solve(x^3 + x^2 + 1, x, MaxDegree = 3):
f := _plus((solutions[i]^7) $i = 1..3)
```

$$-\left(\frac{1}{3}-\frac{\sigma_1}{2}-\frac{1}{18\,\sigma_1}+\frac{\sqrt{3}\left(\frac{1}{9\,\sigma_1}-\sigma_1\right)i}{2}\right)^7+\left(\frac{1}{18\,\sigma_1}+\frac{\sigma_1}{2}-\frac{1}{3}+\frac{\sqrt{3}\left(\frac{1}{9\,\sigma_1}-\sigma_1\right)i}{2}\right)^7$$

$$-\left(\frac{1}{9\,\sigma_1}+\sigma_1+\frac{1}{3}\right)^7$$

where

$$\sigma_1=\left(\frac{29}{54}-\frac{\sqrt{31}\,\sqrt{108}}{108}\right)^{1/3}$$

Normalizing the result returns:

```
normal(f)
```

$$-15$$

To limit the number of internally repeated calls to `normal` due to analysis of algebraic dependencies, use the `Iterations` option. The default number of iterations is 5. Use the `Iterations` option to increase or decrease the number of iterations. For example, normalize the result using just one iteration:

```
normal(f, Iterations = 1)
```

$$-\frac{\frac{181875\sqrt{31}\sqrt{108}}{4}+3168963\left(\frac{\sqrt{31}\sqrt{108}}{108}-\frac{29}{54}\right)^3-\frac{5261843}{2}}{531441\left(\frac{29\sqrt{31}\sqrt{108}}{2916}-\frac{839}{1458}\right)}$$

After two iterations, the result becomes shorter:

```
normal(f, Iterations = 2)
```

$$-\frac{5073840\sqrt{31}\sqrt{108}-293582880}{11664\,(29\sqrt{31}\sqrt{108}-1678)}$$

After three iterations, you get the simplest result:

```
normal(f, Iterations = 3)
```

$$-15$$

Parameters

`f`

An arithmetical expression

`object`

A polynomial of type `DOM_POLY`, list, set, table, array, equation, inequality, or range

Options

`Expand`

Return the numerator and denominator of the normalized expression in expanded form. See “Details” for more information. By default, `Expand = FALSE`.

`List`

Return a list consisting of the numerator and denominator of `f`. By default, `List = FALSE`.

`NoGcd`

Skip computing common divisors of the numerator and denominator of `f`. By default, `NoGcd = FALSE`.

`ToCancel`

Option, specified as `ToCancel = {expr1, expr2, …}`

Cancel out only the specified common divisors `{expr1, expr2,...}`.

`Rationalize`

Option, specified as `Rationalize = None`

Perform only basic rationalization of an irrational input expression. Skip investigating algebraic dependencies. This option works only in conjunction with the `Expand` option. Otherwise, `normal` ignores this option. See “Example 7” on page 1-1416.

`Recursive`

Recursively normalize subexpressions of an irrational expression. By default, `Recursive = TRUE`.

`Iterations`

Option, specified as `Iterations = n`

Specify the number of repeated calls to `normal`. Repeated calls appear when analysis of algebraic dependencies results in new irrational subexpressions. By default, `n = 5`.

Return Values

Object of the same type as the input object, or a list of two arithmetical expressions if the `List` option is used.

Overloaded By

object

See Also

MuPAD Functions

`collect` | `combine` | `denom` | `expand` | `factor` | `gcd` | `indets` | `numer` | `partfrac` | `rationalize` | `rectform` | `rewrite` | `simplify` | `simplifyFraction`

More About

- "Manipulate Expressions"
- "Choose Simplification Functions"

simplifyFraction

Normalize an expression

Syntax

```
simplifyFraction(f, options)
```

```
simplifyFraction(object)
```

Description

`simplifyFraction(f)` returns a normal form of the rational expression `f`. MuPAD regards an expression as normalized when it is a fraction where both numerator and denominator are polynomials whose greatest common divisor is 1.

`simplifyFraction(object)` replaces the operands of `object` with their normalized form.

`normal` and `simplifyFraction` are equivalent.

If argument `f` contains irrational subexpressions such as `sin(x)`, `x^(-1/3)` etc., then these are replaced by auxiliary variables before normalization. After normalization, these variables are replaced by the normalization of the original subexpressions. Algebraic dependencies of the subexpressions are not taken into account. The operands of the non-rational subexpressions are normalized recursively.

If argument `f` contains floating-point numbers, then these are replaced by rational approximants (see `numeric::rationalize`). In the end, `float` is applied to the result.

With the `Expand` option, the normal form is unique for rational expressions: it is the quotient of expanded polynomials whose greatest common divisor is 1. If `f` and `g` are rational expressions, the following statements are equivalent:

- `f` and `g` are mathematically equivalent.
- `normal(f, Expand) = normal(g, Expand)`
- `normal(f - g, Expand) = 0`

A normal form generated without the `Expand` option (which is equivalent to `Expand = FALSE`) is the quotient of products of powers of expanded polynomials, where all factors of the numerator and the denominator are coprime. MuPAD regards factorized expressions, such as $x\,(x + 1)$, and equivalent expanded expressions, such as $x^2 + x$, as normalized. Therefore, if you do not use `Expand`, there is no unique normal form of a rational expression.

If `f` and `g` are rational expressions, these statements are equivalent:

- `f` and `g` are mathematically equivalent.
- `normal(f - g) = 0`

For special objects, `normal` is automatically mapped to its operands. In particular, if `object` is a polynomial of domain type `DOM_POLY`, then its coefficients are normalized. Further, if `object` is a set, list, table or array, respectively, then `normal` is applied to all entries. Further, the left and right sides of equations (type `"_equal"`), inequalities (type `"_unequal"`), and relations (type `"_less"` or `"_leequal"`) are normalized. Further, the operands of ranges (type `"_range"`) are normalized automatically.

Examples

Example 1

Compute the normal form of some rational expressions:

```
normal(x^2 - (x + 1)*(x - 1))
```

$$1$$

```
normal((x^2 - 1)/(x + 1))
```

$$x - 1$$

```
normal(1/(x + 1) + 1/(y - 1))
```

$$\frac{x + y}{(x + 1)\,(y - 1)}$$

The following expression must be regarded as a rational expression in the "indeterminates" `y` and `sin(x)`:

```
normal(1/sin(x)^2 + y/sin(x))
```

$$\frac{y\,\sin(x)+1}{\sin(x)^2}$$

Example 2

Normalize the entries of this list:

```
[(x^2 - 1)/(x + 1), x^2 - (x + 1)*(x - 1)]
```

$$\left[\frac{x^2-1}{x+1},\ x^2-(x-1)\,(x+1)\right]$$

```
normal(%)
```

$$[x-1,\ 1]$$

Now, normalize the coefficients of polynomials:

```
poly((x^2-1)/(x+1)*Y^2 + (x^2-(x+1)*(x-1))*Y - 1, [Y])
```

$$\mathrm{poly}\left(\frac{x^2-1}{x+1}\,Y^2+(-(x-1)\,(x+1)+x^2)\,Y-1,\ [Y]\right)$$

```
normal(%)
```

$$\mathrm{poly}((x-1)\,Y^2+Y-1,\ [Y])$$

Example 3

If you use the `Expand` option, `normal` returns a fraction with the expanded numerator and denominator:

```
normal(x/(x^6 - 1) + x^2/(x^4 - 1), Expand)
```

$$-\frac{x^6+x^4+x^3+x^2+x}{-x^8-x^6+x^2+1}$$

Without `Expand`, a fraction returned by `normal` can contain factored expressions:

```
normal(x/(x^6 - 1) + x^2/(x^4 - 1))
```

$$\frac{x\,(x^5+x^3+x^2+x+1)}{(x^2-1)\,(x^2+1)\,(x^4+x^2+1)}$$

Example 4

If you use the `List` option, `normal` returns a list consisting of the numerator and denominator of the input:

```
normal((x^2-1)/(x^2+2*x+1), List)
```

$$[x-1,\ x+1]$$

Note that `normal(f, List)` is *not* the same as `[numer(f), denom(f)]`:

```
[numer, denom]((x^2-1)/(x^2+2*x+1))
```

$$[x^2-1,\ x^2+2\,x+1]$$

Example 5

To skip calculation of common divisors of the numerator and denominator of an expression, use the `NoGcd` option:

```
y := (x^4 - 1)/(x + 1) + 1:
normal(y);
normal(y, NoGcd)
```

$$x^3-x^2+x$$

$$\frac{x^4 + x}{x + 1}$$

Example 6

To specify common divisors that you want to cancel out, use the `ToCancel` option:

```
y := (x^4 - 1)/(x^2 - 1):
normal(y, ToCancel = {x - 1})
```

$$\frac{x^3 + x^2 + x + 1}{x + 1}$$

Example 7

By default, `normal` calls the `rationalize` function in attempt to rationalize the input expression. You might speed up computations by using `Rationalize = None` in conjunction with the `Expand` option. This combination of options lets you skip investigating algebraic dependencies and, therefore, saves some time:

```
n := exp(u):
a := (n^2 + n)/(n + 1) + 1:
normal(a, Expand, Rationalize = None)
```

$$\frac{e^{2u} + 2\,e^{u} + 1}{e^{u} + 1}$$

Without `Rationalize = None`, MuPAD analyzes algebraic dependencies and returns this result:

```
normal(a, Expand)
```

$$e^{u} + 1$$

Example 8

Disable recursive calls to `normal` for subexpressions by using `Recursive = FALSE`:

```
y := sqrt((x^2 + 2*x + 1)/(x + 1)):
normal(y, Recursive = FALSE)
```

$$\sqrt{\frac{x^2+2\,x+1}{x+1}}$$

Example 9

Solve this equation, and sum up the fifth powers of the solutions:

```
solutions := solve(x^3 + x^2 + 1, x, MaxDegree = 3):
f := _plus((solutions[i]^7) $i = 1..3)
```

$$-\left(\frac{1}{3}-\frac{\sigma_1}{2}-\frac{1}{18\,\sigma_1}+\frac{\sqrt{3}\left(\frac{1}{9\,\sigma_1}-\sigma_1\right)i}{2}\right)^7+\left(\frac{1}{18\,\sigma_1}+\frac{\sigma_1}{2}-\frac{1}{3}+\frac{\sqrt{3}\left(\frac{1}{9\,\sigma_1}-\sigma_1\right)i}{2}\right)^7$$

$$-\left(\frac{1}{9\,\sigma_1}+\sigma_1+\frac{1}{3}\right)^7$$

where

$$\sigma_1=\left(\frac{29}{54}-\frac{\sqrt{31}\,\sqrt{108}}{108}\right)^{1/3}$$

Normalizing the result returns:

```
normal(f)
```

$$-15$$

To limit the number of internally repeated calls to `normal` due to analysis of algebraic dependencies, use the `Iterations` option. The default number of iterations is 5. Use the `Iterations` option to increase or decrease the number of iterations. For example, normalize the result using just one iteration:

```
normal(f, Iterations = 1)
```

$$-\frac{\frac{181875\sqrt{31}\sqrt{108}}{4}+3168963\left(\frac{\sqrt{31}\sqrt{108}}{108}-\frac{29}{54}\right)^3-\frac{5261843}{2}}{531441\left(\frac{29\sqrt{31}\sqrt{108}}{2916}-\frac{839}{1458}\right)}$$

After two iterations, the result becomes shorter:

```
normal(f, Iterations = 2)
```

$$-\frac{5073840\sqrt{31}\sqrt{108}-293582880}{11664\,(29\sqrt{31}\sqrt{108}-1678)}$$

After three iterations, you get the simplest result:

```
normal(f, Iterations = 3)
```

$$-15$$

Parameters

`f`

An arithmetical expression

`object`

A polynomial of type `DOM_POLY`, list, set, table, array, equation, inequality, or range

Options

`Expand`

Return the numerator and denominator of the normalized expression in expanded form. See “Details” for more information. By default, `Expand = FALSE`.

`List`

Return a list consisting of the numerator and denominator of `f`. By default, `List = FALSE`.

NoGcd

Skip computing common divisors of the numerator and denominator of `f`. By default, `NoGcd = FALSE`.

ToCancel

Option, specified as `ToCancel = {expr1, expr2, …}`

Cancel out only the specified common divisors `{expr1, expr2,...}`.

Rationalize

Option, specified as `Rationalize = None`

Perform only basic rationalization of an irrational input expression. Skip investigating algebraic dependencies. This option works only in conjunction with the `Expand` option. Otherwise, `normal` ignores this option. See "Example 7" on page 1-1425.

Recursive

Recursively normalize subexpressions of an irrational expression. By default, `Recursive = TRUE`.

Iterations

Option, specified as `Iterations = n`

Specify the number of repeated calls to `normal`. Repeated calls appear when analysis of algebraic dependencies results in new irrational subexpressions. By default, `n = 5`.

Return Values

Object of the same type as the input object, or a list of two arithmetical expressions if the `List` option is used.

Overloaded By

object

See Also

MuPAD Functions

`collect` | `combine` | `denom` | `expand` | `factor` | `gcd` | `indets` | `normal` | `numer` | `partfrac` | `rationalize` | `rectform` | `rewrite` | `simplify`

More About

- "Manipulate Expressions"
- "Choose Simplification Functions"

NOTEBOOKFILE

Notebook file name

Description

The environment variables NOTEBOOKFILE and NOTEBOOKPATH store the absolute file name and the directory name, respectively, of the current notebook in the MuPAD Notebook app as a string.

Possible values: String

The environment variable NOTEBOOKFILE stores the name of the current notebook that is connected to the MuPAD kernel.

The environment variable NOTEBOOKPATH stores the name of the directory where the current notebook is located.

These variables are useful, for example, when reading files that are located relative to the notebook.

Both variables only have a value if the notebook has a name, which is generally the case when an existing notebook has been opened or a new notebook has been saved.

The name given by NOTEBOOKFILE is an absolute file name.

Both variables are read-only and are write-protected. One cannot assign a new value to NOTEBOOKFILE in order to change the name of the notebook.

NOTEBOOKFILE and NOTEBOOKPATH are only defined in the MuPAD Notebook app. When using the MuPAD engine from MATLAB, the two variables are just normal identifiers.

Examples

Example 1

In the MuPAD Notebook app, you can specify startup commands for a notebook, which are executed when the notebook is connected to a kernel.

In the startup commands you can use NOTEBOOKPATH to read a source file "my_init.mu" which is stored in the directory of the notebook:

```
fread(NOTEBOOKPATH."my_init.mu")
```

See Also

MuPAD Functions
NOTEBOOKPATH | READPATH | WRITEPATH

NOTEBOOKPATH

Notebook path

Description

The environment variables NOTEBOOKFILE and NOTEBOOKPATH store the absolute file name and the directory name, respectively, of the current notebook in the MuPAD Notebook app as a string.

Possible values: String

The environment variable NOTEBOOKFILE stores the name of the current notebook that is connected to the MuPAD kernel.

The environment variable NOTEBOOKPATH stores the name of the directory where the current notebook is located.

These variables are useful, for example, when reading files that are located relative to the notebook.

Both variables only have a value if the notebook has a name, which is generally the case when an existing notebook has been opened or a new notebook has been saved.

The name given by NOTEBOOKFILE is an absolute file name.

Both variables are read-only and are write-protected. One cannot assign a new value to NOTEBOOKFILE in order to change the name of the notebook.

NOTEBOOKFILE and NOTEBOOKPATH are only defined in the MuPAD Notebook app. When using the MuPAD engine from MATLAB, the two variables are just normal identifiers.

Examples

Example 1

In the MuPAD Notebook app, you can specify startup commands for a notebook, which are executed when the notebook is connected to a kernel.

In the startup commands. you can use NOTEBOOKPATH to read a source file "my_init.mu" which is stored in the directory of the notebook:

```
fread(NOTEBOOKPATH."my_init.mu")
```

See Also

MuPAD Functions
NOTEBOOKFILE | READPATH | WRITEPATH

nterms

Number of terms of a polynomial

Syntax

```
nterms(p)
```

```
nterms(f, <vars>)
```

Description

`nterms(p)` returns the number of terms of the polynomial `p`.

If the first argument `f` is not element of a polynomial domain, then `nterms` converts the expression to a polynomial via `poly(f)`. If a list of indeterminates is specified, then the polynomial `poly(f, vars)` is considered.

A zero polynomial has no terms: the return value is 0.

Examples

Example 1

We give some self explaining examples:

```
nterms(x^2*y^2 + x^2 + y + 2, [x, y])
```

```
    4
```

```
nterms(poly(x^2*y^2 + x^2 + y + 2))
```

```
    4
```

```
nterms(poly(0, [x]))
```

```
0
```

Example 2

The following polynomial expression may be regarded as a polynomial in different ways:

```
f := x^2*y^2 + x^2 + y + 2:
nterms(f, [x]), nterms(f, [y]), nterms(f, [x, y]),
nterms(f, [z])
```

```
2, 3, 4, 1
```

```
delete f:
```

Parameters

`p`

A polynomial of type `DOM_POLY`

`f`

A polynomial expression

`vars`

A list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

Return Values

Nonnegative number. `FAIL` is returned if the input cannot be converted to a polynomial.

Overloaded By

p

See Also

MuPAD Functions

`coeff` | `degree` | `degreevec` | `ground` | `lcoeff` | `ldegree` | `lmonomial` | `lterm` | `monomials` | `nthcoeff` | `nthmonomial` | `nthterm` | `poly` | `poly2list` | `tcoeff`

nthcoeff

N-th non-zero coefficient of a polynomial

Syntax

```
nthcoeff(p, n)
```

```
nthcoeff(f, <vars>, n)
```

Description

`nthcoeff(p, n)` returns the `n`-th non-zero coefficient of the polynomial `p`.

`nthcoeff` returns the `n`-th non-zero coefficient with respect to the lexicographical ordering.

The "first" coefficient is the leading coefficient as returned by `lcoeff`, the "last" coefficient is the trailing coefficient as returned by `tcoeff`.

A zero polynomial has no terms: `nthcoeff` returns `FAIL`.

A polynomial expression `f` is first converted to a polynomial with the variables given by `vars`. If no variables are given, they are searched for in `f`. See `poly` about details of the conversion. `FAIL` is returned if `f` cannot be converted to a polynomial.

The result of `nthcoeff` is not fully evaluated. Evaluation can be enforced by the function `eval`. Cf. "Example 2" on page 1-1438.

Examples

Example 1

We give some self explaining examples:

```
p := poly(100*x^100 + 49*x^49 + 7*x^7, [x]):
nthcoeff(p, 1), nthcoeff(p, 2), nthcoeff(p, 3)
```

```
     100, 49, 7

nthcoeff(p, 4)

     FAIL

nthcoeff(poly(0, [x]), 1)

     FAIL

delete p:
```

Example 2

We demonstrate the evaluation strategy of `nthcoeff`:

```
p := poly(3*x^3 + 6*x^2*y^2 + 2, [x]): y := 4:
nthcoeff(p, 2)
```

$$6\,y^2$$

Evaluation is enforced by `eval`:

```
eval(%)

     96

delete p, y:
```

Parameters

p

A polynomial of type `DOM_POLY`

f

A polynomial expression

`vars`

A list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

`n`

A positive integer

Return Values

Element of the coefficient domain of the polynomial. An expression is returned if a polynomial expression is given as input. `FAIL` is returned if `n` is larger than the actual number of terms.

Overloaded By

p

See Also

MuPAD Functions

`coeff` | `collect` | `degree` | `degreevec` | `ground` | `lcoeff` | `ldegree` | `lmonomial` | `lterm` | `monomials` | `nterms` | `nthmonomial` | `nthterm` | `poly` | `poly2list` | `tcoeff`

nthmonomial

N-th monomial of a polynomial

Syntax

```
nthmonomial(p, n)
```

```
nthmonomial(f, <vars>, n)
```

Description

`nthmonomial(p, n)` returns the `n`-th non-trivial monomial of the polynomial `p`.

`nthmonomial` returns the `n`-th non-trivial monomial with respect to the lexicographical ordering.

The "first" monomial is the leading monomial as returned by `lmonomial`.

A zero polynomial has no terms: `nthmonomial` returns `FAIL`.

A polynomial expression `f` is first converted to a polynomial with the variables given by `vars`. If no variables are given, they are searched for in `f`. See `poly` about details of the conversion. The result is returned as polynomial expression. `FAIL` is returned if `f` cannot be converted to a polynomial.

The result of `nthmonomial` is not fully evaluated. It can be evaluated by the functions `mapcoeffs` and `eval`. Cf. "Example 2" on page 1-1441.

Examples

Example 1

We give some self explaining examples:

```
p := poly(100*x^100 + 49*x^49 + 7*x^7, [x]):
nthmonomial(p, 1), nthmonomial(p, 2), nthmonomial(p, 3)
```

$$\text{poly}(100\,x^{100}, [x]),\ \text{poly}(49\,x^{49}, [x]),\ \text{poly}(7\,x^{7}, [x])$$

```
nthmonomial(p, 4)
```

$$\text{FAIL}$$

```
nthmonomial(poly(0, [x]), 1)
```

$$\text{FAIL}$$

```
delete p:
```

Example 2

We demonstrate the evaluation strategy of `nthmonomial`:

```
p := poly(3*x^3 + 6*x^2*y^2 + 2, [x]): y := 4:
nthmonomial(p, 2)
```

$$\text{poly}((6\,y^{2})\,x^{2}, [x])$$

Evaluation is enforced by `eval`:

```
mapcoeffs(%, eval)
```

$$\text{poly}(96\,x^{2}, [x])$$

```
delete p, y:
```

Parameters

p

A polynomial of type `DOM_POLY`

f

A polynomial expression

vars

A list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

n

A positive integer

Return Values

Polynomial of the same type as p. An expression is returned if a polynomial expression is given as input. FAIL is returned if n is larger than the actual number of terms of the polynomial.

Overloaded By

p

See Also

MuPAD Functions

coeff | degree | degreevec | ground | lcoeff | ldegree | lmonomial | lterm | monomials | nterms | nthcoeff | nthterm | poly | poly2list | tcoeff

nthterm

N-th term of a polynomial

Syntax

```
nthterm(p, n)
```

```
nthterm(f, <vars>, n)
```

Description

`nthterm(p, n)` returns the `n`-th non-zero term of the polynomial`p`.

`nthterm` returns the `n`-th non-zero term with respect to the lexicographical ordering.

The “first” term is the leading term as returned by `lterm`.

A zero polynomial has no terms: `nthterm` returns `FAIL`.

The identity *nthterm*(*p*, *n*) *nthcoeff*(*p*, *n*) = *nthmonomial*(*p*, *n*) holds.

A polynomial expression `f` is first converted to a polynomial with the variables given by `vars`. If no variables are given, they are searched for in `f`. See `poly` about details of the conversion. The result is returned as polynomial expression. `FAIL` is returned if `f` cannot be converted to a polynomial.

Examples

Example 1

We give some self explaining examples:

```
p := poly(100*x^100 + 49*x^49 + 7*x^7, [x]):
nthterm(p, 1), nthterm(p, 2), nthterm(p, 3)
```

$$\mathrm{poly}(x^{100}, [x]), \mathrm{poly}(x^{49}, [x]), \mathrm{poly}(x^{7}, [x])$$

```
nthterm(p, 4)
```

FAIL

```
nthterm(poly(0, [x]), 1)
```

FAIL

```
delete p:
```

Example 2

The n-th monomial is the product of the n-th coefficient and the n-th term:

```
p := poly(2*x^2*y + 3*x*y^2 + 6, [x, y]):
mapcoeffs(nthterm(p, 2), nthcoeff(p, 2)) =
nthmonomial(p, 2)
```

$$\mathrm{poly}(3\,x\,y^2, [x, y]) = \mathrm{poly}(3\,x\,y^2, [x, y])$$

```
delete p:
```

Parameters

p

A polynomial of type DOM_POLY

f

A polynomial expression

vars

A list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

n

A positive integer

Return Values

Polynomial of the same type as `p`. An expression is returned if a polynomial expression is given as input. `FAIL` is returned if `n` is larger than the actual number of terms of the polynomial.

Overloaded By

p

See Also

MuPAD Functions

`coeff` | `degree` | `degreevec` | `ground` | `lcoeff` | `ldegree` | `lmonomial` | `lterm` | `monomials` | `nterms` | `nthcoeff` | `nthmonomial` | `poly` | `poly2list` | `tcoeff`

null

Generate the void object of type DOM_NULL

Syntax

```
null()
```

Description

`null()` returns the void object of domain type `DOM_NULL`. It represents an empty sequence of MuPAD expressions or statements.

The void object does not produce any output on the screen.

Various systems functions such as `print` or `reset` return the void object.

The void object is removed from sequences (“flattening”). It can be used to remove elements from lists or sets. Cf. “Example 2” on page 1-1447.

Examples

Example 1

`null()` returns the void object which does not produce any screen output:

```
null()
```

The resulting object is of domain type `DOM_NULL`:

```
domtype(null())
```

```
DOM_NULL
```

This object represents the empty expression sequence and the empty statement sequence:

```
domtype(_exprseq()), domtype(_stmtseq())
```

DOM_NULL, DOM_NULL

Some system functions such as `print` return the void object:

```
print("Hello world!"):
```

"Hello world!"

```
domtype(%)
```

DOM_NULL

Example 2

The void object is removed from lists, sets, and expression sequences:

```
[null(), a, b, null(), c], {null(), a, b, null(), c},
f(null(), a, b, null(), c)
```

$[a, b, c], \{a, b, c\}, f(a, b, c)$

```
a + null() + b = _plus(a, null(), b)
```

$a + b = a + b$

```
subsop([a, x, b], 2 = null()), subs({a, x, b}, x = null())
```

$[a, b], \{a, b\}$

However, `null()` is a valid entry in arrays and tables:

```
a := array(1..2): a[1] := 1: a[2] := null(): a
```

(1 null())

```
domtype(a[1]), domtype(a[2])
```

DOM_INT, DOM_NULL

```
t := table(null() = "void", 1 = 2.5, b = null())
```

1	2.5
b	null()
null()	"void"

```
domtype(t[b]), t[]
```

DOM_NULL, "void"

```
delete a, t:
```

Example 3

The void object remains if you delete all elements from an expression sequence:

```
a := (1, b): delete a[1]: delete a[1]: domtype(a)
```

DOM_NULL

The operand function `op` returns the void object when applied to an object with no operands:

```
domtype(op([])), domtype(op({})), domtype(op(f()))
```

DOM_NULL, DOM_NULL, DOM_NULL

```
delete a:
```

Return Values

Void object of domain type `DOM_NULL`.

See Also

MuPAD Functions
`_exprseq` | `_stmtseq` | `FAIL` | `NIL`

numer

Numerator of a rational expression

Syntax

```
numer(f)
```

Description

`numer(f)` returns the numerator of the expression `f`.

`numer` regards the input as a rational expression: non-rational subexpressions such as `sin(x)`, `x^(1/2)` etc. are internally replaced by "temporary variables". The numerator of this rationalized expression is computed, the temporary variables are finally replaced by the original subexpressions.

Note: Numerator and denominator are not necessarily cancelled: the numerator returned by `numer` may have a non-trivial `gcd` with the denominator returned by `denom`. Preprocess the expression by `normal` to enforce cancellation of common factors. Cf. "Example 2" on page 1-1451.

Examples

Example 1

We compute the numerators of some expressions:

```
numer(-3/4)
```

```
     -3
```

```
numer(x + 1/(2/3*x -2/x))
```

$$x\,(2\,x^2-3)$$

```
numer((cos(x)^2 -1)/(cos(x) -1))
```

$$\cos(x)^2-1$$

Example 2

`numer` performs no cancellations if the rational expression is of the form "numerator/denominator":

```
r := (x^2 - 1)/(x^3 - x^2 + x - 1): numer(r)
```

$$x^2-1$$

This numerator has a common factor with the denominator of `r`; `normal` enforces cancellation of common factors:

```
numer(normal(r))
```

$$x+1$$

However, automatic normalization occurs if the input expression is a sum:

```
numer(r + x/(x + 1) + 1/(x + 1) - 1)
```

$$x+1$$

```
delete r:
```

Parameters

`f`

An arithmetical expression

Return Values

Arithmetical expression.

Overloaded By

f

See Also

MuPAD Functions
`denom` | `factor` | `gcd` | `normal`

O

Domain of order terms (Landau symbols)

Syntax

```
O(f, <x = x0, y = y0, …>)
```

Description

`O(f, x = x0)` represents the Landau symbol $O(f, x \to x_0)$.

Mathematically, for a function f in the variables $(x, y, \ldots)$, the Landau symbol

$$g := O(f, x \to x_0, y \to y_0, \ldots)$$

is a function in these variables with the following property: there exists a constant c and a neighborhood of the limit point $(x_0, y_0, \ldots)$ such that $|g| \le c\,|f|$ for all values $(x, y, \ldots)$ in that neighborhood.

Note: Typically, Landau symbols are used to denote the order terms ("error terms") of series expansions. Note, however, that the series expansions produced by `asympt`, `series`, and `taylor` represent order terms as a part of the data structures `Series::Puiseux` and `Series::gseries`; they do *not* use the domain `O`.

With the equations `x = x0`, `y = y0` etc., `f` is regarded as a function of the specified variables. All other identifiers contained in `f` are regarded as constant parameters.

If no variables and limit points are specified, then all identifiers in `f` are used as variables, each tending to the default limit point 0.

Variables tending to 0 are not printed on the screen.

The variables of an order term may be obtained with the function `indets`. The limit points may be queried with the function `O::points`.

The arithmetical operations +, -, *, /, and ^ are overloaded for order terms.

Automatic simplifications are currently restricted to polynomial expressions f. Univariate polynomial expressions are reduced to the leading monomial of the expansion around the limit point. In multivariate polynomial expressions, all terms are discarded that are divisible by lower order terms. For non-polynomial expressions, only integer factors are removed.

Examples

Example 1

For polynomial expressions, certain simplifications occur:

```
O(x^4 + 2*x^2), O(7*x^3), O(x, x = 1)
```

$$O(x^2),\ O(x^3),\ O(1, x = 1)$$

A zero limit point is not printed on the screen:

```
O(1), O(1, x = 1), O(x^2/(y + 1), x = 0, y = -1, z = PI)
```

$$O(1),\ O(1, x = 1),\ O\left(\frac{x^2}{y+1}, y = -1, z = \pi\right)$$

The arithmetical operations are overloaded for order terms:

```
7*O(x), O(x^2) + O(x^13), O(x^3) - O(x^3), O(x^2)^2 + O(x^4)
```

$$O(x),\ O(x^2),\ O(x^3),\ O(x^4)$$

Example 2

For multivariate polynomial expression, higher order terms are discarded if they are divisible by lower order terms:

```
O(15*x*y^2 + 3*x^2*y + x^2*y^2)
```

$$O(x^2\,y + 5\,x\,y^2)$$

```
O(x + x^2*y) = O(x)*O(1 + x*y)
```

$$O(x) = O(x)$$

Example 3

We demonstrate how to access the variables and the limit points of an order term:

```
a := O(x^2*y^2)
```

$$O(x^2\,y^2)$$

```
indets(a) = O::indets(a), O::points(a)
```

$$\{x, y\} = \{x, y\},\ \{x = 0, y = 0\}$$

```
delete a:
```

Parameters

`f`

An arithmetical expression representing a function in `x`, `y` etc.

`x, y, …`

The variables: identifiers

`x0, y0, …`

The limit points: arithmetical expressions

Return Values

Element of the domain `O`.

See Also

MuPAD Functions
`asympt` | `limit` | `series` | `taylor`

ode

Domain of ordinary differential equations

Syntax

```
ode(eq, y(x))

ode({eq, <inits>}, y(x))

ode({eq1, eq2, , …, <inits>}, {y1(x), y2(x), …})
```

Description

`ode(eq, y(x))` represents an ordinary differential equation (ODE) for the function `y(x)`.

`ode({eq1, eq2, ...}, {y1(x), y2(x), ...})` represents a system of ODEs for the functions `y1(x)`, `y2(x)` etc.

In the equations `eq`, `eq1` etc., the unknown functions must be represented by `y(x)`, `y1(x)` etc. Derivatives may be represented either by the `diff` function or by the differential operator `D`. Note that the token `'` provides a handy shortcut: `y'(x) = D(y)(x)` means the same as `diff(y(x), x)`.

The unknown functions must be univariate in the independent variable `x`. Multivariate expressions such as `y(x, t)` are not accepted.

The `ode` function does not accept `piecewise` input.

Initial and boundary conditions are defined by sequences of equations involving the unknown functions or their derivatives on the left hand side. The corresponding values must be specified on the right hand side of the equations. In particular, the differential operator `D` (or the token `'`) must be used to specify values of derivatives at some point. E.g.,

$$\{\{y(1) = 2,\ y'(0) = 0,\ y''(0) = 1\}\}$$

is a valid sequence of boundary conditions for `inits`.

Boundary conditions of the first and second kind are allowed. Mixed conditions are not accepted.

The initial/boundary points and the corresponding initial/boundary values may be symbolic expressions.

For scalar initial value or boundary value problems, use `ode({eq, inits}, y(x))` to specify the conditions.

For systems of ODEs, there must be as many equations as unknown functions.

The main purpose of the `ode` domain is to provide an environment for overloading the function `solve`.

In the case of one single equation (possibly together with initial or boundary conditions), `solve` returns a set of explicit solutions or an implicit solution. Each element of the set represents a solution branch.

In the case of a system of equations, `solve` returns a set of lists of equations for the unknown functions. Each list represents a solution branch.

An symbolic `solve` call is returned if no solution is found.

Examples

Example 1

In the following, we show how to create and solve a scalar ODE. First, we define the ODE $x^2\,y'(x) + 3\,x\,y(x) = \frac{\sin(x)}{x}$. We use the quote token `'` to represent derivatives:

```
eq := ode(x^2*y'(x) + 3*x*y(x) = sin(x)/x, y(x))
```

$$\mathrm{ode}\left(x^2\,y'(x) - \frac{\sin(x)}{x} + 3\,x\,y(x),\ y(x)\right)$$

We get an element of the domain `ode` which we can now solve:

```
solve(eq)
```

$$\left\{\frac{C2}{x^3} - \frac{\cos(x)}{x^3}\right\}$$

```
delete eq:
```

Example 2

An initial value problem is defined as a set consisting of the ODE and the initial conditions:

```
ivp := ode({f''(t) + 4*f(t) = sin(2*t),
            f(0) = a, f'(0) = b}, f(t))
```

$$\text{ode}(\{f(0) = a,\ f'(0) = b,\ f''(t) + 4\,f(t) - \sin(2\,t)\},\ f(t))$$

```
solve(ivp)
```

$$\left\{\frac{3\sin(2\,t)}{32} - \frac{\sin(6\,t)}{32} + \frac{b\sin(2\,t)}{2} - \cos(2\,t)\left(\frac{t}{4} - \frac{\sin(4\,t)}{16}\right) + a\cos(2\,t)\right\}$$

```
delete ivp:
```

Example 3

With some restrictions, it is also possible to solve systems of ODEs. First, define a system:

```
sys := {x'(t) - x(t) + y(t) = 0, y'(t) - x(t) - y(t) = 0}
```

$$\{x'(t) - x(t) + y(t) = 0,\ y'(t) - y(t) - x(t) = 0\}$$

A call to `solve` yields the general solution with arbitrary parameters:

```
solution := solve(ode(sys, {x(t), y(t)}))
```

$$\{[y(t) = C8\,e^t\cos(t) + C7\,e^t\sin(t),\ x(t) = C7\,e^t\cos(t) - C8\,e^t\sin(t)]\}$$

To verify the result, substitute it back into the system `sys`. However, for the substitution, you must rewrite the system into a notation using the `diff` function:

```
eval(subs(rewrite(sys, diff), op(solution)))
```

$$\{0 = 0\}$$

```
delete sys, solution:
```

If you have a system of differential equations in a matrix form, extract the components of the matrix to a set of differential equations:

```
Y:= matrix([x(t), y(t)]):
A:= matrix([[1, 2], [-1, 6]]):
s := ode({op(diff(Y, t) - A*Y)}, {x(t), y(t)})
```

$$\text{ode}(\{x'(t) - x(t) - 2\,y(t),\ y'(t) - 6\,y(t) + x(t)\},\ \{x(t),\ y(t)\})$$

Now, use the `solve` function to solve the system:

```
solve(s)
```

$$\left\{\left[y(t) = \text{C9}\,\sigma_2\left(\frac{\sqrt{17}}{4} + \frac{5}{4}\right) - \text{C10}\,\sigma_1\left(\frac{\sqrt{17}}{4} - \frac{5}{4}\right),\ x(t) = \text{C9}\,\sigma_2 + \text{C10}\,\sigma_1\right]\right\}$$

where

$$\sigma_1 = e^{-\frac{t\,(\sqrt{17}-7)}{2}}$$

$$\sigma_2 = e^{\frac{t\,(\sqrt{17}+7)}{2}}$$

Example 4

In this example, we point out the various return formats of `ode`'s solve facility. First, we solve an ODE with an initial condition. The solution involves a symbolic integral:

```
solve(ode({y'(x) + x*y(x) = cos(x), y(0) = 3}, y(x)))
```

$$\left\{3\,e^{-\frac{x^2}{2}} + e^{-\frac{x^2}{2}} \int_0^x e^{\frac{y^2}{2}} \cos(y)\,dy\right\}$$

For the next equation, we get an implicit solution:

```
solve(ode((x*y'(x) - y(x))^4*exp(x*y'(x) - y(x))
          - ln(x*y'(x) - y(x)), y(x)))
```

$$\text{solve}\left(e^{C13\,x - X1236}\,(X1236 - C13\,x)^4 - \ln(C13\,x - X1236) = 0,\ X1236\right)$$

This is an *algebraic* equation for `y`. Its solution defines `y` as a function of `x` and an arbitrary parameter `C` followed by a number automatically generated by MuPAD (constant of integration). However, the algebraic equation does not have a solution in closed form.

```
delete sys, solution:
```

Example 5

It may happen that MuPAD cannot solve a given equation. In such a case, a symbolic `solve` command is returned:

```
solve(ode(x*diff(y(x),x)-y(x)*(x*ln(x^3/y(x))+2), y(x)))
```

$$\text{solve}\left(\text{ode}\left(x\,y'(x) - y(x)\left(x\ln\left(\frac{x^3}{y(x)}\right) + 2\right),\ y(x)\right)\right)$$

Example 6

The MuPAD ODE solver contains algebraic algorithms for computing Liouvillian and non-Liouvillian solutions of linear ordinary differential equations. These algorithms are based on differential Galois theory and on additional methods for finding solutions of linear ordinary differential equations given in terms of special functions. For the famous Kovacic's example

$$y'' + \left(\frac{\frac{3}{16}}{x^2} + \frac{\frac{2}{9}}{(x-1)^2} - \frac{\frac{\frac{3}{16}}{x}}{x-1} \right) y = 0$$

,

the solution can be found as:

```
solve(ode(y''(x) + (3/(16*x^2) + 2/(9*(x - 1)^2)
                    - 3/(16*x*(x - 1)))*y(x), y(x)))
```

$$\left\{ C21\, x^{1/4} (x-1)^{2/3}\, {}_2F_1\left(\frac{1}{4}, \frac{7}{12}; \frac{1}{2}; x\right) + C22\, x^{3/4} (x-1)^{2/3}\, {}_2F_1\left(\frac{3}{4}, \frac{13}{12}; \frac{3}{2}; x\right) \right\}$$

MuPAD may find Liouvillian and non-Liouvillian solutions for higher order equations as well. However, in case of Liouvillian solutions, there is no guarantee that all of them are found.

MuPAD also finds non-Liouvillian solutions in terms of the Bessel, Airy, and Whittaker functions:

```
eq := y'(x) + y(x)^2 + b + a*x
```

$$y'(x) + y(x)^2 + b + a\,x$$

```
solve(ode(eq, y(x)))
```

$$\left\{ \frac{\sqrt{3}\ \text{airyBi}\left(\sigma_1, 1\right) i + \text{airyBi}\left(\sigma_1, 1\right) + C24\ \text{airyAi}\left(\sigma_1, 1\right) + \sqrt{3}\ C24\ \text{airyAi}\left(\sigma_1, 1\right) i}{2\ \text{airyBi}\left(\sigma_1, 0\right) \left(\frac{1}{a}\right)^{1/3} + 2\ C24\ \text{airyAi}\left(\sigma_1, 0\right) \left(\frac{1}{a}\right)^{1/3}} \right\}$$

where

$$\sigma_1 = \frac{b + \sqrt{3}\, b\, i + a\, x + \sqrt{3}\, a\, x\, i}{2\, a \left(\frac{1}{a}\right)^{1/3}}$$

We check this solution:

```
simplify(eval(subs(rewrite(eq, diff), y(x) = op(%))))
```

0

Example 7

It is also possible to compute the series solutions of an ordinary differential equation (cf. `ode::series` for further details):

```
series(ode(y''(x) + 4*y(x) = sin(w*x), y(x)), x = 0, 8)
```

$$\left\{y(0)+x\,y'(0)-2\,x^2\,y(0)+x^3\left(\frac{w}{6}-\frac{\sigma_1}{3}\right)+\frac{2\,x^4\,y(0)}{3}-x^5\left(-\frac{\sigma_1}{15}+\frac{w^3}{120}+\frac{w}{30}\right)-\frac{4\,x^6\,y(0)}{45}\right.$$
$$\left.+x^7\left(-\frac{4\,y'(0)}{315}+\frac{w^5}{5040}+\frac{w^3}{1260}+\frac{w}{315}\right)+O(x^8)\right\}$$

where

$$\sigma_1 = 2\,y'(0)$$

Parameters

`eq, eq1, eq2, …`

Equations or arithmetical expressions in the unknown functions and their derivatives with respect to `x`. An arithmetical expression is regarded as an equation with vanishing right hand side.

`y, y1, y2, …`

The unknown functions: identifiers

`x`

The independent variable: an identifier

`inits`

The initial or boundary conditions: a sequence of equations

Return Values

Object of type `ode`.

References

[1] E. Kamke. “Differentialgleichungen: Lösungmethoden und Lösungen”. B.G. Teubner, Stuttgart, 1997.

[2] G.M. Murphy. “Ordinary differential equations and their solutions”. Van Nostrand, Princeton, 1960.

[3] D. Zwillinger. “Handbook of differential equations”. San Diego: Academic Press, 1992.

[4] W. Fakler. Algebraische Algorithmen zur Lösung von linearen Differentialgleichungen. Stuttgart, Leipzig: Teubner, Reihe MuPAD Reports, 1999.

[5] M. van der Put and M.F. Singer. “Galois theory of linear differential equations”. Grundlehren der Mathematischen Wissenschaften, 328, Springer-Verlag, Berlin, 2003.

[6] F. Ulmer and M.F. Singer. Liouvillian and algebraic solutions of second and third order linear differential equations. “J. Symb. Comp.”, 16:37-74, 1993.

See Also

MuPAD Domains
`Dom::LinearOrdinaryDifferentialOperator`

MuPAD Functions
`numeric::odesolve` | `numeric::odesolve2` | `ode::series` | `ode::solve`

MuPAD Graphical Primitives
`plot::Ode2d` | `plot::Ode3d`

op

Operands of an object

Syntax

```
op(object)

op(object, i)

op(object, i .. j)

op(object, [i1, i2, …])
```

Description

`op(object)` returns all operands of the object.

`op(object, i)` returns the `i`-th operand.

`op(object, i..j)` returns the `i`-th to `j`-th operands.

MuPAD objects are composed of simpler parts: the “operands”. The function `op` is the tool to decompose objects and to extract individual parts. The actual definition of an operand depends on the type of the object. The 'Background' section below explains the meaning for some of the basic data types.

`op(object)` returns a sequence of all operands except the 0-th one. This call is equivalent to `op(object, 1..nops(object))`. Cf. “Example 1” on page 1-1466.

`op(object, i)` returns the `i`-th operand. Cf. “Example 2” on page 1-1467.

`op(object, i..j)` returns the `i`-th to `j`-th operands as an expression sequence; `i` and `j` must be nonnegative integers with `i` smaller or equal to `j`. This sequence is equivalent to `op(object, k) $k = i..j`. Cf. “Example 3” on page 1-1467.

`op(object, [i1, i2, ...])` is an abbreviation for the recursive call `op (... op ( op(object, i1) , i2) , ...)` if `i1, i2, ...` are integers.

A call such as `op(object, [i..j, i2])` with integers `i < j` corresponds to `map(op(object, i..j), op, i2)`. Cf. “Example 4” on page 1-1468.

`op` returns `FAIL` if the specified operand does not exist. Cf. “Example 5” on page 1-1469.

Expressions of domain type `DOM_EXPR`, arrays, hfarrays, and floating point intervals have a 0-th operand.

- For expressions, this is “the operator” connecting the other operands. In particular, for symbolic function calls, it is the name of the function.
- For array and hfarrays, the 0-th operand is a sequence consisting of an integer (the dimension of the array) and a range for each array index.
- For a floating-point interval, the value of the 0-th operand depends on the precise type of the interval: If the interval is a union of rectangles, the 0-th operand is `hold(_union)`. If the interval is not a union and consists only of real numbers, the 0-th operand is hold(hull). In the remaining case of a rectangle with non-vanishing imaginary part, the 0-th operand is `FAIL`.

Other basic data types such as lists or sets do not have a 0-th operand. Cf. “Example 6” on page 1-1469.

For library domains, `op` is overloadable. In the `"op"` method, the internal representation can be accessed with `extop`. It is sufficient to handle the cases `op(x)`, `op(x, i)`, and `op(x, i..j)` in the overloading method, the call `op(x, [i1, i2, ...])` needs not be considered. Cf. “Example 7” on page 1-1470.

`op` is not overloadable for kernel domains.

Examples

Example 1

The call `op(object)` returns all operands:

```
op([a, b, c, [d, e], x + y])
```

$$a,\ b,\ c,\ [d,\ e],\ x+y$$

```
op(a + b + c^d)
```

$$a,\ b,\ c^d$$

```
op(f(x1, x2, x3))
```

$$x1,\ x2,\ x3$$

Example 2

The call `op(object, i)` extracts a single operand:

```
op([a, b, c, [d, e], x + y], 4)
```

$$[d,\ e]$$

```
op(a + b + c^d, 3)
```

$$c^d$$

```
op(f(x1, x2, x3), 2)
```

$$x2$$

Example 3

The call `op(object, i..j)` extracts a range of operands:

```
op([a, b, c, [d, e], x + y], 3..5)
```

$$c,\ [d,\ e],\ x+y$$

```
op(a + b + c^d, 2..3)
```

$$b,\ c^d$$

```
op(f(x1, x2, x3), 2..3)
```

$$x2,\ x3$$

A range may include the 0-th operand if it exists:

```
op(a + b + c^d, 0..2)
```

$$\_plus, a, b$$

```
op(f(x1, x2, x3), 0..2)
```

$$f, x1, x2$$

Example 4

The call `op(object, [i1, i2, ...])` specifies suboperands:

```
op([a, b, c, [d, e], x + y], [4, 1])
```

$$d$$

```
op(a + b + c^d, [3, 2])
```

$$d$$

```
op(f(x1, x2, x3 + 17), [3, 2])
```

$$17$$

Also ranges of suboperands can be specified:

```
op([a, b, c, [d, e], x + y], [4..5, 2])
```

$$e, y$$

```
op(a + b + c^d, [2..3, 1])
```

$$b, c$$

```
op(f(x1, x2, x3 + 17), [2..3, 1])
```

```
x2, x3
```

Example 5

Nonexisting operands are returned as `FAIL`:

```
op([a, b, c, [d, e], x + y], 8), op(a + b + c^d, 4),
op(f(x1, x2, x3), 4)
```

```
FAIL, FAIL, FAIL
```

Example 6

For expressions of type `DOM_EXPR`, the 0-th operand is "the operator" connecting the other operands:

```
op(a + b + c, 0), op(a*b*c, 0), op(a^b, 0), op(a[1, 2], 0)
```

```
_plus, _mult, _power, _index
```

For symbolic function calls, it is the name of the function:

```
op(f(x1, x2, x3), 0), op(sin(x + y), 0), op(besselJ(0, x), 0)
```

```
f, sin, besselJ
```

The 0-th operand of an array is a sequence consisting of the dimension of the array and a range for each array index:

```
op(array(3..100), 0)
```

```
1, 3..100
```

```
op(array(1..2, 1..3, 2..4), 0)
```

```
3, 1..2, 1..3, 2..4
```

```
op(hfarray(3..100), 0)
```

```
1, 3..100
```

```
op(hfarray(1..2, 1..3, 2..4), 0)
```

```
3, 1..2, 1..3, 2..4
```

No 0-th operand exists for other kernel domains:

```
op([1, 2, 3], 0), op({1, 2, 3}, 0), op(table(1 = y), 0)
```

```
FAIL, FAIL, FAIL
```

Example 7

For library domains, `op` is overloadable. First, a new domain `d` is defined via `newDomain`. The `"new"` method serves for creating elements of this type. The internal representation of the domain is a list of all arguments of this `"new"` method:

```
d := newDomain("d"):  d::new := () -> new(dom, [args()]):
```

The `"op"` method of this domain is defined. It is to return the elements of a sorted copy of the internal list which is accessed via `extop`:

```
d::op := proc(x, i = null())
           local internalList;
         begin
           internalList := extop(x, 1);
           op(sort(internalList), i)
         end_proc:
```

By overloading, this method is called when the operands of an object of type `d` are requested via `op`:

```
e := d(3, 7, 1):
op(e);
op(e, 2);
op(e, 1..2)
```

```
1, 3, 7
```

```
3
```

```
1, 3
```

```
delete d, e:
```

Example 8

Identifiers, integers, real floating-point numbers, character strings, and the Boolean constants are “atomic” objects. The only operand is the object itself:

```
op(x), op(17), op(0.1234), op("Hello World!")
```

```
x, 17, 0.1234, "Hello World!"
```

For rational numbers, the operands are the numerator and the denominator:

```
op(17/3)
```

```
17, 3
```

For complex numbers, the operands are the real part and the imaginary part:

```
op(17 - 7/3*I)
```

$$17, -\frac{7}{3}$$

Example 9

For sets, op returns the elements according to the *internal* order. Note that this order may differ from the ordering with which sets are printed on the screen:

```
s := {i^2 $ i = 1..19}
```

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361}
```

```
op(s)
```

```
1, 4, 361, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324
```

Indexed access to set elements uses the ordering visible on the screen:

```
s[1], s[2], s[3]
```

```
1, 4, 9
```

Note that access to set elements via `op` is *much faster* than indexed calls:

```
s := {sqrt(i) $ i = 1..500}:
time([op(s)])/time([s[i] $ i = 1..nops(s)]);
```

$$\frac{1}{364}$$

```
delete s:
```

Example 10

The operands of a list are its entries:

```
op([a, b, c, [d, e]])
```

$$a, b, c, [d, e]$$

```
op([[a11, a12], [a21, a22]], [2, 1])
```

```
a21
```

Example 11

Internally, the operands of arrays and hfarrays form a “linear” sequence containing all entries:

```
op(array(1..2, 1..2, [[11, 12], [21, 22]]))
```

```
11, 12, 21, 22
```

```
op(hfarray(1..2, 1..2, [[11, 12], [21, 22]]))
```

$$11.0,\ 12.0,\ 21.0,\ 22.0$$

Undefined entries are returned as `NIL`:

```
op(array(1..2, 1..2))
```

$$\mathrm{NIL},\ \mathrm{NIL},\ \mathrm{NIL},\ \mathrm{NIL}$$

Example 12

The operands of a table consist of equations relating the indices and the corresponding entries:

```
T := table((1, 2) = x + y, "diff(sin)" = cos, a = b)
```

	a	b
	$1, 2$	$x + y$
	"diff(sin)"	cos

```
op(T)
```

$$(1, 2) = x + y,\ \text{"diff(sin)"} = \cos,\ a = b$$

```
delete T:
```

Example 13

Expression sequences are not flattened:

```
op((a, b, c), 2)
```

$$b$$

Note, however, that the arguments passed to `op` are evaluated. In the following call, evaluation of `x` flattens this object:

```
x := hold((1, 2), (3, 4)):  op(x, 1)
```

```
    1
```

Use `val` to prevent simplification of `x`:

```
op(val(x), 1)
```

```
    1, 2
```

```
delete x:
```

Parameters

`object`

An arbitrary MuPAD object

`i, j`

Nonnegative integers

`i1, i2, …`

Nonnegative integers or ranges of such integers

Return Values

sequence of operands or the requested operand. `FAIL` is returned if no corresponding operand exists.

Overloaded By

object

Algorithms

We explain the meaning of “operands” for some basic data types:

- Identifiers, integers, real floating-point numbers, character strings, as well as the Boolean constants are “atomic” objects. They have only one operand: the object itself. Cf. “Example 8” on page 1-1471.
- A rational number of type `DOM_RAT` has two operands: the numerator and the denominator. Cf. “Example 8” on page 1-1471.
- A complex number of type `DOM_COMPLEX` has two operands: the real part and the imaginary part. Cf. “Example 8” on page 1-1471.
- The operands of a set are its elements.

Note: Note that the ordering of the elements as printed on the screen does not necessarily coincide with the internal ordering referred to by `op`. Cf. “Example 9” on page 1-1471.

- The operands of a list are its elements. Cf. “Example 10” on page 1-1472.
- The operands of arrays and hfarrays are its entries. Undefined entries are returned as `NIL`. Cf. “Example 11” on page 1-1472 and “Example 6” on page 1-1469.
- The operands of tables are the equations associating an index with the corresponding entry. Cf. “Example 12” on page 1-1473.
- The operands of an expression sequence are its elements. Note that such sequences are not flattened by `op`. Cf. “Example 13” on page 1-1473.
- The operands of a symbolic function call such as `f(x, y, ...)` are the arguments `x`, `y` etc. The function name `f` is the 0-the operand.
- In general, the operands of expressions of type `DOM_EXPR` are given by their internal representation. There is a 0-th operand (“the operator”) corresponding to the type of the expression. Internally, the operator is a system function, the expression corresponds to a function call. E.g., `a + b + c` has to be interpreted as `_plus(a, b, c)`, a symbolic indexed call such as `A[i, j]` corresponds to `_index(A, i, j)`. The name of the system function is the 0-th operand (i.e., `_plus` and `_index` in the previous examples), the arguments of the function call are the further operands.

See Also

MuPAD Functions

`_index` | `contains` | `extnops` | `extop` | `extsubsop` | `map` | `new` | `nops` | `select` | `split` | `subs` | `subsex` | `subsop` | `zip`

operator

Define a new operator symbol

Syntax

```
operator(symb, f, <Prefix | Postfix | Binary | Nary, prio>, <Global>)
operator(symb, Delete, <Global>)
```

Description

`operator(symb, f, T, prio)` defines a new operator symbol `symb` of type `T` with priority `prio`. The function `f` evaluates expressions using the new operator.

`operator(symb, Delete)` removes the definition of the operator symbol `symb`.

`operator` is used to define new user-defined operator symbols or to delete them.

Given the operator symbol `"++"`, say, with evaluating function `f`, the following expressions are built by the parser, depending on the type of the operator:

- `Prefix`:

  The input `++x` results in `f(x)`.
- `Postfix`:

  The input `x++` results in `f(x)`.
- `Binary`:

  The input `x ++ y ++ z` results in `f(f(x, y), z)`.
- `Nary`:

  The input `x ++ y ++ z` results in `f(x, y, z)`.

There may exist operator symbols which are prefixes of other operator symbols. The scanner reads as many characters as possible and chooses the longest matching operator symbol. Cf. “Example 3” on page 1-1478.

It is not possible to define two operators with the same symbol. So one may not define a unary ++ and a binary ++ at the same time.

The following restrictions exist for the operator symbol string `symb`:

- It may not be longer than 32 characters.
- It may not start with a white-space.
- It may not start with a \ (backslash) character.

Thus, the strings `" @"` and `"/"` are not allowed. Please note that currently `operator` does not check these restrictions.

Builtin operators may be redefined.

It is not possible to define out-fix operators like $|x|$ or 3-nary or other types of operators.

The new operator symbol is also used if files are read, with one exception: if a file is read with the function `read` using the option `Plain`, the new operator is not taken into account. (This option is used if MuPAD library files are read, because otherwise user-defined operators could change the meaning of the source code in an uncontrolled way.)

If the operator is defined while reading a file with option `Plain`, the definition will be used for the remainder of the file and then be deleted automatically. If the operator is defined with the option `Global`, this behavior is changed and the operator will not be active while reading the file, but will exist at the interactive level instead.

Environment Interactions

The new operator symbol `symb` is known by the parser and may be used to enter expressions. The new operator symbol will *not* be used when reading files using the function `read` with the option `Plain`.

The function `f` corresponding to the new operator will always be converted into a `function environment` containing an additional output routine for the operator output, unless it contained an output routine already.

Examples

Example 1

This example shows how to define an operator symbol for the bit-shift operation (as in the language C):

```
bitshiftleft := (a, b) -> a * 2^b:
operator("<<", bitshiftleft, Binary, 950):
```

After this call, the symbol << can be used to enter expressions:

```
2 << 1, x << y
```

$$4, 2^{y} x$$

```
operator("<<", Delete):
```

Example 2

Identifiers can be used as operator symbols:

```
operator("x", _vector_product, Binary, 1000):

PRETTYPRINT := FALSE:
print(Plain, a x b x c)
```

```
(a x b) x c
```

```
PRETTYPRINT := TRUE:
operator("x", Delete):
```

Example 3

This example shows that the scanner tries to match the longest operator symbol:

```
operator("~", F, Prefix, 1000):
operator("~>", F1, Prefix, 1000):
operator("~~>", F2, Prefix, 1000):

print(Plain, ~~ x, ~~> x, ~ ~> x, ~~~> x)
```

```
~ ~ x, ~~> x, ~ ~> x, ~ ~~> x

operator("~", Delete):
operator("~>", Delete):
operator("~~>", Delete):
```

Parameters

`symb`

The operator symbol: a character string.

`f`

The function evaluating expressions using the operator.

`prio`

The priority of the operator: an integer between 1 and 1999. The default is 1300.

Options

`Prefix`

The operator is regarded as a unary operator with prefix notation. Given the operator symbol `"++"` and the evaluation function `f`, the input `++x` is parsed as the expression `f(x)`.

`Postfix`

The operator is regarded as a unary operator with postfix notation. Given the operator symbol `"++"` and the evaluation function `f`, the input `x++` is parsed as the expression `f(x)`.

`Binary`

The operator is regarded as a non-associative binary operator with infix notation. Given the operator symbol `"++"` and the evaluation function `f`, the input `x ++ y ++ z` is parsed as the expression `f(f(x, y), z)`, i.e. the operator binds left-to-right.

`Nary`

The operator is regarded as an associative n-ary operator with infix notation. Given the operator symbol `"++"` and the evaluation function `f`, the input `x ++ y ++ z` is parsed as the expression `f(x, y, z)`.

`Delete`

The operator with symbol `symb` is deleted

`Global`

When defining an operator inside library or package code (technically: inside a file which is read with the option `Plain`), the option `Global` changes the meaning of the operator definition: Instead of defining an operator for the remainder of the file, it defines an operator for the interactive level.

Return Values

Void object of type `DOM_NULL`.

Algorithms

When the scanner reads a new token, it first discards any whitespace and backslash characters. Then it tries to match user-defined operator symbols. The longest user-defined operator symbol matching the scanned characters is made the next token. If no user-defined operator symbol matches, it scans for the built-in tokens.

The parser uses both recursive-descend and a operator precedence parsing. Built-in and user-defined operators are parsed using operator precedence.

ORDER

Default number of terms in series expansions

Description

The environment variable `ORDER` controls the default number of terms that the system returns when you compute a series expansion.

Possible values: Positive integer less than 2^{31}. The default value is 6.

The functions `taylor`, `series`, and `asympt` have an optional third argument specifying the desired number of terms of the requested series expansion, counting from the dominant term on (relative order). If this optional argument is missing, then the value of `ORDER` is used instead.

`ORDER` may also affect the results returned by the function `limit`.

Deletion via the statement "`delete ORDER`" resets `ORDER` to its default value 6. Executing the function `reset` also restores the default value.

In some cases, the number of terms returned by `taylor`, `series`, or `asympt` may not agree with the value of `ORDER`. Cf. "Example 2" on page 1-1482.

Examples

Example 1

In the following example, we compute the first 6 terms of the series expansion of the function `exp(x)/x^2` around the origin:

```
series(exp(x)/x^2, x = 0)
```

$$\frac{1}{x^2} + \frac{1}{x} + \frac{1}{2} + \frac{x}{6} + \frac{x^2}{24} + \frac{x^3}{120} + O(x^4)$$

To obtain the first 10 terms, we specify the third argument of `series`:

```
series(exp(x)/x^2, x = 0, 10)
```

$$\frac{1}{x^2}+\frac{1}{x}+\frac{1}{2}+\frac{x}{6}+\frac{x^2}{24}+\frac{x^3}{120}+\frac{x^4}{720}+\frac{x^5}{5040}+\frac{x^6}{40320}+\frac{x^7}{362880}+O(x^8)$$

Alternatively, we increase the value of `ORDER`. This affects all subsequent calls to `series` or any other function returning a series expansion:

```
ORDER := 10: series(exp(x)/x^2, x = 0)
```

$$\frac{1}{x^2}+\frac{1}{x}+\frac{1}{2}+\frac{x}{6}+\frac{x^2}{24}+\frac{x^3}{120}+\frac{x^4}{720}+\frac{x^5}{5040}+\frac{x^6}{40320}+\frac{x^7}{362880}+O(x^8)$$

```
taylor(x^2/(1 - x), x = 0)
```

$$x^2+x^3+x^4+x^5+x^6+x^7+x^8+x^9+x^{10}+x^{11}+O(x^{12})$$

Finally, we reset `ORDER` to its default value 6:

```
delete ORDER: taylor(x^2/(1 - x), x = 0)
```

$$x^2+x^3+x^4+x^5+x^6+x^7+O(x^8)$$

Example 2

The number of terms returned by `series` may differ from the value of `ORDER` when cancellation or rational exponents occur:

```
ORDER := 3:
```

```
series(exp(x) - 1 - x - x^2/2 - x^3/6, x = 0)
```

$$\frac{x^4}{24}+\frac{x^5}{120}+O(x^6)$$

```
series(1/(1 - sqrt(x)), x = 0)
```

$$1+\sqrt{x}+x+x^{3/2}+x^2+x^{5/2}+O(x^3)$$

```
delete ORDER:
```

See Also

MuPAD Functions

asympt | limit | O | series | taylor

pade

Pade approximation

Syntax

```
pade(f, x, <[m, n]>)
pade(f, x = x0, <[m, n]>)
```

Description

`pade(f, ...)` computes a Pade approximant of the expression `f`.

The Pade approximant of order $[m, n]$ around $x = x_0$ is a rational expression

$$\frac{(x-x_0)^p\left(a_0+a_1\left(x-x_0\right)+\ldots+a_m\left(x-x_0\right)^m\right)}{1+b_1\left(x-x_0\right)+\ldots+b_n\left(x-x_0\right)^n}$$

approximating f. The parameters p and a_0 are given by the leading order term $f = a_0 (x - x_0)^p + O((x - x_0)^{p+1})$ of the series expansion of f around $x = x_0$. The parameters $a_1, \ldots, b_n$ are chosen such that the series expansion of the Pade approximant coincides with the series expansion of f to the maximal possible order.

The expansion points `infinity`, `-infinity`, and `complexInfinity` are not allowed.

If no series expansion of f can be computed, then `FAIL` is returned. Note that `series` must be able to produce a Taylor series or a Laurent series of f, i.e., an expansion in terms of integer powers of $x - x_0$ must exist.

Examples

Example 1

The Pade approximant is a rational approximation of a series expansion:

```
f := cos(x)/(1 + x): P := pade(f, x, [2, 2])
```

$$\frac{-7x^2+2x+12}{x^2+14x+12}$$

For most expressions of leading order 0, the series expansion of the Pade approximant coincides with the series expansion of the expression through order $m + n$:

```
S := series(f, x, 6)
```

$$1-x+\frac{x^2}{2}-\frac{x^3}{2}+\frac{13x^4}{24}-\frac{13x^5}{24}+O(x^6)$$

This differs from the expansion of the Pade approximant at order 5:

```
series(P, x, 6)
```

$$1-x+\frac{x^2}{2}-\frac{x^3}{2}+\frac{13x^4}{24}-\frac{85x^5}{144}+O(x^6)$$

The series expansion can be used directly as input to`pade`:

```
pade(S, x, [2, 3]), pade(S, x, [3, 2])
```

$$-\frac{5x^2-12}{x^3+x^2+12x+12}, \frac{-7x^3+7x^2+12x-12}{13x^2-12}$$

Both Pade approximants approximate `f` through order $m + n = 5$:

```
map([%], series, x)
```

$$\left[1-x+\frac{x^2}{2}-\frac{x^3}{2}+\frac{13x^4}{24}-\frac{13x^5}{24}+O(x^6), 1-x+\frac{x^2}{2}-\frac{x^3}{2}+\frac{13x^4}{24}-\frac{13x^5}{24}+O(x^6)\right]$$

```
delete f, P, S:
```

Example 2

The following expression does not have a Laurent expansion around $x = 0$:

```
series(x^(1/3)/(1 - x), x)
```

$$x^{1/3} + x^{4/3} + x^{7/3} + x^{10/3} + x^{13/3} + x^{16/3} + O\left(x^{19/3}\right)$$

Consequently, `pade` fails:

```
pade(x^(1/3)/(1 - x), x, [3, 2])
```

FAIL

Example 3

Note that the specified orders [m, n] do not necessarily coincide with the orders of the numerator and the denominator if the series expansion does not start with a constant term:

```
pade(x^10*exp(x), x, [2, 2]), pade(x^(-10)*exp(x), x, [2, 2])
```

$$\frac{x^{10}\,(x^2 + 6\,x + 12)}{x^2 - 6\,x + 12}, \frac{x^2 + 6\,x + 12}{x^{10}\,(x^2 - 6\,x + 12)}$$

Parameters

`f`

An arithmetical expression or a series of domain type `Series::Puiseux` generated by the function `series`

`x`

An identifier

`x0`

An arithmetical expression. If `x0` is not specified, then `x0` = 0 is assumed.

`[m, n]`

A list of nonnegative integers specifying the order of the approximation. The default values are `[3, 3]`.

Return Values

Arithmetical expression or `FAIL`.

See Also

MuPAD Functions

`series`

partfrac

Partial fraction decomposition

Syntax

```
partfrac(f, <x>)
```

```
partfrac(f, x, options)
```

Description

`partfrac(f, x)` returns the partial fraction decomposition of the rational expression `f` with respect to the variable `x`.

Consider the rational expression $f(x) = g(x) + \frac{p(x)}{q(x)}$ with polynomials g, p, q, such that `degree (p) < degree(q)`.

Factor of the denominator into non-constant and pair-wise coprime polynomials q_i with integer exponents e_i:

$$q(x) = q_1(x)^{e_1} q_2(x)^{e_2} \ldots$$

The partial fraction decomposition based on this factorization is a representation

$$f(x) = g(x) + \frac{p_{1,1}(x)}{q_1(x)} + \ldots + \frac{p_{1,e_1}(x)}{q_1(x)^{e_1}} + \frac{p_{2,1}(x)}{q_2(x)} + \ldots + \frac{p_{2,e_2}(x)}{q_2(x)^{e_2}} + \ldots$$

where $p_{i,j}$ are polynomials, such that `degree(p`$_{i,j}$`) < degree(q`$_i$`)`. In particular, $p_{i,j}$ are constants if q_i is a linear polynomial.

`partfrac` uses factors q_i found by the `factor` function. This function finds factorization over the field implied by the coefficients of the denominator. See “Example 2” on page 1-1490.

If `f` has only one indeterminate, and you do not use options, then you can omit the second argument `x` in a call to `partfrac`. Otherwise, specify the indeterminate as a second parameter.

`partfrac` can also find partial fraction decomposition with respect to expressions instead of variables. See "Example 3" on page 1-1490.

The option `Full` invokes a full factorization of the denominator into linear factors. The `MaxDegree` option determines whether partial fraction decomposition is a symbolic `sum` over `RootOf` or an expression in radicals. In general, roots belonging to an irreducible factor of the denominator of degree five or larger cannot be expressed in terms of radicals. See "Example 6" on page 1-1491.

Examples

Example 1

Find partial fraction decomposition of the following expressions. You can omit specifying a variable because these rational expressions are univariate.

```
partfrac(x^2/(x^3 - 3*x + 2))
```

$$\frac{5}{9\,(x-1)}+\frac{1}{3\,(x-1)^2}+\frac{4}{9\,(x+2)}$$

```
partfrac(23 + (x^4 + x^3)/(x^3 - 3*x + 2))
```

$$x+\frac{19}{9\,(x-1)}+\frac{2}{3\,(x-1)^2}+\frac{8}{9\,(x+2)}+24$$

```
partfrac(x^3/(x^2 + 3*I*x - 2))
```

$$x-\frac{7\,x+6\,\mathrm{i}}{x^2+3\,x\,\mathrm{i}-2}-3\,\mathrm{i}$$

Find partial fraction decomposition of the following expression containing two variables, `x` and `y`. For multivariate expressions, specify the variable with respect to which you compute the partial fraction decomposition.

```
f := x^2/(x^2 - y^2):
partfrac(f, x), partfrac(f, y)
```

$$\frac{y}{2\,(x-y)} - \frac{y}{2\,(x+y)} + 1,\ \frac{x}{2\,(x+y)} + \frac{x}{2\,(x-y)}$$

```
delete f:
```

Example 2

Find the partial fraction decomposition of this expression.

```
partfrac(1/(x^2 - 2), x)
```

$$\frac{1}{x^2-2}$$

The denominator x^2 - 2 does not factor over the rational numbers.

```
factor(x^2 - 2)
```

$$x^2-2$$

Extend the coefficient field used by `factor` and `partfrac` by using the `Adjoin` option.

```
partfrac(1/(x^2 - 2), x, Adjoin = [sqrt(2)])
```

$$\frac{\sqrt{2}}{4\,(x-\sqrt{2})} - \frac{\sqrt{2}}{4\,(x+\sqrt{2})}$$

Example 3

Find the partial fraction decomposition with respect to an expression, such as `sin(x)`.

```
partfrac(1/(sin(x)^4 - sin(x)^2 + sin(x) - 1), sin(x))
```

$$\frac{1}{3\,(\sin(x)-1)}-\frac{\frac{\sin(x)^2}{3}+\frac{2\sin(x)}{3}+\frac{2}{3}}{\sin(x)^3+\sin(x)^2+1}$$

Example 4

Return a list consisting of the numerators and denominators of the partial fraction decomposition by using the `List` option.

```
partfrac(x^2/(x^3 - 3*x + 2), x, List)
```

$$\left[\frac{4}{9},\ \frac{5}{9},\ \frac{1}{3}\right],\ [x+2,\ x-1,\ (x-1)^2]$$

Example 5

Find the partial fraction decomposition using numeric factorization over the field real numbers, `R_`.

```
partfrac(1/(x^3 - 2), x, Domain = R_)
```

$$\frac{0.2099868416}{x-1.25992105}-\frac{0.2099868417\,x+0.529133684}{x^2+1.25992105\,x+1.587401052}$$

Find the partial fraction decomposition of the same expression using numeric factorization over the field complex numbers, `C_`.

```
partfrac(1/(x^3 - 2), x, Domain = C_)
```

$$\frac{0.2099868416}{x-1.25992105}+\frac{-0.1049934208-0.1818539393\,i}{x+0.6299605249+1.091123636\,i}+\frac{-0.1049934208+0.1818539393\,i}{x+0.6299605249-1.091123636\,i}$$

Example 6

Find the partial fraction decomposition factoring the denominator into linear factors symbolically. For this, use the `Full` option.

```
partfrac(1/(x^3 + x - 2), x, Full)
```

$$\frac{1}{4\,(x-1)}+\frac{-\frac{1}{8}+\frac{3\sqrt{7}\,i}{56}}{x+\frac{1}{2}-\frac{\sqrt{7}\,i}{2}}-\frac{\frac{1}{8}+\frac{3\sqrt{7}\,i}{56}}{x+\frac{1}{2}+\frac{\sqrt{7}\,i}{2}}$$

For irreducible denominators of the third and higher degrees, the partial fraction decomposition is a symbolic sum over the roots.

```
S:= partfrac(1/(x^3 + x - 3), x, Full)
```

$$\sum_{\text{alpha3}\,\in\,\text{RootOf}\left(z^3+z-3,\,z\right)}\left(-\frac{\frac{6\,\text{alpha3}^2}{247}+\frac{27\,\text{alpha3}}{247}+\frac{4}{247}}{\text{alpha3}-x}\right)$$

MuPAD uses the `freeze` function to keep the result in the form of an unevaluated symbolic sum. To evaluate this symbolic sum, use `unfreeze`. Evaluating this symbolic sum simplifies it back to the original input.

```
unfreeze(S); delete S:
```

$$\frac{1}{x^3+x-3}$$

Parameters

`f`

Rational expression in `x`

`x`

Indeterminate: typically, an identifier or an indexed identifier

Options

`Full`

Factor the denominator completely into linear factors, and find the partial fraction decomposition with respect to that factorization.

`List`

Return a list consisting of the numerators and denominators of the partial fraction decomposition.

`MaxDegree`

Option, specified as `MaxDegree = n`

Adjoin only the coefficients of the denominator with the algebraic degree not exceeding `n` to the field over which the denominator is factored. If you also use `Full`, then `partfrac` does not use explicit formulas involving radicals to solve polynomial equations of a degree higher than `n`.

`Adjoin`

Option, specified as `Adjoin = g`

Factor the denominator over the smallest field containing the rational numbers, all coefficients of the denominator, and the elements of `g`.

`Domain`

Option, specified as `Domain = d`

Factor the denominator over the domain `d`, where `d` is one of the following: `Expr`, `R_`, or `C_`. By default, `d = Expr`. For more details, see `factor`.

`Mapcoeffs`

Option, specified as `Mapcoeffs = mp`

When building the resulting expression, insert `mp(c)` instead of each coefficient `c`.

Return Values

arithmetical expression.

Overloaded By

f

See Also

MuPAD Functions

`collect` | `denom` | `divide` | `expand` | `factor` | `normal` | `numer` | `rectform` | `rewrite` | `simplify`

pathname

Create a platform dependent path name

Syntax

```
pathname(dir, subdir, …)
```

```
pathname(<Root>, dir, subdir, …)
```

Description

`pathname(dir, subdir, ...)` returns a relative path name valid on the used operating system.

`pathname` is used to specify pathnames via MuPAD strings. Directories and subdirectories are concatenated in a suitable way creating a valid pathname for the currently used operating system. For example, this mechanism may be used to specify the location of library files independent of the platform.

In order to create valid path names for the operating systems supported by MuPAD, the conventions holding for the corresponding operating system must be complied with. In particular, the names must not contain the characters "/" , "\" or ":". Compliance with these conventions is tested by `pathname`.

Under Microsoft® Windows, `pathname` does not allow to specify a volume to become part of the path name. Names are always relative to the current volume.

Examples:

Call	Platform	Result
`pathname("lib", "linalg")`	UNIX (Linux/ Mac OS X)	`"lib/linalg/"`
	Microsoft Windows	`"lib\\linalg\\"`
`pathname(Root, "lib", "linalg")`	UNIX (Linux/Mac OS X)	`"/lib/linalg/"`

Call	Platform	Result
	Microsoft Windows	`"\\lib\\linalg\\"`

Examples

Example 1

The following examples are created on a UNIX/Linux system:

```
pathname("lib", "linalg")
```

```
"lib/linalg/"
```

```
pathname(Root, "lib", "linalg") . "det.mu"
```

```
"/lib/linalg/det.mu"
```

Parameters

`dir, subdir, …`

Names of directories: character strings

Options

`Root`

Makes `pathname` generate an absolute path name

Return Values

String.

See Also

MuPAD Functions

`fclose` | `finput` | `fopen` | `fprint` | `fread` | `ftextinput` | `import::readbitmap` | `import::readdata` | `print` | `protocol` | `read` | `READPATH` | `write` | `WRITEPATH`

pdivide

Pseudo-division of polynomials

Syntax

```
pdivide(p, q, <[x]>, <order>, options)
```

```
pdivide(p, q, <[x1, x2, …]>, <order>, options)
```

```
pdivide(p, q1, q2, …, <[x1, x2, …]>, <order>, options)
```

Description

`pdivide(p, q)` performs pseudo-division of polynomials or polynomial expressions `p` and `q`. The function returns the factor `b`, the pseudo-quotient `s`, and the pseudo-remainder `r`, such that `b*p = s*q + r`.

`pdivide(p, q1, q2, q3, ..., qN)` performs pseudo-division of a polynomial or a polynomial expression `p` by polynomials or polynomial expressions `q1, q2, q3, ..., qN`.

`pdivide(p, q)` returns the sequence `b, s, r`, where `b` is an element of the coefficient ring of the polynomials. The pseudo-quotient `s` and pseudo-remainder `r` satisfy these conditions: `b*p = s*q + r`, `degree(p) = degree(s) + degree(q)`, and `degree(r) < degree(q)`.

By default, `pdivide` determines the factor `b` as `b = lcoeff (q)^( degree (p) - degree(q) + 1)`. `AnyFactor` enables `pdivide` to use other values of `b`. See “Example 2” on page 1-1500.

`pdivide` operates on polynomials or polynomial expressions.

Polynomials must be of the same type, meaning that their variables and coefficient rings must be identical.

When you call `pdivide` for polynomial expressions, MuPAD internally converts these expressions to polynomials. See the `poly` function. If the expressions cannot be converted to polynomials, `pdivide` returns `FAIL`. See “Example 3” on page 1-1501.

If you call `pdivide` for polynomials, it returns polynomials. If you call `pdivide` for polynomial expressions, it returns polynomial expressions.

If you perform pseudo-division of polynomial expressions that contain multiple variables, you can specify particular variables to be treated as variables. The `pdivide` function treats all other variables as symbolic parameters. By default, `pdivide` assumes that all variables in polynomial expressions are variables, and none of them is a symbolic parameter. See "Example 4" on page 1-1501.

`pdivide(p, q1, q2, q3, ..., qN)` returns the factor `b`, pseudo-quotients `s1, s2, ..., sN` and the pseudo-remainder `r`, such that `b*p = s1*q1 + s2*q2 + ... + sN*qN + r`.

When performing pseudo-division of a polynomial by one or more polynomials, you can select the term ordering. The ordering accepts these values:

- `LexOrder` sets the lexicographical ordering.
- `DegreeOrder` sets the total degree ordering. When using this ordering, MuPAD sorts the terms of a polynomial according to the total degree of each term (the sum of the exponents of the variables).
- `DegInvLexOrder` sets the total degree inverse lexicographic ordering. When using this ordering, MuPAD sorts the terms of a polynomial according to the total degree of each term (the sum of the exponents of the variables). If several terms have equal total degrees, MuPAD sorts them using the inverse lexicographic ordering.
- Your custom term ordering of type `Dom::MonomOrdering`.

See "Example 5" on page 1-1502.

In contrast to `divide`, `pdivide` does not require that the coefficient ring of the polynomials implements a `"_divide"` slot because coefficients are not divided in this algorithm. See "Example 6" on page 1-1503.

Examples

Example 1

Perform pseudo-division of these two polynomials:

```
p:= poly(x^3 + x + 1):  q:= poly(3*x^2 + x + 1):
```

```
[b, s, r] := [pdivide(p, q)]
```

$$[9, \text{poly}(3x - 1, [x]), \text{poly}(7x + 10, [x])]$$

The result satisfies this equation:

```
p*b = s*q + r
```

$$\text{poly}(9x^3 + 9x + 9, [x]) = \text{poly}(9x^3 + 9x + 9, [x])$$

Now compute the pseudo-quotient and pseudo-remainder separately:

```
pdivide(p, q, Quo), pdivide(p, q, Rem)
```

$$\text{poly}(3x - 1, [x]), \text{poly}(7x + 10, [x])$$

```
delete p, q, b, s, r:
```

Example 2

By default, `pdivide` performs pseudo-division of `p` by `q` with the factor `b` determined by the formula `b = lcoeff (q)^( degree (p) - degree(q) + 1)`:

```
p := 4*x^2 + 3: q := 2*x + 2:
b = lcoeff(q)^(degree(p) - degree(q) + 1);
pdivide(p, q)
```

$$b = 4$$

$$4, 8x - 8, 28$$

To enable `pdivide` to alter the value of `b`, use `AnyFactor`:

```
pdivide(4*x^2 + 3, 2*x + 2, AnyFactor)
```

$$1, 2x - 2, 7$$

Example 3

If an expression cannot be converted to a polynomial, `pdivide` returns `FAIL`:

```
pdivide(1/x, x)
```

FAIL

Example 4

When performing pseudo-division of multivariate polynomials, you can specify the list of variables. The `pdivide` function assumes all other variables are symbolic parameters. For example, divide the following two polynomial expressions specifying that `x`, `y`, and `a` are variables. The resulting pseudo-quotient is 0, and the pseudo-remainder equals the dividend `p`:

```
p := x^3 + x + y:
q := a*x^2 + x + 1:
pdivide(p, q, [x, y, a])
```

$$1,\ 0,\ x^3 + x + y$$

Divide these expressions specifying that `x` and `y` are variables. MuPAD assumes that `a` is a symbolic parameter. Here, both the pseudo-quotient and pseudo-remainder are not equal to 0:

```
pdivide(p, q, [x, y])
```

$$a^2,\ a\,x - 1,\ a^2\,y + x\,(a^2 - a + 1) + 1$$

Now divide the same polynomial expressions specifying that only `y` is a variable. MuPAD assumes that `x` and `a` are symbolic parameters. Here the pseudo-remainder is 0:

```
pdivide(p, q, [y])
```

$$(a\,x^2 + x + 1)^2,\ y\,(a\,x^2 + x + 1) + (x^3 + x)\,(a\,x^2 + x + 1),\ 0$$

By default, the `pdivide` function treats polynomial expressions with multiple variables as multivariate polynomial expressions. The function does not assume that any of the variables are symbolic parameters:

```
pdivide(x^3 + x + y, a*x^2 + x + 1)
```

$$1, 0, x^3 + x + y$$

Example 5

`pdivide` lets you perform pseudo-division of a polynomial (or polynomial expression) by multiple polynomials (or polynomial expressions):

```
p := 4*x^4 + a*x^2*y^4:
q1 := x^3 - a:
q2 := x + y:
[b, s1, s2, r] := [pdivide(p, q1, q2)]
```

$$[-1, x^2 y^4, -x^4 y^4 + x^3 y^5 - 4 x^3 - x^2 y^6 + 4 x^2 y + x y^7 - 4 x y^2 - y^8 + 4 y^3, y^9 - 4 y^4]$$

The result satisfies the condition `b*p = s1*q1 + s2*q2 + r`:

```
testeq(b*p, s1*q1 + s2*q2 + r)
```

$$\text{TRUE}$$

When dividing a polynomial by multiple polynomials, you can select the term ordering:

```
pdivide(p, q1, q2, LexOrder)
```

$$-1, x^2 y^4, -x^4 y^4 + x^3 y^5 - 4 x^3 - x^2 y^6 + 4 x^2 y + x y^7 - 4 x y^2 - y^8 + 4 y^3, y^9 - 4 y^4$$

```
pdivide(p, q1, q2, DegreeOrder)
```

$$1, 4 x, -a y^5 + a x y^4 + 4 a, a y^6 - 4 a y$$

Example 6

The coefficient ring can be an arbitrary ring. For example, here the residue class ring of integers modulo 8 represents the coefficient ring:

```
pdivide(poly(x^3 + x + 1, IntMod(8)),
        poly(4*x^3 + x + 1, IntMod(8)))
```

$$4,\ \mathrm{poly}(1,\ [x],\ \mathrm{IntMod}(8)),\ \mathrm{poly}(3\,x+3,\ [x],\ \mathrm{IntMod}(8))$$

Note that `pdivide` does not require divisibility of the coefficients.

Parameters

`p, q`

Univariate or multivariate polynomials or polynomial expressions.

`p, q1, q2, …`

Univariate or multivariate polynomials or polynomial expressions.

`x`

The indeterminate of the polynomial, which is typically an identifier or an indexed identifier. `pdivide` treats the expressions as univariate polynomials in the indeterminate `x`.

`x1, x2, …`

The indeterminates of the polynomial, which are typically identifiers or indexed identifiers. `pdivide` treats multivariate expressions as multivariate polynomials in these indeterminates.

`order`

The term ordering when performing pseudo-division of one multivariate polynomial by one or more multivariate polynomials: `LexOrder`, `DegreeOrder`, `DegInvLexOrder`, or a custom term ordering of type `Dom::MonomOrdering`. The default is the lexicographical ordering `LexOrder`.

Options

`Quo, Rem`

Return only the pseudo-quotient or pseudo-remainder. By default, `pdivide` returns the sequence containing the factor `b`, pseudo-quotient `s` (or pseudo-quotients `s1, s2, ...`), and pseudo-remainder `r`. See “Example 1” on page 1-1499.

`AnyFactor`

Allow flexibility for the factor `b`. Without this option, `b = lcoeff (q)^( degree (p) - degree(q) + 1)`.

Return Values

Polynomial, or polynomial expression, or a sequence containing an element of the coefficient ring of the input polynomials and polynomials (or polynomial expressions), or the value `FAIL`.

Overloaded By

f, g, p, q

See Also

MuPAD Functions

`content` | `degree` | `divide` | `factor` | `gcd` | `gcdex` | `ground` | `lcoeff` | `multcoeffs` | `poly`

piecewise

Domain of conditionally defined objects

Syntax

```
piecewise([condition1, object1], [condition2, object2], …, <[Otherwise, objectN]>, <Ex
```

Description

`piecewise([condition1, object1], [condition2, object2], ...)` defines a conditional object that equals `object1` if `condition1` is provably true, `object2` if `condition2` is provably true, and so on. Typically, such objects define piecewise functions or express solutions based on a case analysis of the free parameters of the mathematical problem. See “Example 1” on page 1-1507.

A pair `[condition, object]` is called a *branch*. If `condition` is provably false, then `piecewise` discards the entire branch. If `condition` is provably true, then `piecewise` returns the corresponding `object`. If neither condition in a piecewise object is provably true, `piecewise` returns an object of type `piecewise` that contains all branches, except for branches with provably false conditions.

If all conditions are provably false, or if you call `piecewise` without any branches, then `piecewise` returns `undefined`. See “Example 1” on page 1-1507.

Conditions do not need to be exhaustive or exclusive. If conditions contain parameters, and you substitute values for the parameters, all conditions can become false. Also, several conditions can become true.

If several conditions are simultaneously true, `piecewise` returns the object from the first branch that contains the condition recognized as true. Ensure that all objects corresponding to the true conditions have the same mathematical meaning. Do not rely on the system to recognize the first mathematically true condition as true. Alternatively, you can use the `ExclusiveConditions` option to fix the order of the branches.

`piecewise([condition1, object1], [condition2, object2], ..., [Otherwise, objectN])` checks the conditions, and, if they are not satisfied, discards

them and returns `objectN`. The `Otherwise` condition occurs in the last branch. It can occur only once. It remains unchanged as long as there are other branches, but it is treated as true when all other branches are discarded because their conditions are false. See "Example 2" on page 1-1507.

The system checks the truth of the conditions for current values and properties of all involved identifiers each time it evaluates an object of type `piecewise`. Thus, it simplifies piecewise expressions under various different assumptions.

`piecewise` objects can be nested: both conditions and objects can be `piecewise` objects themselves. `piecewise` automatically "flattens" such objects. For example, `piecewise([conditionA, piecewise([conditionB, objectC])])` becomes `piecewise([conditionA and conditionB, objectC])`. See "Example 3" on page 1-1508.

Arithmetical and set-theoretic operations work for `piecewise` objects, provided these operations are defined for all objects contained in the branches. If `f` is such an operation and `p1, p2, ...` are `piecewise` objects, then `f(p1, p2, ...)` is the `piecewise` object consisting of all branches of the form `[condition1 and condition2 and ..., f(object1, object2, ...)]`, where `[condition1, object1]` is a branch of `p1`, `[condition2, object2]` is a branch of `p2`, and so on. In other words, applying `f` commutes with any assignment to free parameters in the conditions. See "Example 4" on page 1-1508.

`piecewise` objects can also be mixed with other objects in such operations. In such cases, if `p1` is not a `piecewise` object, the system treats it as a `piecewise` object with the only branch `[TRUE, p1]`. See "Example 5" on page 1-1509.

`diff`, `float`, `limit`, `int` and similar functions handle expressions involving `piecewise`. When you use a `piecewise` argument in unary operators and functions with one argument, the system maps the operator or function to the objects in each branch. See "Example 6" on page 1-1509, "Example 7" on page 1-1509, and "Example 8" on page 1-1510.

`piecewise` differs from the `if` and `case` branching statements. First, `piecewise` uses the property mechanism when deciding the truth of the conditions. Therefore, the result depends on the properties of the identifiers that appear in the conditions. Second, `piecewise` treats conditions mathematically, while `if` and `case` evaluate them syntactically. Third, `piecewise` internally sorts the branches. If conditions in several branches are true, `piecewise` can return any of these branches. See "Example 9" on page 1-1510.

The `ExclusiveConditions` option fixes the order of branches in a piecewise expression. If the condition in the first branch returns `TRUE`, then `piecewise` returns the expression from the first branch. If a true condition appears in any further branch, then `piecewise` returns the expression from that branch and removes all subsequent branches. Thus, `piecewise` with `ExclusiveConditions` is very similar to an `if-elif-end_if` statement. Nevertheless, `piecewise` with `ExclusiveConditions` still takes into account assumptions on identifiers and treats conditions mathematically while `if-elif-end_if` treats them syntactically. See “Example 10” on page 1-1511.

Environment Interactions

`piecewise` takes into account properties of identifiers.

Examples

Example 1

Define this rectangular function `f`. Without additional information about the variable x, the system cannot evaluate the conditions to `TRUE` or `FALSE`. Therefore, it returns the `piecewise` object.

```
f := x -> piecewise([x < 0 or x > 1, 0], [x >= 0 and x <= 1, 1])
```

$$x \to \begin{cases} 0 & \text{if } x < 0 \vee 1 < x \\ 1 & \text{if } 0 \le x \wedge x \le 1 \end{cases}$$

Call the function `f` with the following arguments. Every time you call this piecewise function, the system checks the conditions in its branches and evaluates the function.

```
f(0), f(2), f(I)
```

$$1,\ 0,\ \text{undefined}$$

Example 2

Create this piecewise function using the syntax that includes `Otherwise`:

```
pw:= piecewise([x > 0 and x < 1, 1], [Otherwise, 0])
```

$$\begin{cases} 1 & \text{if } x \in (0, 1) \\ 0 & \text{otherwise} \end{cases}$$

Evaluate pw for these three values:

```
pw | x = 1/2;
pw | x = 2;
pw | x = I;
```

$$1$$

$$0$$

$$0$$

For further computations, delete the identifier pw:

```
delete pw:
```

Example 3

Create this nested piecewise expression. MuPAD flattens nested `piecewise` objects.

```
p1 := piecewise([a > 0, a^2], [a <= 0, -a^2]):
piecewise([b > 0, a + b], [b = 0, p1 + b], [b < 0, a + b])
```

$$\begin{cases} a^2 & \text{if } 0 < a \wedge b = 0 \\ -a^2 & \text{if } a \le 0 \wedge b = 0 \\ a + b & \text{if } b \ne 0 \wedge b \in \mathbb{R} \end{cases}$$

Example 4

Find the sum of these piecewise functions. You can perform most operations on piecewise functions the same way as you would on ordinary arithmetical expressions. The result of an arithmetical operation is only defined at the points where all of the arguments are defined:

```
piecewise([x > 0, 1], [x < -3, x^2]) + piecewise([x < 2, x])
```

$$\begin{cases} x^2 + x & \text{if } x < -3 \\ x + 1 & \text{if } x \in (0, 2) \end{cases}$$

Example 5

Solve this equation. The solver returns the result as a piecewise set:

```
S := solve(a*x = 0, x)
```

$$\begin{cases} \mathbb{C} & \text{if } a = 0 \\ \{0\} & \text{if } a \neq 0 \end{cases}$$

You can use set-theoretic operations work for such sets. For example, find the intersection of this set and the interval `(3, 5)`:

```
S intersect Dom::Interval(3, 5)
```

$$\begin{cases} (3, 5) & \text{if } a = 0 \\ \varnothing & \text{if } a \neq 0 \end{cases}$$

Example 6

Many unary functions are overloaded for `piecewise` by mapping them to the objects in all branches of the input:

```
f := piecewise([x >= 0, arcsin(x)], [x < 0, arccos(x)]):
sin(f)
```

$$\begin{cases} \sqrt{1 - x^2} & \text{if } x < 0 \\ x & \text{if } 0 \le x \end{cases}$$

Example 7

Find the limit of this piecewise function:

```
limit(piecewise([a > 0, x],[a < 0 and x > 1, 1/x],
                [a < 0 and x <= 1, -x]), x = infinity)
```

$$\begin{cases} \infty & \text{if } 0 < a \\ 0 & \text{if } a < 0 \end{cases}$$

Example 8

Find the integral of this piecewise function:

```
int(piecewise([x < 0, x^2], [x > 0, x^3]), x = -1..1)
```

$$\frac{7}{12}$$

Example 9

Create this piecewise function. Here, `piecewise` cannot determine if any branch is true or false. To do that, `piecewise` needs additional information about the identifier `a`.

```
p1 := piecewise([a = 0, 0], [a <> 0, 1/a])
```

$$\begin{cases} 0 & \text{if } a = 0 \\ \frac{1}{a} & \text{if } a \neq 0 \end{cases}$$

Create a similar structure by using `if-then-else`. The `if-then-else` structure evaluates the conditions syntactically. Here, `a = 0` is technically false because the identifier `a` and the integer `0` are different objects.

```
p2 := (if a = 0 then 0 else 1/a end)
```

$$\frac{1}{a}$$

`piecewise` takes properties of identifiers into account:

```
p1 := piecewise([a + b = 0, 0], [Otherwise, 1/a]) assuming a + b = 0
```

```
0
```

`if-then-else` does not:

```
p2 := (if a + b = 0 then 0 else 1/a end) assuming a + b = 0
```

$$\frac{1}{a}$$

For further computations, delete identifiers `a`, `b`, `p1`, and `p2`:

```
delete a, b, p1, p2:
```

Example 10

Create this piecewise expression:

```
p := piecewise([x > 0, 1], [y > 0, 2])
```

$$\begin{cases} 1 & \text{if } 0 < x \\ 2 & \text{if } 0 < y \end{cases}$$

Evaluate the expression at `y = 1`:

```
p | y = 1
```

```
2
```

Now, create the piecewise expression with the same branches, but this time use `ExclusiveConditions` to fix the order of the branches. When you use this option, any branch can be true only if the previous branches are false.

```
pE := piecewise([x > 0, x], [y > 0, y], ExclusiveConditions)
```

$$\begin{cases} x & \text{if } 0 < x \\ y & \text{if } 0 < y \\ \text{ExclusiveConditions}_2 & \text{if } \text{ExclusiveConditions}_1 \end{cases}$$

Evaluate the expression at `y = 1`:

```
pE | y = 1
```

$$\begin{cases} x & \text{if } 0 < x \\ 1 & \text{if TRUE} \\ \text{ExclusiveConditions}_2 & \text{if ExclusiveConditions}_1 \end{cases}$$

When you use `ExclusiveConditions`, `piecewise` acts the same way as an `if-then-else` statement, but does not ignore properties of identifiers. For example, set the assumption that `x = 0`:

```
assume(x = 0)
```

The `piecewise` function call returns 0 because it uses the assumption on identifier `x`:

```
p := piecewise([x = 0, x], [Otherwise, 1/x^2])
```

$$x$$

The corresponding `if-then-else` statement ignores the assumption, and, therefore, returns `1/x^2`:

```
pIf := (if x = 0 then x else 1/x^2 end)
```

$$\frac{1}{x^2}$$

For further computations, delete identifiers `p`, `pE`, `x`, and `pIf`:

```
delete p, pE, x, pIf:
```

Example 11

Find a set of accumulation points of this piecewise function by calling `limit` with the `Intervals` option:

```
limit(piecewise([a > 0, sin(x)], [a < 0 and x > 1, 1/x],
         [a < 0 and x <= 1, -x]), x = infinity, Intervals)
```

$$\begin{cases} [-1, 1] & \text{if } 0 < a \\ \{0\} & \text{if } a < 0 \end{cases}$$

Example 12

Rewrite the `sign` function in terms of a `piecewise` object:

```
f := rewrite(sign(x), piecewise)
```

$$\begin{cases} 0 & \text{if } x = 0 \\ 1 & \text{if } 0 < x \\ -1 & \text{if } x < 0 \\ \frac{x}{\sqrt{\Im(x)^2 + \Re(x)^2}} & \text{if } x \notin \mathbb{R} \end{cases}$$

Example 13

Create this `piecewise` object:

```
f := piecewise([x > 0, 1], [x < -3, x^2])
```

$$\begin{cases} 1 & \text{if } 0 < x \\ x^2 & \text{if } x < -3 \end{cases}$$

Extract a particular condition or object:

```
piecewise::condition(f, 1), piecewise::expression(f, 2)
```

$$0 < x,\ x^2$$

The index operator has the same meaning as `piecewise::expression` and can be typed faster:

```
f[2]
```

$$x^2$$

The `piecewise::branch` function extracts whole branches:

```
piecewise::branch(f, 1)
```

$$[0 < x,\ 1]$$

You can form another `piecewise` object from the branches for which the condition satisfies a given selection criterion, or split the input into two `piecewise` objects, as the system functions `select` and `split` do it for lists:

```
piecewise::selectConditions(f, has, 0)
```

$$\{1 \text{ if } 0 < x$$

```
piecewise::splitConditions(f, has, 0)
```

$$[\{1 \text{ if } 0 < x,\ \{x^2 \text{ if } x < -3,\ \text{undefined}]$$

You can also create a copy of `f` with some branches added or removed:

```
piecewise::remove(f, 1)
```

$$\{x^2 \text{ if } x < -3$$

```
piecewise::insert(f, [x > -3 and x < 0, sin(x)])
```

$$\begin{cases} 1 & \text{if } 0 < x \\ x^2 & \text{if } x < -3 \\ \sin(x) & \text{if } x \in (-3, 0) \end{cases}$$

Parameters

`condition1, condition2, …`

Boolean constants, or expressions representing logical formulas

`object1, object2, …`

Arbitrary objects

`Otherwise`

Identifier that specifies the last condition. This condition is always treated as a true condition.

Options

`ExclusiveConditions`

The `ExclusiveConditions` option fixes the order of branches in a piecewise expression. This option causes `piecewise` to automatically remove branches with false conditions. Thus, `piecewise` with `ExclusiveConditions` is almost equivalent to an `if-elif-end_if` statement, except that `piecewise` takes into account assumptions on identifiers. For example, if the condition in the first branch returns `TRUE`, then `piecewise` returns the expression from the first branch. If a true condition appears in any further branch, then `piecewise` returns the expression from that branch and removes all subsequent branches.

Methods

Mathematical Methods

`_in` — Membership with `piecewise` on the left side

```
_in(p, S)
```

`contains` — Apply the function `contains` to the objects in all branches

```
contains(p, a)
```

This method overloads the function `contains`. The objects in all branches must be valid first arguments for `contains`.

`diff` — (partial) differentiation

```
diff(p, <x, …>)
```

If no variables are given, `p` is returned.

`discont` — Determine the discontinuities of a piecewise function

```
discont(p, x, <F>)
```

```
discont(p, x = a .. b, <F>)
```

The objects in all branches of `p` must be arithmetical expressions.

The optional third parameter has the same meaning as for the function `discont`.

As for the function `discont`, only discontinuities in the given interval `[a,b]` are returned when calling `piecewise(p, x = a..b)`.

`disregardPoints` — Heuristic for simplifying conditions

```
disregardPoints(p)
```

`expand` — Apply the function `expand` to the objects in all branches

```
expand(p)
```

`factor` — Apply the function `factor` to the objects in all branches

```
factor(p)
```

`getElement` — Get any element of a piecewise set

```
getElement(p)
```

The result is `FAIL` if no such common element is found.

This method overloads the function `solvelib::getElement`.

`has` — Test for the existence of a subobject

```
has(p, a)
```

`int` — Definite and indefinite integration of a piecewise function

```
int(p, x, <r>)
```

If a range `a..b` is given, this method computes the definite integral of `p` when `x` runs through that range.

`ilaplace` — Apply the function `ilaplace` to the objects in all branches

```
ilaplace(p, x, t)
```

`isFinite` — Test whether a piecewise set is finite

```
isFinite(p)
```

This method overloads `solvelib::isFinite`.

`laplace` — Apply the function `laplace` to the objects in all branches

```
laplace(p, x, t)
```

`limit` — Compute the limit of a piecewise function

```
limit(p, x, <Left | Right | Real>, <Intervals>, <NoWarning>)
```

```
limit(p, x = x0, <Left | Right | Real>, <Intervals>, <NoWarning>)
```

When called with the `Intervals` option, the method returns the set of accumulation points of a function.

If the method cannot find the function limit and cannot prove the limit does not exist, the function call returns an unevaluated `limit` function.

If the limit of a function does not exist, the method returns the special value `undefined`.

This method overloads the function `limit`.

`normal` — Apply the function `normal` to the objects in all branches

```
normal(p)
```

`partfrac` — Apply the function `partfrac` to the objects in all branches

```
partfrac(p)
```

`restrict` — Impose an additional condition

```
restrict(p, C)
```

`set2expr` — Membership with `piecewise` on the right side

```
set2expr(p, x)
```

The objects in all branches of `p` must represent sets.

This method overloads the system function `_in`.

`simplify` — Simplify a conditionally defined object

```
simplify(p)
```

`solve` — Solve a conditionally defined equation or inequality

```
solve(p, x, <option1, option2, …>)
```

For each branch `[condition, object]` of `p`, with `object` being an equation or inequality, the method determines the set of all values `x` such that both `condition` and `object` become true mathematically, and returns the union of all obtained sets. The return value can be a conditionally defined set.

This method overloads the function `solve`. See the corresponding help page for a description of the available options and an overview of the types of sets that can be returned.

`solveConditions` — Isolate a given identifier in all conditions

```
solveConditions(p, x)
```

`Union` — Union of a system of sets

```
Union(p, x, indexset)
```

The objects in all branches of `p` must represent sets.

For each branch `[condition, object]` of `p`, this method does the following. It substitutes for `x` in `object` all values from `indexset` satisfying `condition` and takes the union over all obtained sets. Then it returns the union over the resulting sets for all branches.

This method overloads the function `solvelib::Union`.

Access Methods

`_concat` — Merge piecewise objects

```
_concat(p, …)
```

`branch` — Nth branch

```
branch(p, n)
```

`op` — Branches

```
op(p)
```

```
op(p, n)
```

`op(p, n)` returns the `n`th branch of `p` as a list. If $n = 0$, then `piecewise` is returned.

`setBranch` — Replace the `i`th branch

```
setBranch(p, i, b)
```

`numberOfBranches` — Number of branches

```
numberOfBranches(p)
```

`condition` — Condition in a specific branch

```
condition(p, i)
```

`setCondition` — Replace the condition in a specific branch by another

```
setCondition(p, i, cond)
```

`expression` — Object in a specific branch

```
expression(p, i)
```

Instead of `piecewise::expression(p, i)`, the index operator `p[i]` can be used synonymously.

`_index` — Object in a specific branch

```
_index(p, i)
```

`piecewise::expression` can be used synonymously.

`setExpression` — Replace the object in a specific branch by another

```
setExpression(p, i, a)
```

`insert` — Insert a branch

```
insert(p, b)
```

`b` can either be a branch extracted from another conditionally defined object using `extop`, or a list `[condition, object]`.

See "Example 13" on page 1-1513.

`extmap` — Apply a function to the objects in all branches

```
extmap(p, f, <a, …>)
```

`mapConditions` — Apply a function to the conditions in all branches

```
mapConditions(p, f, <a, …>)
```

`map` — Apply the function `map` to the objects in all branches

```
map(p, f, <a, …>)
```

`map(p, f)` is equivalent to `piecewise::extmap(p, map, f)`.

`remove` — Remove a branch

```
remove(p, i)
```

`splitBranch` — Split a branch into two branches

```
splitBranch(p, i, newcondition)
```

`selectConditions` — Select branches depending on their condition

```
selectConditions(p, f, <a, …>)
```

For every condition in `p`, `f(condition a, …)` must return a Boolean constant.

If none of the conditions satisfies the selection criterion, `undefined` is returned.

`selectExpressions` — Select branches depending on their object

```
selectExpressions(p, f, <a, …>)
```

For every object in `p`, `f(object a, …)` must return a Boolean constant.

If none of the objects satisfies the selection criterion, `undefined` is returned.

`splitConditions` — Split branches depending on conditions

```
splitConditions(p, f, <a, …>)
```

For every condition in `p`, `f(condition a, …)` must return a Boolean constant.

See "Example 13" on page 1-1513.

`subs` — Substitution

```
subs(p, s, …)
```

This method overloads the function `subs`. The calling syntax is identical to that function. See the corresponding help page for a description of the various types that are allowed for `s`.

`zip` — Apply a binary operation pointwise

```
zip(p1, p2, f)
```

If we regard conditionally defined objects as functions from the set A of parameter values to a set B of objects, this method implements the canonical extension of the binary operation f on B to the binary operation g on the set B^A of all functions from A to B via $g(p1, p2)(a) = f(p1(a), p2(a))$ for all a in A.

If only one of the first two arguments—`p1`, say—is of type `piecewise`, then each branch `[condition, object]` of `p1` is replaced by `[condition, f(object, p2)]`.

If neither `p1` nor `p2` are of type `piecewise`, then `piecewise::zip(p1, p2, f)` returns `f(p1, p2)`.

Algorithms

The operands of a `piecewise` object (the branches) are pairs consisting of a condition and the object valid under that condition.

Methods overloading system functions always assume that they have been called via overloading, and that there is some conditionally defined object among their arguments. All other methods do not assume that one of their arguments is of type `piecewise`. This simplifies the use of `piecewise`: it is always allowed to enter `p:=piecewise(...)` and to call some method of `piecewise` with `p` as an argument. You do not need to care about the special case where `p` is not of type `piecewise` because some condition in its definition is true or all conditions are false.

See Also

MuPAD Functions

`assume` | `bool` | `case` | `if` | `is`

plot

Display graphical objects on the screen

Syntax

`plot(object`$_1$`, <object`$_2$`, …>, <attribute`$_1$`, attribute`$_2$`, …>)`

Description

`plot( object`$_1$`, object`$_2$`, ...)` displays the graphical objects object$_1$, object$_2$ etc. on the screen.

This function calls `plot::easy` for preprocessing its input.

The parameters object$_1$, object$_2$ etc. must be accepted by `plot::easy` or directly be graphical objects generated by routines of the plot library. This library provides many such objects including:

- function graphs (`plot::Function2d`, `plot::Function3d`),
- curves (`plot::Curve2d`, `plot::Curve3d`),
- points (`plot::Point2d`, `plot::Point3d`),
- lines (`plot::Line2d`, `plot::Line3d`),
- polygons (`plot::Polygon2d`, `plot::Polygon3d`),
- surfaces (of domain type `plot::Surface`)

and many more. Cf. “Example 1” on page 1-1524.

There are also many high level objects such as `plot::VectorField2d`, `plot::Ode2d`, `plot::Ode3d`, `plot::Implicit2d`, `plot::Implicit3d` etc. that can also be rendered by `plot`, `display`. Cf. “Example 2” on page 1-1526.

Graphical attributes `attribute1`, `attribute2` etc. are specified by equations of the form `AttributeName = AttributeValue`. There are several hundred such attributes that allow to modify almost any aspect of the graphics.

Note: The graphical objects `object1`, `object2` etc. must have the same dimension. A mix of 2D and 3D objects in one plot is not supported!

The command `plot()` creates an empty graphical 2D scene.

Examples

Example 1

The following calls return objects representing the graphs of the sine and the cosine function on the interval [0, 2 π]:

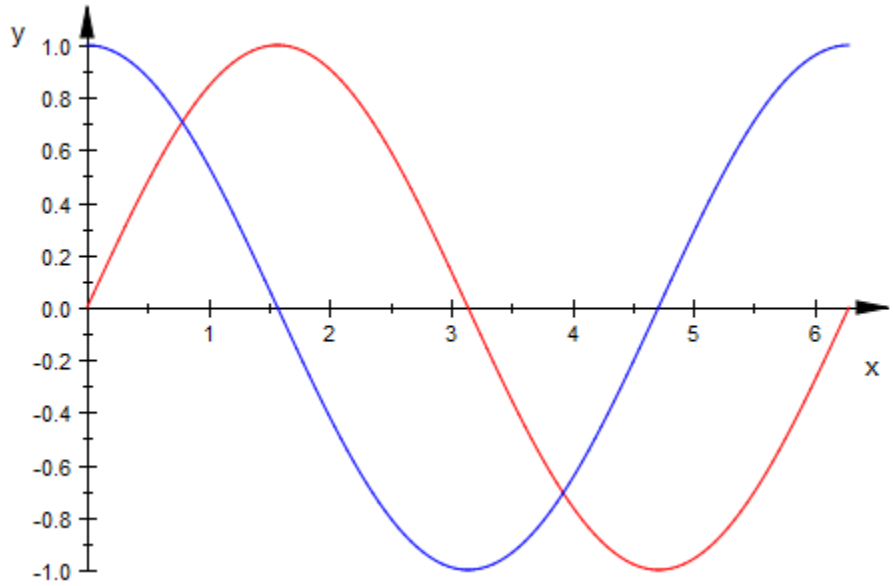
```
f1 := plot::Function2d(sin(x), x = 0..2*PI, Color = RGB::Red);
f2 := plot::Function2d(cos(x), x = 0..2*PI, Color = RGB::Blue)
```

$$\text{plot::Function2d}(\sin(x),\ x = 0..2\ \pi)$$

$$\text{plot::Function2d}(\cos(x),\ x = 0..2\ \pi)$$

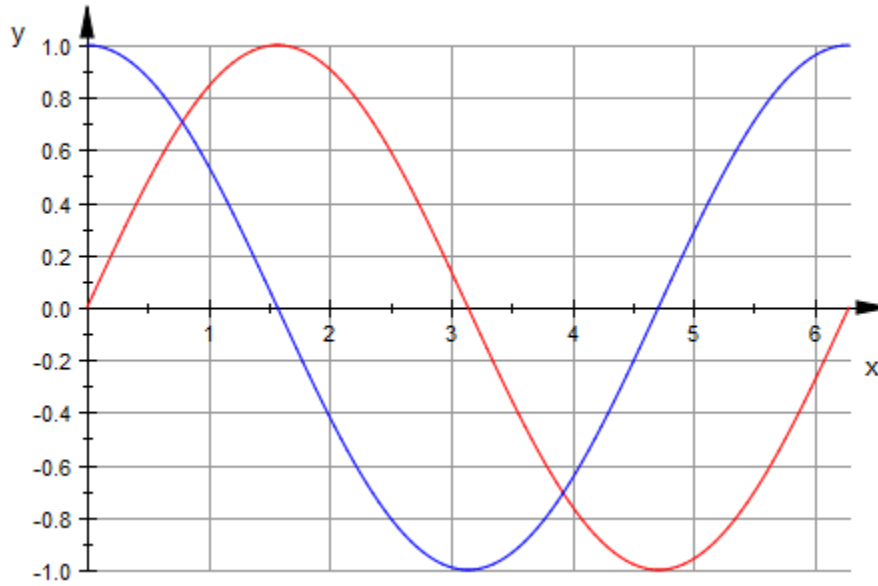
The following call renders these graphs:

```
plot(f1, f2)
```



Apart from the explicitly requested colors, this call uses the default values of all graphical attributes. If different values are desired, an arbitrary number of attributes may be passed as additional parameters to `plot`, `display`. For example, to draw grid lines in the background of the previous plot, we enter:

```
plot(f1, f2, GridVisible = TRUE)
```



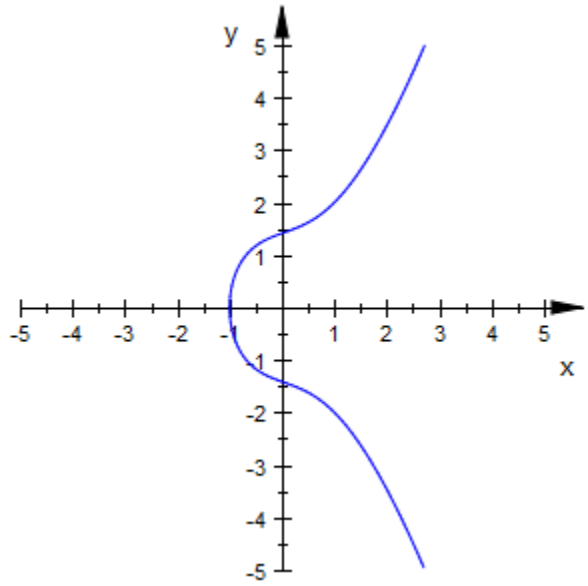

```
delete f1, f2:
```

Example 2

The plot library contains various routines for creating more complex graphical objects such as vectorfields, solution curves of ordinary differential equations, and implicitly defined curves.

For example, to plot the implicitly defined curve $x^2 + x + 2 = y^2$ with x, y from the interval [- 5, 5], we use the function `plot::Implicit2d`:

```
plot(plot::Implicit2d(x^3 + x + 2 = y^2,
                      x = -5..5, y = -5..5),
     Scaling = Constrained)
```



Here we used the `Scaling` attribute to guarantee an aspect ratio 1:1 between the x and y coordinates independent of the window size.

Parameters

`object`$_1$`, object`$_2$`, …`

2D or 3D graphical objects of the plot library or expressions accepted by `plot::easy`

`attribute`$_1$`, attribute`$_2$`, …`

Graphical attributes of the form `AttributeName = AttributeValue`

Overloaded By

object_1

Algorithms

Technically, `plot` is not a function but a domain representing the library plot library. Thus, when calling `plot(...)`, the method `plot` is called.

See Also

MuPAD Functions

`display` | `plot::easy` | `plotfunc2d` | `plotfunc3d`

More About

- "Use Graphics"

display

Display graphical objects on the screen

Syntax

```
display(object1, <object2, …>, <attribute1, attribute2, …>)
```

Description

`display( object1, object2, ...)` displays the graphical objects object$_1$, object$_2$ etc. on the screen.

This function calls `plot::easy` for preprocessing its input.

The parameters object$_1$, object$_2$ etc. must be accepted by `plot::easy` or directly be graphical objects generated by routines of the plot library. This library provides many such objects including:

- function graphs (`plot::Function2d`, `plot::Function3d`),
- curves (`plot::Curve2d`, `plot::Curve3d`),
- points (`plot::Point2d`, `plot::Point3d`),
- lines (`plot::Line2d`, `plot::Line3d`),
- polygons (`plot::Polygon2d`, `plot::Polygon3d`),
- surfaces (of domain type `plot::Surface`)

and many more. Cf. “Example 1” on page 1-1530.

There are also many high level objects such as `plot::VectorField2d`, `plot::Ode2d`, `plot::Ode3d`, `plot::Implicit2d`, `plot::Implicit3d` etc. that can also be rendered by `plot`, `display`. Cf. “Example 2” on page 1-1532.

Graphical attributes `attribute1`, `attribute2` etc. are specified by equations of the form `AttributeName = AttributeValue`. There are several hundred such attributes that allow to modify almost any aspect of the graphics.

Note: The graphical objects `object1`, `object2` etc. must have the same dimension. A mix of 2D and 3D objects in one plot is not supported!

The command `display()` creates an empty graphical 2D scene.

Examples

Example 1

The following calls return objects representing the graphs of the sine and the cosine function on the interval [0, 2 π]:

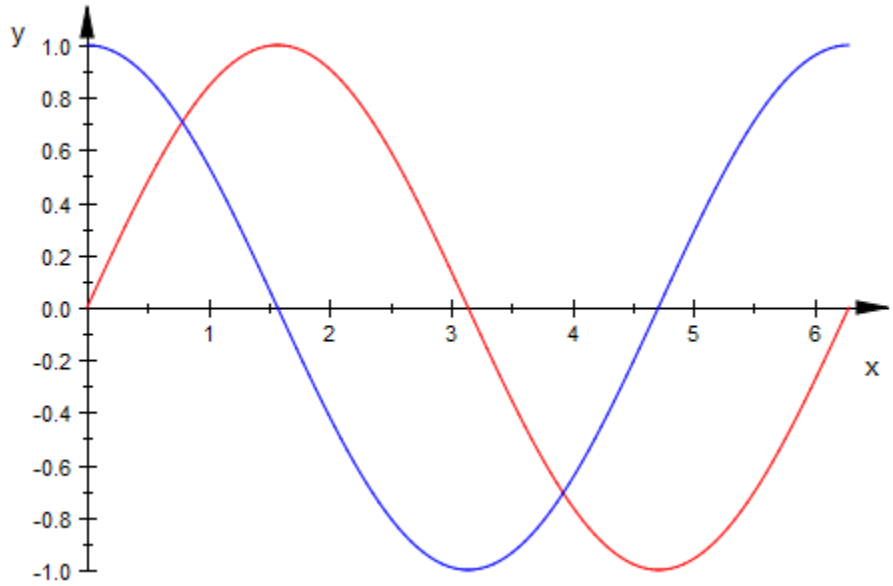
```
f1 := plot::Function2d(sin(x), x = 0..2*PI, Color = RGB::Red);
f2 := plot::Function2d(cos(x), x = 0..2*PI, Color = RGB::Blue)
```

$$\text{plot::Function2d}(\sin(x),\ x = 0..2\ \pi)$$

$$\text{plot::Function2d}(\cos(x),\ x = 0..2\ \pi)$$

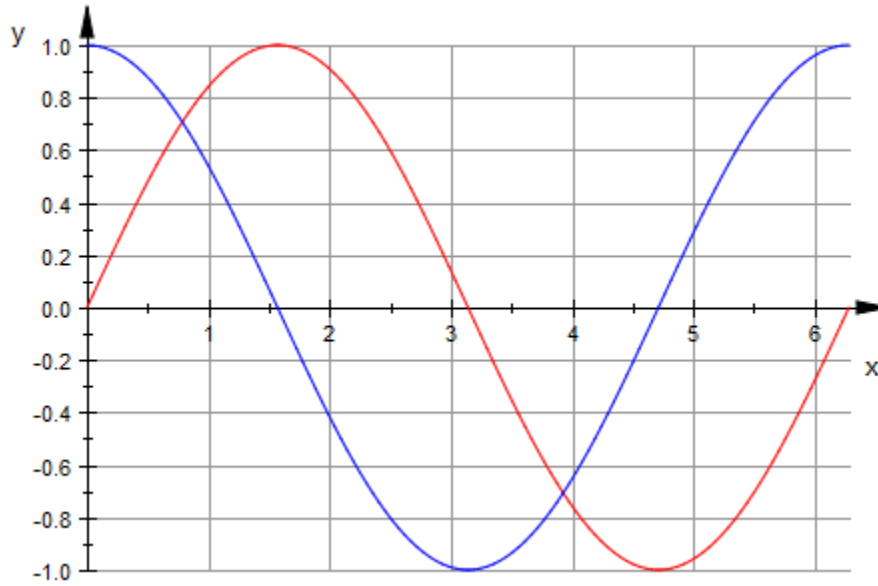
The following call renders these graphs:

```
plot(f1, f2)
```



Apart from the explicitly requested colors, this call uses the default values of all graphical attributes. If different values are desired, an arbitrary number of attributes may be passed as additional parameters to `plot`, `display`. For example, to draw grid lines in the background of the previous plot, we enter:

```
plot(f1, f2, GridVisible = TRUE)
```



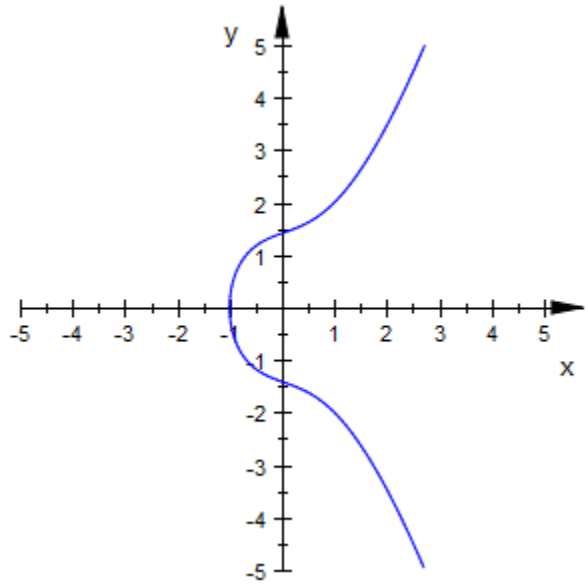

```
delete f1, f2:
```

Example 2

The plot library contains various routines for creating more complex graphical objects such as vectorfields, solution curves of ordinary differential equations, and implicitly defined curves.

For example, to plot the implicitly defined curve $x^2 + x + 2 = y^2$ with x, y from the interval [- 5, 5], we use the function `plot::Implicit2d`:

```
plot(plot::Implicit2d(x^3 + x + 2 = y^2,
                      x = -5..5, y = -5..5),
     Scaling = Constrained)
```



Here we used the `Scaling` attribute to guarantee an aspect ratio 1:1 between the x and y coordinates independent of the window size.

Parameters

`object`$_1$`, object`$_2$`, …`

2D or 3D graphical objects of the plot library or expressions accepted by `plot::easy`

`attribute`$_1$`, attribute`$_2$`, …`

Graphical attributes of the form `AttributeName = AttributeValue`

Overloaded By

object_1

Algorithms

Technically, `display` is not a function but a domain representing the library plot library. Thus, when calling `display(...)`, the method `display::new(...)` is called.

See Also

MuPAD Functions

`plot` | `plot::easy` | `plotfunc2d` | `plotfunc3d`

More About

- "Use Graphics"

plotfunc2d

Function plots in 2D

Syntax

```
plotfunc2d(f1, f2, …, <Colors = [c1, c2, …]>, <attributes>)

plotfunc2d(f1, f2, …, x = xmin .. xmax, <Colors = [c1, c2, …]>, <attributes>)

plotfunc2d(f1, f2, …, x = xmin .. xmax, a = amin .. amax, <Colors = [c1, c2, …]>, <attribut
```

Description

`plotfunc2d(f1, f2, ...)` generates a 2D plot of the univariate functions `f1`, `f2` etc.

We strongly recommend reading the introduction to `plotfunc2d` in Section 2.1 ("2D Function Graphs") of the plot document.

The functions to be plotted must not contain any symbolic parameters apart from the variable `x` and the animation parameter `a`. Exact numerical values such as `PI`, `sqrt(2)` etc. are accepted.

Animations are triggered by specifying a range $a = a_{min} .. a_{max}$ for a parameter `a` that is different from the indedependent variable `x`. Thus, in animations, both the *x*-range $x = x_{min} .. x_{max}$ as well as the animation range $a = a_{min} .. a_{max}$ must be specified. See "Example 2" on page 1-1539.

Non-real function values are ignored. See "Example 3" on page 1-1540.

Functions with singularities are handled. See "Example 4" on page 1-1541 and "Example 5" on page 1-1545. If unbounded functions are plotted, the vertical viewing range is clipped, automatically. An explicit vertical viewing range `ymin .. ymax` may be requested via `ViewingBoxYRange` = `` `y_{min}` .. `y_{max}` `` or `YRange` = `` `y_{min}` .. `y_{max}` ``.

Discontinuities and piecewise defined functions are handled. See "Example 6" on page 1-1546 and "Example 7" on page 1-1547.

The `plot` library provides the routine `plot::Function2d` which allows to create a function graph as a graphical primitive, and to combine it with other graphical objects.

A variety of graphical attributes can be specified for fine tuning the graphical output. Such attributes are passed as equations `AttributeName` = `AttributeValue` to the `plotfunc2d` command.

Section 2.3 (“Attributes for plotfunc2d and plotfunc3d”) provides an overview of the available attributes.

In particular, all attributes accepted by the graphical primitive `plot::Function2d` for function graphs are accepted by `plotfunc2d`. These attributes allow to specify the mesh for the numerical evaluation, the line width etc. The help page of plot::Function2d provides a concise list.

Further, all attributes accepted by `plot::CoordinateSystem2d` are accepted by `plotfunc2d`. These attributes include the specification of a viewing box, of the axes, their tick marks and tick labels, the coordinate type (such as linear versus logarithmic plots), grid lines etc. The help page of plot::CoordinateSystem2d provides a concise list.

Further, all attributes accepted by `plot::Scene2d` are accepted by `plotfunc2d`. These attributes include the specification of the layout of the graphical scene, the background color etc. The help page of plot::Scene2d provides a concise list.

Further, all attributes accepted by `plot::Canvas` are accepted by `plotfunc2d`. These attributes include the specification of the size of the graphics, of further layout parameters etc. The help page of plot::Canvas provides a concise list.

A graphical attribute such as `Mesh` = `500` (setting the number of mesh points for the numerical evaluation to 500) is applied to *all* functions in the call `plotfunc2d(`f_1`, `f_2`, …)`. If separate attributes are appropriate, use the equivalent call

`plot(plot::Function2d(`f_1`, `$attr_1$`), plot::Function2d(`f_2`, `$attr_2$`), …)`,

in which the attributes $attr_1$, $attr_2$ etc. can be set separately for each function.

Apart from few exceptions, `plotfunc2d` uses the standard default values for the graphical attributes (see the help page of `plot::Function2d`). The exceptions are:

- If more than one function is plotted, `plotfunc2d` automatically creates a legend. Use an explicit `LegendVisible` = `FALSE` to suppress the legend.

- `AdaptiveMesh` is set to 2, i.e., `plotfunc2d` uses adaptive function evaluation unless `AdaptiveMesh` = `0` is requested in `plotfunc2d`.
- If a parameter range such as `x = `x_{min}` .. `x_{max}`` is passed to `plotfunc2d`, the name `x` is used as the title for the horizontal axis. Pass the attribute `XAxisTitle` if a different label for the horizontal axis is desired.

Environment Interactions

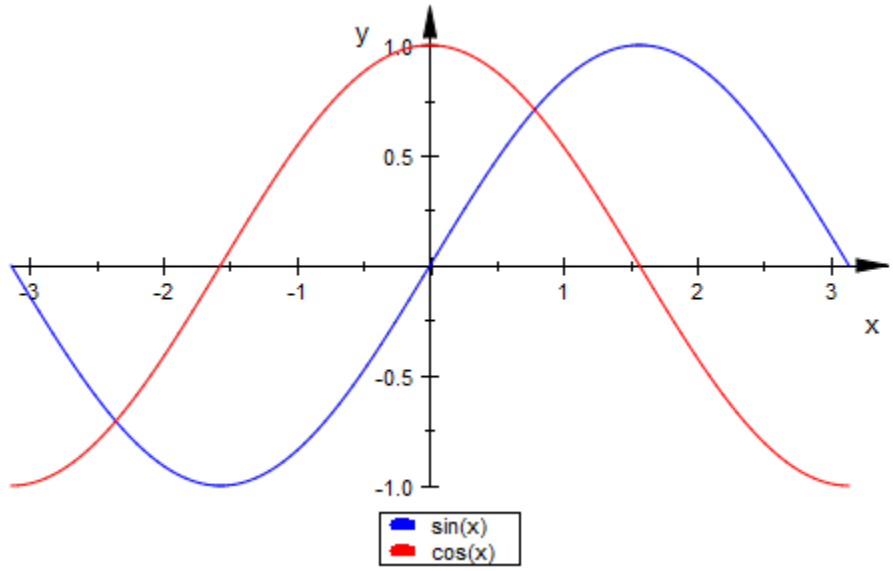
The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision. Make sure that `DIGITS` is set to a sufficiently small value (such as the default value 10) to avoid the costs of computing unnecessarily precise plot data.

Examples

Example 1

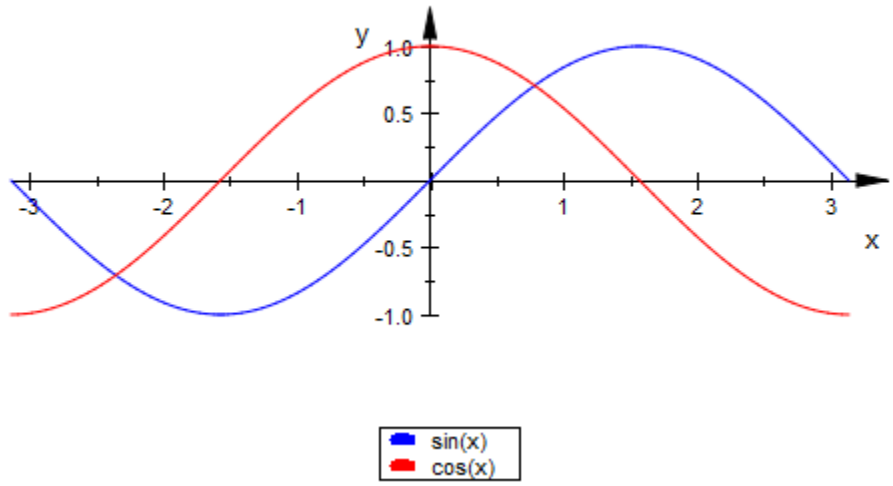
The following command draws the sine function and the cosine function on the interval [-π, π]:

```
plotfunc2d(sin(x), cos(x), x = -PI .. PI):
```



With the attribute `Scaling` = `Constrained`, the *y*-axis has the same scale as the *x*-axis:

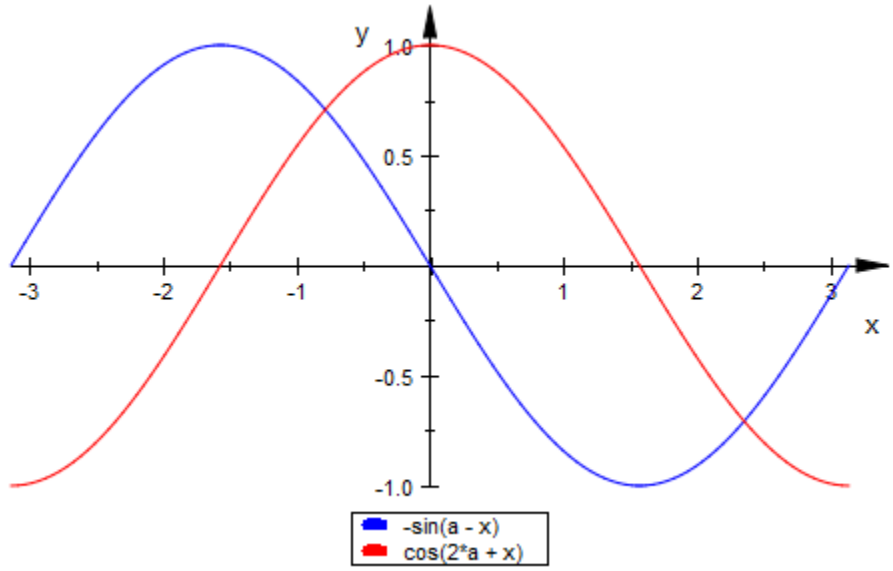
```
plotfunc2d(sin(x), cos(x), x = -PI .. PI, Scaling = Constrained):
```



Example 2

When creating an animation, a range for the independent variable *x* must be specified. An additional second range triggers the animation:

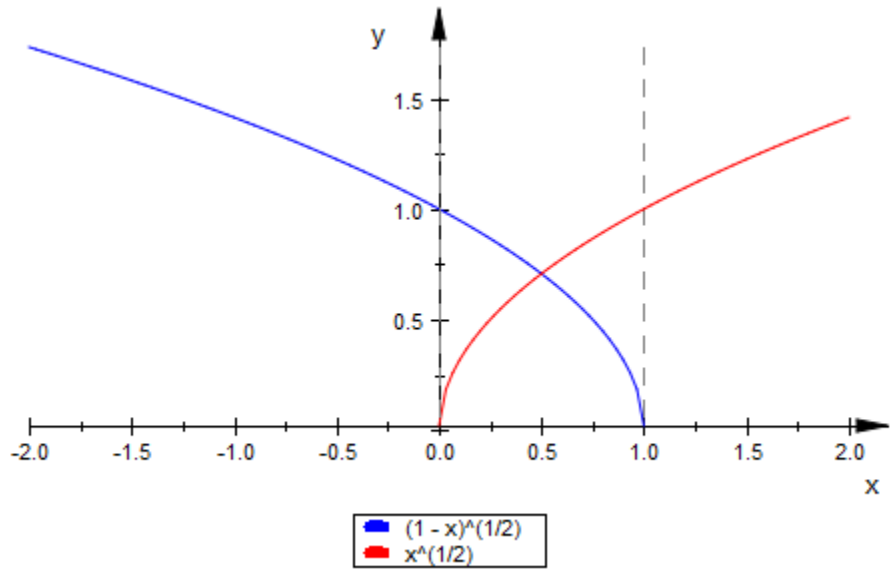
```
plotfunc2d(sin(x - a), cos(x + 2*a),
           x = -PI .. PI, a = -PI .. PI)
```



Example 3

Only real function values are plotted:

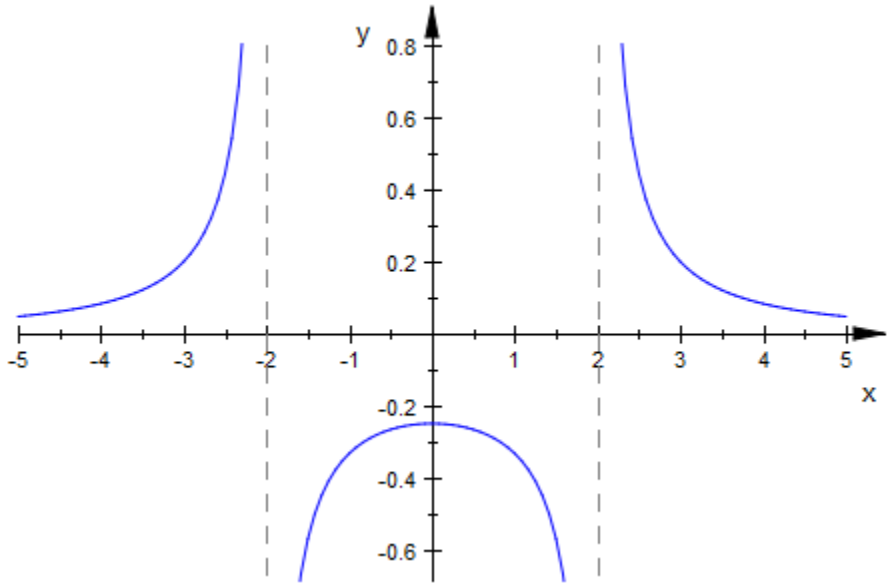
```
plotfunc2d(sqrt(1 - x), sqrt(x), x = -2 .. 2):
```



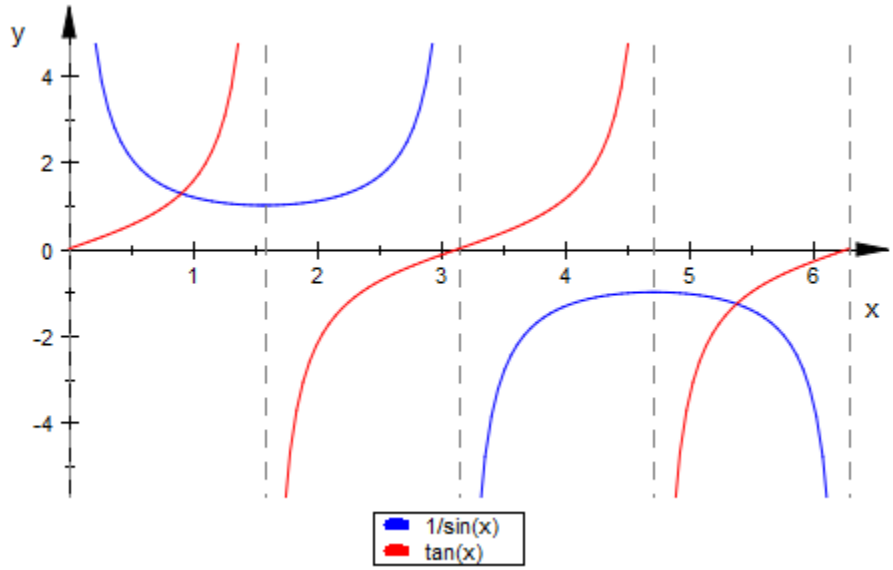
Example 4

The following functions have singularities in the specified interval:

```
plotfunc2d(x/(x^3 - 4*x), x = -5 .. 5):
```

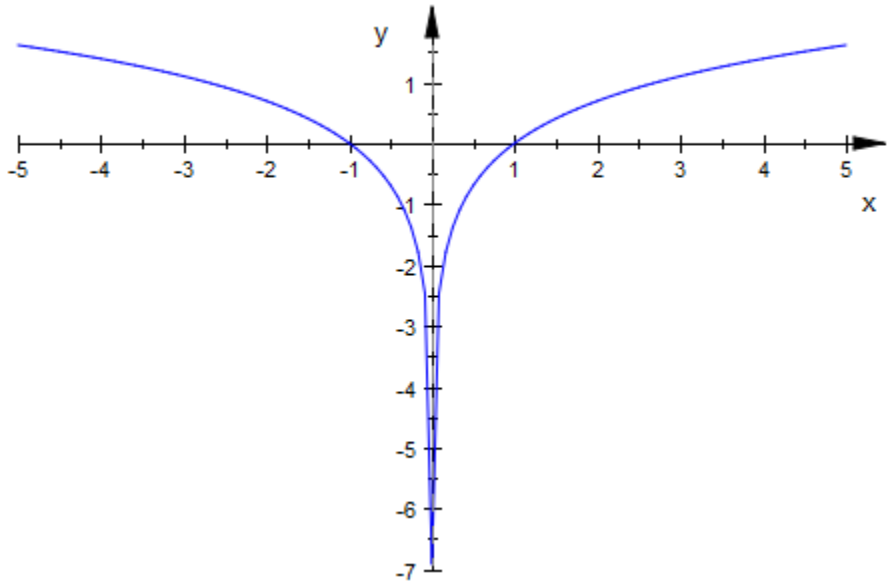



```
plotfunc2d(1/sin(x), tan(x), x = 0 .. 2*PI):
```



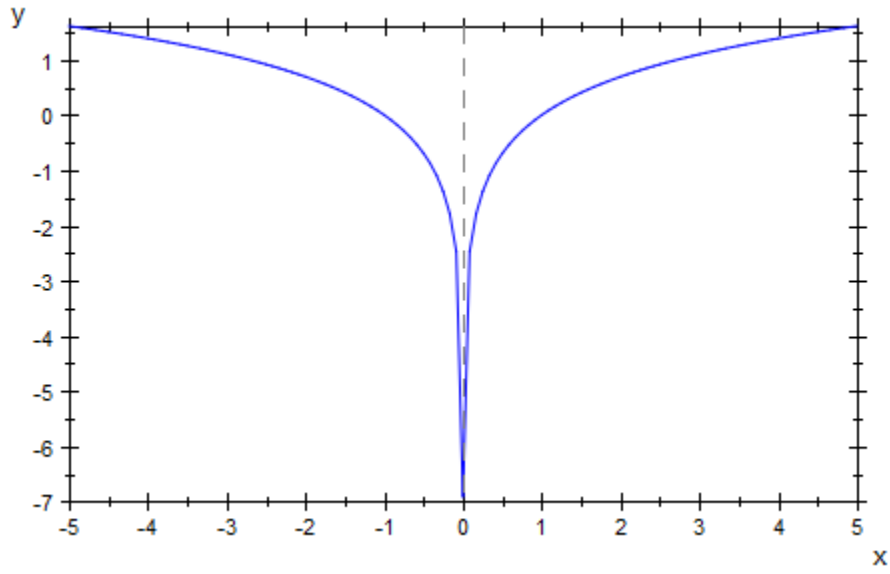
Note that the automatic clipping may in some cases lead to an incorrect impression, such as the following image where the function appears to converge to about - 4.6 (but actually goes to - ∞ for small absolute values of *x*:

```
plotfunc2d(ln(abs(x)))
```



In this case, the asymptote which points to the pole is not seen because of the axis:

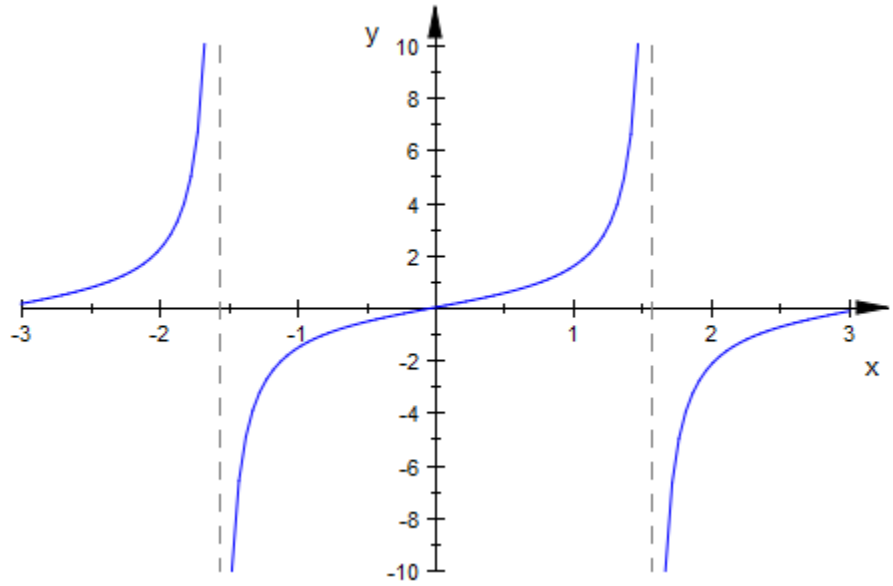
```
plotfunc2d(ln(abs(x)), Axes=Boxed)
```



Example 5

We specify a vertical range to which the function graph is restricted:

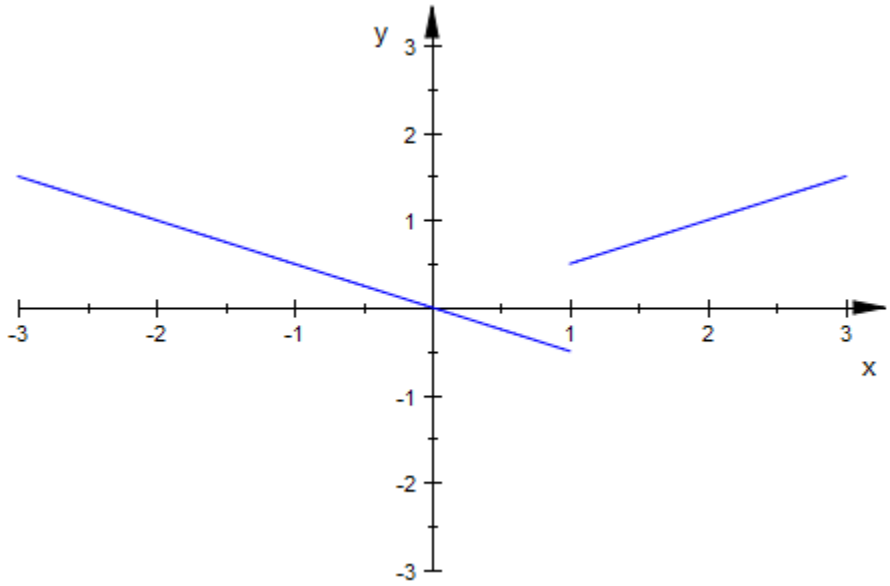
```
plotfunc2d(tan(x), x = -3 .. 3, YRange = -10 .. 10):
```



Example 6

The following function has a jump discontinuity:

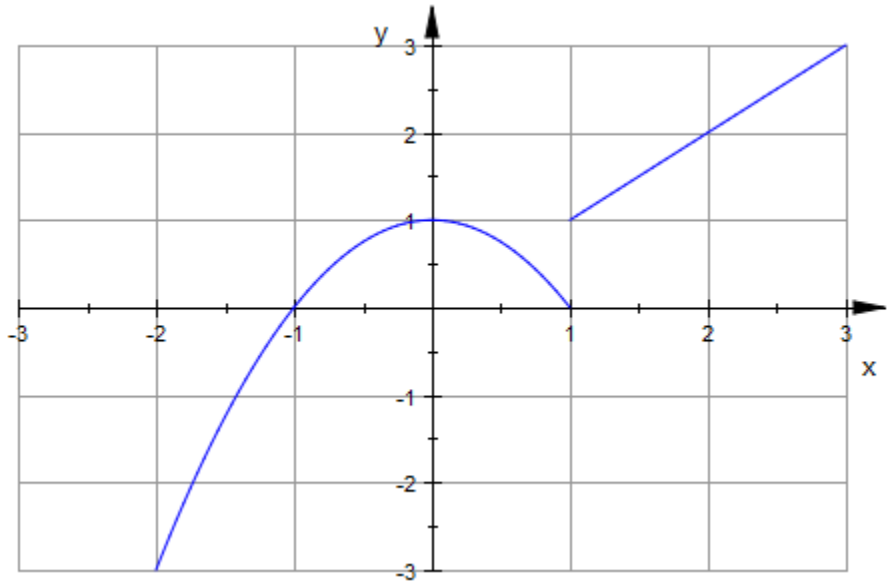
```
plotfunc2d((x^2 - x)/(2*abs(x - 1)), x = -3 .. 3,
           YRange = -3 .. 3)
```



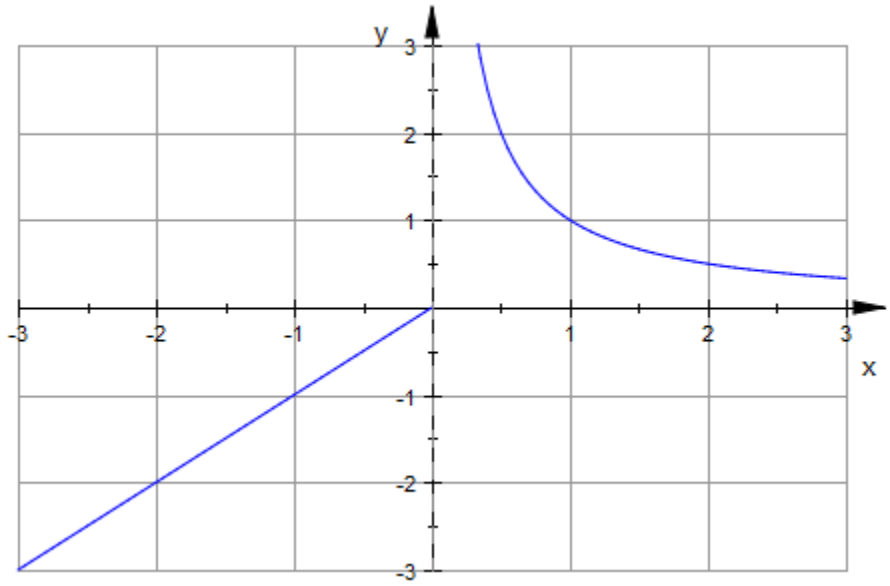
Example 7

Piecewise defined functions are handled:

```
f := piecewise([x < 1, -x^2 + 1], [x >= 1, x]):
plotfunc2d(f(x), x = -3 .. 3, YRange = -3 .. 3,
           GridVisible = TRUE, TicksDistance = 1)
```




```
f := piecewise([x <= 0, x], [x > 0, 1/x]):
plotfunc2d(f(x), x = -3 .. 3, YRange = -3 .. 3,
           GridVisible = TRUE, TicksDistance = 1)
```

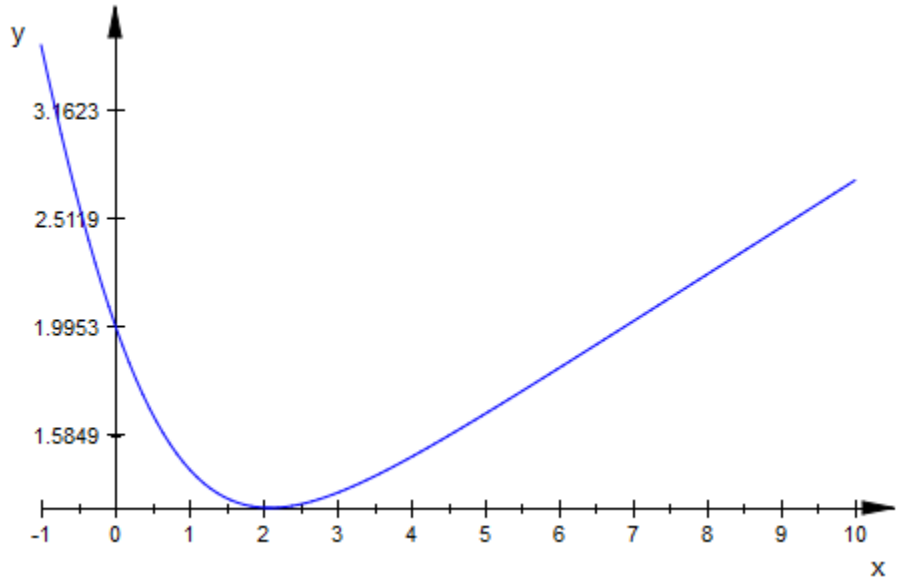



```
delete f:
```

Example 8

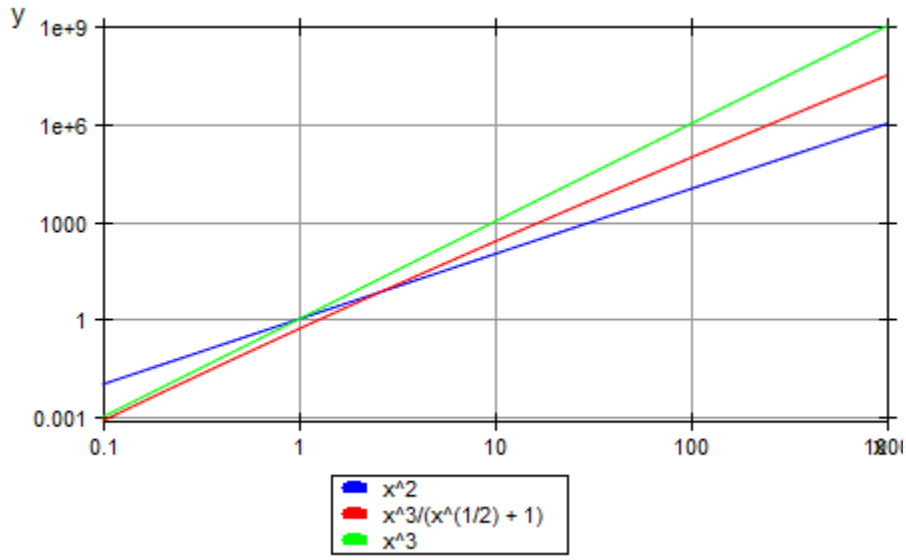
We use the attribute `CoordinateType` to create a logarithmic plot:

```
plotfunc2d(exp(x/10) + exp(-x), x = -1 .. 10,
           CoordinateType = LinLog)
```



We demonstrate various further graphical attributes in a doubly logarithmic plot:

```
plotfunc2d(x^2, x^3/(1 + x^(1/2)), x^3,
           x = 1/10 .. 10^3,
           CoordinateType = LogLog,
           Axes = Boxed,
           DiscontinuitySearch = FALSE,
           GridVisible = TRUE,
           TicksNumber = None,
           TicksAt = [[10^i $ i = -1 .. 3],
                      [10^i $ i in {-3, 0, 3, 6, 9}]
                     ]):
```



Parameters

`f1, f2, …`

The functions: arithmetical expressions or `piecewise` objects in the indeterminate `x` and the animation parameter `a`. Alternatively, procedures that accept 1 input parameter *x* or 2 input parameters *x*, *a* and return a real numerical value when the input parameters are numerical.

`x`

The independent variable: an identifier or an indexed identifier.

x_{min} `..` x_{max}

The plot range: x_{min}, x_{max} must be numerical real values or expressions of the animation parameter *a*. If not specified, the default range `x = -5 .. 5` is used.

`a`

The animation parameter: an identifier or an indexed identifier.

a_{min} .. a_{max}

The animation range: a_{min}, a_{max} must be numerical real values.

c_1, c_2, …

The colors for f_1, f_2 etc.: RGB or RGBa values. The length of the color list needs not coincide with the number of functions in the plot. The colors are used cyclically; surplus colors are ignored.

attributes

An arbitrary number of graphical attributes. Each attribute is given by an equation of the form `AttributeName` = `AttributeValue`.

Return Values

MuPAD graphics tool is called to render the graphical scene. The `null()` object is returned to the MuPAD session.

See Also

MuPAD Functions

`display` | `plot` | `plot::easy` | `plotfunc3d`

MuPAD Graphical Primitives

`plot::Function2d` | `plot::Function3d`

More About

- "2D Function Graphs: plotfunc2d"

plotfunc3d

Function plots in 3D

Syntax

```
plotfunc3d(f1, f2, …, <Colors = [c1, c2, …]>, <attributes>)

plotfunc3d(f1, f2, …, x = xmin .. xmax, <Colors = [c1, c2, …]>, <attributes>)

plotfunc3d(f1, f2, …, x = xmin .. xmax, y = ymin .. ymax, <Colors = [c1, c2, …]>, <attribute

plotfunc3d(f1, f2, …, x = xmin .. xmax, y = ymin .. ymax, a = amin .. amax, <Colors = [c1, c2,
```

Description

`plotfunc3d(f1, f2, ...)` generates a 3D plot of the bivariate functions `f1`, `f2` etc.

The functions to be plotted must not contain any symbolic parameters apart from the variables `x`, `y` and the animation parameter `a`. Exact numerical values such as `PI`, `sqrt(2)` etc. are accepted.

Animations are triggered by specifying a range `a = `a_{min}` .. `a_{max}`` for a parameter `a` that is different from the indedependent variables `x`, `y`. Thus, in animations, the *x*-range `x = `x_{min}` .. `x_{max}``, the *y*-range `y = `y_{min}` .. `y_{max}`` as well as the animation range `a = `a_{min}` .. `a_{max}`` must be specified. See “Example 2” on page 1-1555.

If unbounded functions are plotted, the range of the *z* coordinate is clipped, automatically. An explicit *z* range `z = `z_{min}` .. `z_{max}`` may be requested via `ViewingBoxZRange = `z_{min}` .. `z_{max}`` or `ZRange = `z_{min}` .. `z_{max}``.

Discontinuities and piecewise defined functions are handled. See “Example 6” on page 1-1561 and “Example 7” on page 1-1562.

The `plot` library provides the routine `plot::Function3d` which allows to create a function graph as a graphical primitive, and to combine it with other graphical objects.

A variety of graphical attributes can be specified for fine tuning the graphical output. Such attributes are passed as equations `AttributeName = AttributeValue` to the `plotfunc3d` command.

Section 2.3 (“Attributes for plotfunc2d and plotfunc3d”) provides an overview of the available attributes.

In particular, all attributes accepted by the graphical primitive `plot::Function3d` for function graphs are accepted by `plotfunc3d`. These attributes allow to specify the mesh for the numerical evaluation, the line width etc. The help page of plot::Function3d provides a concise list.

Further, all attributes accepted by `plot::CoordinateSystem3d` are accepted by `plotfunc3d`. These attributes include the specification of a viewing box, of the axes, their tick marks and tick labels, the coordinate type (such as linear versus logarithmic plots), grid lines etc. The help page of plot::CoordinateSystem3d provides a concise list.

Further, all attributes accepted by `plot::Scene3d` are accepted by `plotfunc3d`. These attributes include the specification of the layout of the graphical scene, the background color etc. The help page of plot::Scene3d provides a concise list.

Further, all attributes accepted by `plot::Canvas` are accepted by `plotfunc3d`. These attributes include the specification of the size of the graphics, of further layout parameters etc. The help page of plot::Canvas provides a concise list.

A graphical attribute such as `Mesh = [20, 20]` (setting the number of mesh points for the numerical evaluation to 20 in each direction) is applied to *all* functions in the call `plotfunc3d(`f_1`, `f_2`, …)`. If separate attributes are appropriate, use the equivalent call

`plot(plot::Function3d(`f_1`, `$attr_1$`), plot::Function3d(`f_2`, `$attr_2$`), …)`,

in which the attributes $attr_1$, $attr_2$ etc. can be set separately for each function.

Apart from few exceptions, `plotfunc3d` uses the standard default values for the graphical attributes (see the help page of `plot::Function3d`). The exceptions are:

- If more than one function is plotted, `plotfunc3d` automatically creates a legend. Use an explicit `LegendVisible = FALSE` to suppress the legend.
- If parameter ranges such as `x = xmin .. xmax`, `y = ymin .. ymax` are passed to `plotfunc3d`, the names `x`, `y` are used as the titles for the corresponding axis. Pass the attributes `XAxisTitle`, `XAxisTitle` if different labels are desired.

Environment Interactions

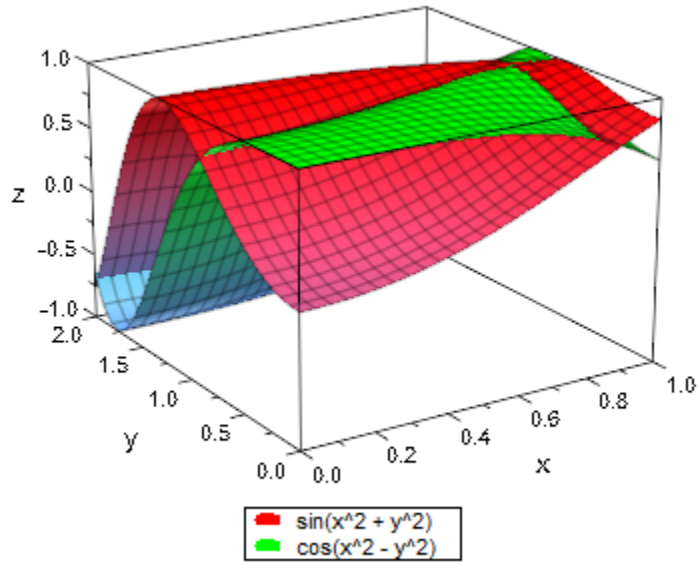
The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision. Make sure that `DIGITS` is set to a sufficiently small value (such as the default value 10) to avoid the costs of computing unnecessarily precise plot data.

Examples

Example 1

The following command draws two functions over the unit square:

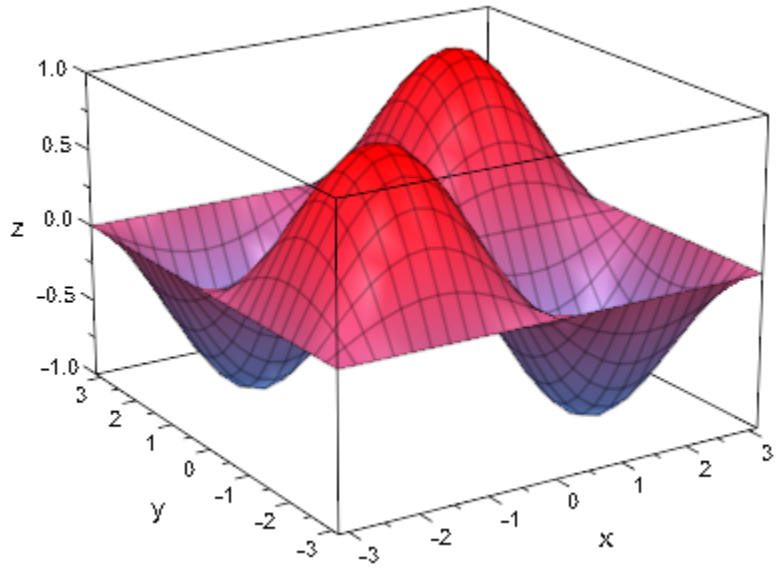
```
plotfunc3d(sin(x^2 + y^2), cos(x^2 - y^2), x = 0..1, y = 0..2)
```



Example 2

When creating an animation, ranges for the independent variables *x*, *y* must be specified. An additional third range triggers the animation:

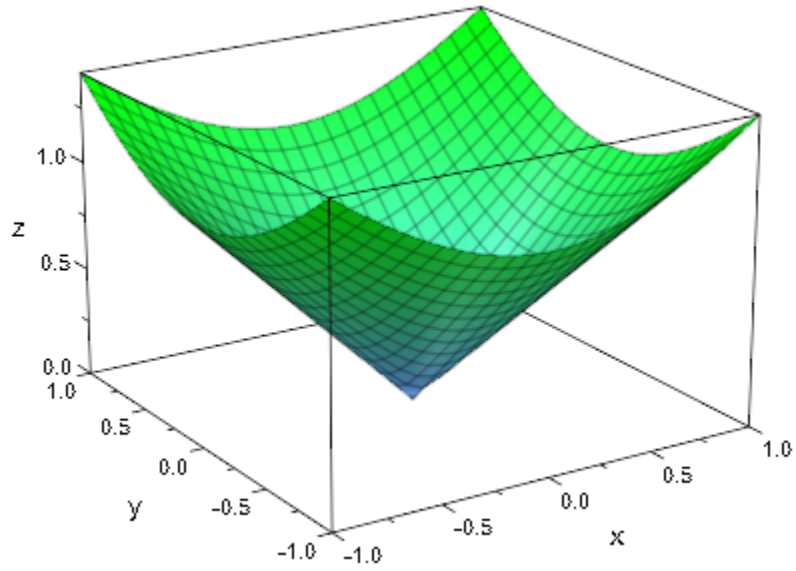
```
plotfunc3d(sin(x - a)*sin(y - a),
           x = -PI .. PI, y = -PI .. PI, a = -PI .. PI)
```



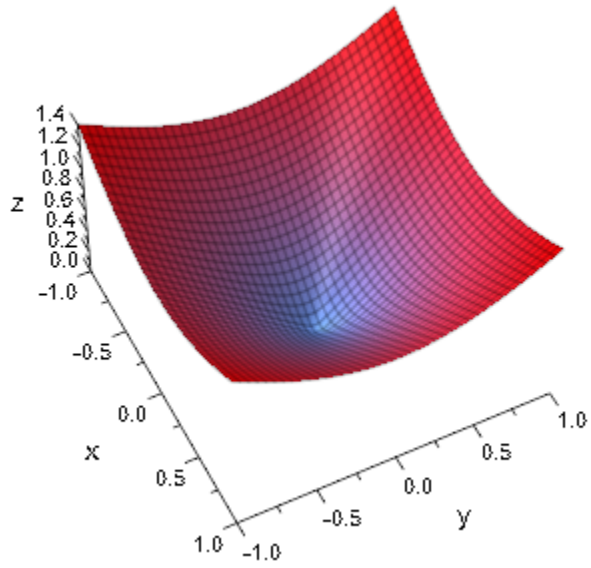
Example 3

We demonstrate the effect of various graphical attributes:

```
plotfunc3d(abs(x + I*y), x = -1..1, y = -1..1,
           FillColor = RGB::Green, TicksDistance = 0.5)
```



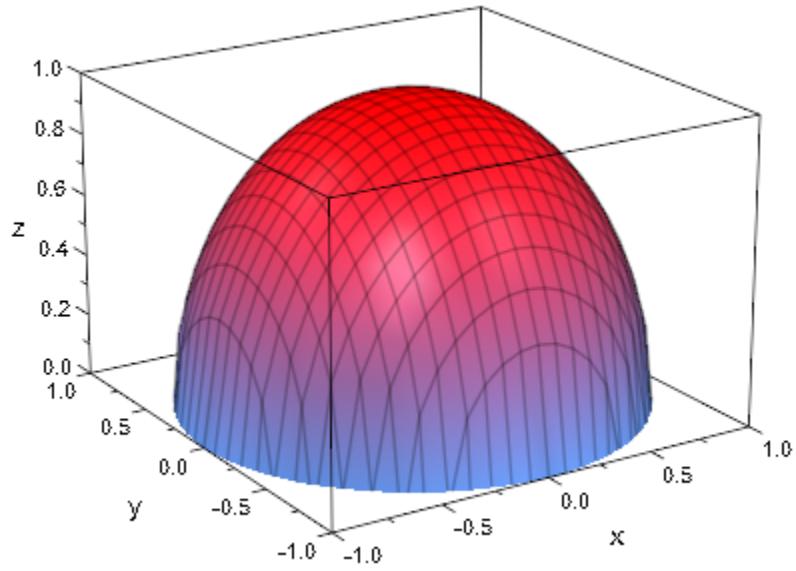

```
plotfunc3d(abs(x + I*y), x = -1..1, y = -1..1,
           Mesh = [40, 40], Axes = Frame,
           CameraDirection = [10, -5, 15])
```



Example 4

Points where the function to plot are not real-valued are left out from the plot:

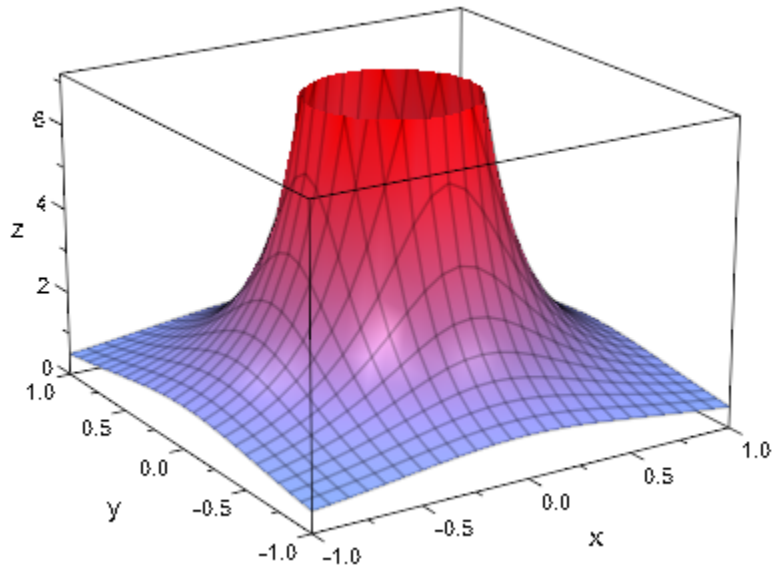
```
plotfunc3d(sqrt(1 - x^2 - y^2), x = -1..1, y = -1..1):
```



Example 5

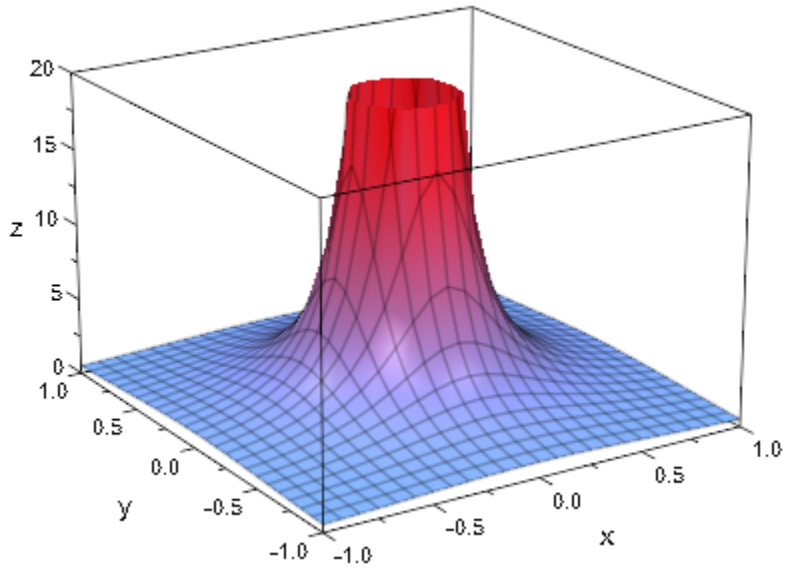
Singular functions are handled. The vertical coordinate range is automatically restricted by a heuristics:

```
plotfunc3d(1/(x^2 + y^2), x = -1..1, y = -1..1):
```



If the heuristics produces an inappropriate vertical range, you can request an appropriate range by the attribute `ViewingBoxZRange` or `ZRange`:

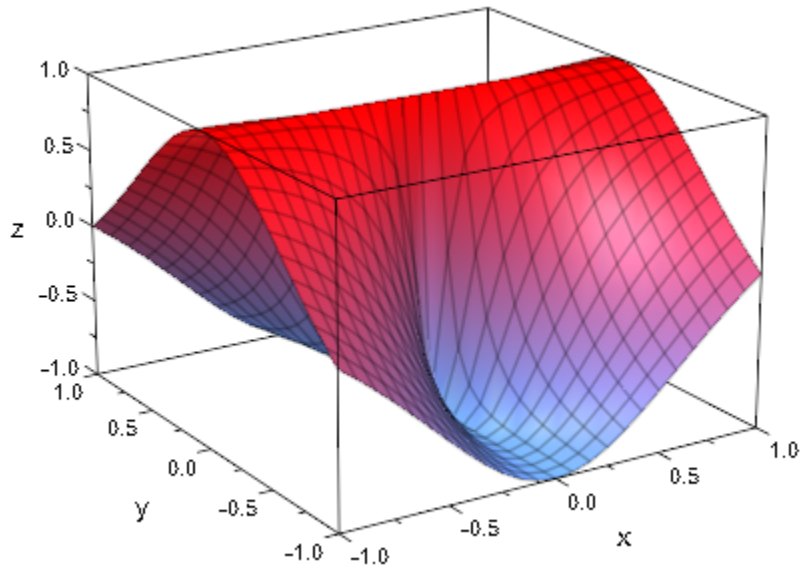
```
plotfunc3d(1/(x^2 + y^2), x = -1..1, y = -1..1,
           ZRange = 0 .. 20):
```



Example 6

The following function has a discontinuity at the origin:

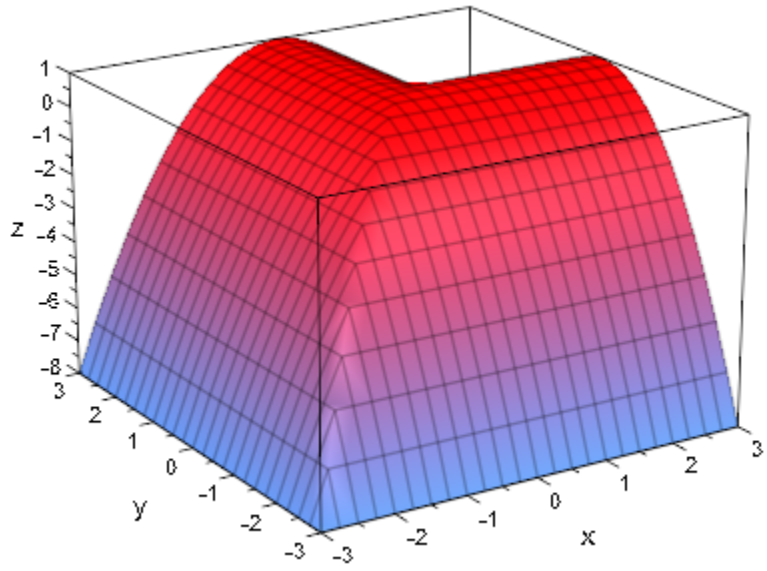
```
plotfunc3d((x^2 - y^2)/(x^2 + y^2),
           x = -1 .. 1, y = -1 .. 1)
```



Example 7

Piecewise defined functions are handled:

```
f := piecewise([x < y, 1 - x^2], [x >= y, 1 - y^2]):
plotfunc3d(f(x, y), x = -3..3, y = -3..3, TicksDistance = 1)
```




```
delete f:
```

Parameters

`f1, f2, …`

The functions: arithmetical expressions or `piecewise` objects in the indeterminates `x`, `y` and the animation parameter `a`. Alternatively, procedures that accept 2 input parameter *x*, *y* or 3 input parameters *x*, *y*, *a* and return a real numerical value when the input parameters are numerical.

`x`

The first independent variable: an identifier or an indexed identifier.

x_{min} `..` x_{max}

The range of `x`: x_{min}, x_{max} must be numerical real values or expressions of the animation parameter *a*. If not specified, the default range `x = -5 .. 5` is used.

y

The second independent variable: an identifier or an indexed identifier.

y_{min} .. y_{max}

The range of `y`: y_{min}, y_{max} must be numerical real values or expressions of the animation parameter a. If not specified, the default range `y = -5 .. 5` is used.

a

The animation parameter: an identifier or an indexed identifier.

a_{min} .. a_{max}

The animation range: a_{min}, a_{max} must be numerical real values.

c_1, c_2, …

The colors for f_1, f_2 etc.: RGB or RGBa values. The length of the color list needs not coincide with the number of functions in the plot. The colors are used cyclically; surplus colors are ignored.

attributes

An arbitrary number of graphical attributes. Each attribute is given by an equation of the form `AttributeName = AttributeValue`.

Return Values

MuPAD graphics tool is called to render the graphical scene. The `null()` object is returned to the MuPAD session.

See Also

MuPAD Functions
`display` | `plot` | `plot::easy` | `plotfunc2d`

MuPAD Graphical Primitives
`plot::Function2d` | `plot::Function3d`

More About

- "3D Function Graphs: plotfunc3d"

pochhammer

The Pochhammer symbol

Syntax

```
pochhammer(x, n)
```

Description

`pochhammer(x, n)` represents the Pochhammer symbol $(x)_n = \frac{\Gamma(x+n)}{\Gamma(x)}$.

If `n` is a positive integer, then $(x)_n = x\,(x+1)\,\ldots\,(x+n-1)$. This is extended analytically to arbitrary complex arguments via $(x)_n = \frac{\Gamma(x+n)}{\Gamma(x)}$, where *gamma* is the `gamma` function.

If both x and $x + n$ are non-positive integers, `pochhammer(x, n)` produces the limit $\lim_{t \to 0} \frac{\Gamma(x+n+t)}{\Gamma(x+t)}$.

If both `x` and `n` are numerical values, then an explicit numerical result is returned. Otherwise, a symbolic function call is returned.

If `n` is a negative integer, then the identity `pochhammer(x, n) = 1/pochhammer(x + n, -n)` is used to express the result.

The following special cases are implemented: `pochhammer(x, 0) = 1`, `pochhammer(x, 1) = x`, `pochhammer(x,-1) = 1/(x - 1)`, `pochhammer(1, n) = gamma(n + 1)`, `pochhammer(2, n) = gamma(n + 2)`.

If `n` is a positive integer, then `expand(pochhammer(x, n))` yields the expanded polynomial $x\,(x + 1)\,\ldots\,(x + n - 1)$.

If `n` is not an integer, then `expand(pochhammer(x, n))` yields a representation in terms of `gamma`.

Environment Interactions

When called with floating-point arguments, this function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

`pochhammer` returns explicit results if both arguments are numbers:

```
pochhammer(3, 5), pochhammer(3/2, 2), pochhammer(7/2, I - 1/2)
```

$$2520,\ \frac{15}{4},\ \frac{8\,\Gamma(3+i)}{15\,\sqrt{\pi}}$$

Some special cases are implemented:

```
pochhammer(x, -1), pochhammer(x, 0), pochhammer(x, 1)
```

$$\frac{1}{x-1},\ 1,\ x$$

```
pochhammer(1, n), pochhammer(2, n)
```

$$\Gamma(n+1),\ \Gamma(n+2)$$

A symbolic call is returned for other arguments:

```
pochhammer(x, 2), pochhammer(3, n), pochhammer(x + I, n)
```

$$(x)_2,\ (3)_n,\ (x+i)_n$$

Example 2

`expand` turns a symbolic `pochhammer` call into an explicit polynomial expression or rewrites it in terms of the `gamma` function if that function is known to be defined at its argument:

```
expand(pochhammer(x, 3))
```

$$x^3 + 3\,x^2 + 2\,x$$

```
expand(pochhammer(x, -3))
```

$$\frac{1}{x^3 - 6\,x^2 + 11\,x - 6}$$

```
expand(pochhammer(x, n)) assuming x>0 and n>0
```

$$\frac{\Gamma(n+x)}{\Gamma(x)}$$

```
expand(pochhammer(x + 1, n)) assuming x>0 and n>0
```

$$\frac{\Gamma(n+x)}{\Gamma(x)} + \frac{n\,\Gamma(n+x)}{x\,\Gamma(x)}$$

You can also use `rewrite` with the targets `gamma` or `fact` to rewrite `pochhammer`:

```
rewrite(pochhammer(x + 1, n), gamma)
```

$$\frac{\Gamma(n+x+1)}{\Gamma(x+1)}$$

```
rewrite(pochhammer(x + 1, n), fact)
```

$$\frac{(n+x)!}{x!}$$

Example 3

`diff` and `series` act on symbolic `pochhammer` calls:

```
diff(pochhammer(x, n), x)
```

$$(x)_n\,(\psi(n+x)-\psi(x))$$

```
diff(pochhammer(x, n), n)
```

$$\psi(n+x)\,(x)_n$$

```
series(pochhammer(x, -3), x = 2)
```

$$-\frac{1}{x-2}-(x-2)-(x-2)^3+O((x-2)^5)$$

Parameters

x

An arithmetical expression

n

An arithmetical expression

Return Values

Arithmetical expression.

Overloaded By

n, x

See Also

MuPAD Functions
`fact` | `gamma`

poles

Poles of expression or function

Syntax

```
poles(f, x)
poles(f, x = a..b)
poles(f, x, options)
poles(f, x = a..b, options)
```

Description

`poles(f, x)` finds nonremovable singularities of f. These singularities are called the poles of f. Here, f is a function of the variable x. See "Example 1" on page 1-1570.

`poles(f, x = a..b)` finds the poles in the interval (a,b). See "Example 2" on page 1-1571.

If `poles` cannot find all nonremovable singularities and cannot prove that they do not exist, it returns an unevaluated call. See "Example 3" on page 1-1571.

If `poles` can prove that f has no poles (either in the specified interval (a,b) or in the complex plane), it returns an empty set. See "Example 4" on page 1-1571.

`a` and `b` must be real numbers or infinities. If you provide complex numbers, `poles` uses an empty interval and returns an empty set.

Examples

Example 1

Find the poles of these expressions:

```
poles(1/(x - I), x);
```

```
poles(sin(x)/(x - 1), x)
```

$$\{i\}$$

$$\{1\}$$

Example 2

Find the poles of the tangent function in the interval `(-PI, PI)`:

```
poles(tan(x), x = -PI..PI)
```

$$\left\{-\frac{\pi}{2}, \frac{\pi}{2}\right\}$$

Example 3

The tangent function has an infinite number of poles. If you do not specify the interval, `poles` cannot find all of them and, therefore, returns an unevaluated call:

```
poles(tan(x), x)
```

$$\text{poles}(\tan(x), x)$$

Example 4

If `poles` can prove that the expression or function does not have any poles in the specified interval, it returns an empty set:

```
poles(tan(x), x = -1..1)
```

$$\varnothing$$

Example 5

Use `Multiple` to find the poles of this expression and their orders. Restrict the search interval to `(-pi, 10*pi)`:

```
poles(tan(x)/(x - 1)^3, x = -PI..PI, Multiple)
```

$$\left\{[1, 3], \left[-\frac{\pi}{2}, 1\right], \left[\frac{\pi}{2}, 1\right]\right\}$$

Example 6

Use `Residues` to find the poles of this expression and their residues:

```
poles(a/x^2/(x - 1), x, Residues)
```

$$\{[0, -a], [1, a]\}$$

Example 7

Use `Multiple` and `Residues` to find the poles of this expression and their orders and residues:

```
poles(a/x^2/(x - 1), x, Multiple, Residues)
```

$$\{[0, 2, -a], [1, 1, a]\}$$

Parameters

`f`

Arithmetical expression representing a function in `x`.

`x`

Identifier.

`a, b`

Real numbers (including infinities) that specify the search interval for function poles. If you do not specify the interval `(a, b)`, then `poles` uses the entire complex plane.

Options

`Multiple`

When you use this option, `poles` finds the poles of f and their orders. It returns a set of lists. Each list contains two entries: the value of a pole and its order.

See “Example 5” on page 1-1571.

`Residues`

When you use this option, `poles` finds the poles of f and their residues. It returns a set of lists. Each list contains two entries: the value of a pole and its residue.

See “Example 6” on page 1-1572.

Return Values

Set or set of lists. Without the options, `poles` returns a set containing the values of poles. With `Multiple` or `Residues`, it returns a set of lists. Each list contains the value of a pole and its order or residue, respectively. With both options, `poles` returns a set of lists. Each list contains the value of a pole, its order, and residue.

See Also

MuPAD Functions

`discont` | `limit` | `solve`

poly, Expr, IntMod

Create a polynomial

Syntax

```
poly(f, <[x1, x2, …]>, <ring>)

poly(p, <[x1, x2, …]>, <ring>)

poly(list, [x1, x2, …], <ring>)

poly(coeffs, [x], <ring>)
```

Description

`poly(f)` converts a polynomial expression `f` to a polynomial of the kernel domain `DOM_POLY`.

The kernel domain `DOM_POLY` represents polynomials. The arithmetic for this data structure is more efficient than the arithmetic for polynomial expressions. Moreover, this domain allows you to use special coefficient rings that cannot be represented by expressions. The function `poly` is the tool for generating polynomials of this type.

`poly(f, [x1, x2, ...], ring)` converts the expression `f` to a polynomial in the indeterminates `x1, x2, ...` over the specified coefficient ring. The `poly` function does not require an expanded form of the expression `f`. The function internally expands expressions.

If you do not specify indeterminates, MuPAD searches for them internally. If MuPAD cannot identify indeterminates, it returns `FAIL`.

By default, the `poly` function uses the coefficient ring of arbitrary MuPAD expressions. In this case, you can use arbitrary MuPAD expressions as coefficients.

If the `poly` function cannot convert an expression to a polynomial, the function returns `FAIL`. See “Example 10” on page 1-1581.

If `f` is a domain element, the system calls `f::dom::poly` for the conversion into a polynomial. If `f` contains domain elements, the system recursively calls `f::dom::poly` for domain elements inside `f`. See "Example 11" on page 1-1582.

`poly(p, [x1, x2, ...], ring)` converts a polynomial `p` of the type `DOM_POLY` to a polynomial in the indeterminates `x1, x2, ...` over the specified coefficient ring. The indeterminates and the coefficient ring are part of the data structure `DOM_POLY`. Using this function call, you can change the indeterminates and the coefficient ring of a polynomial.

If you do not specify indeterminates, `poly` uses the indeterminates of the original polynomial `p`.

If you do not specify a coefficient ring, `poly` uses the ring of the original polynomial `p`.

See "Example 8" on page 1-1581 and "Example 9" on page 1-1581.

`poly(list, [x])` converts a list of coefficients $[a_0, a_1, a_2, \ldots]$ to a univariate polynomial $a_0 + a_1 x + a_2 x^2 + \ldots$. See "Example 3" on page 1-1578.

For a univariate polynomial `p`, the call `poly(list, [x])` converts the result of the call `coeff(p, All)` back to a polynomial.

`poly(list, [x1, x2, ...], ring)` converts a list of coefficients and exponents to a polynomial in the indeterminates `x1, x2, ...` over the specified coefficient ring. See "Example 4" on page 1-1579 and "Example 7" on page 1-1580.

This call is the fastest method to create polynomials of the type `DOM_POLY` because the input already has the form that MuPAD uses internally.

The list must contain an element for each nonzero monomial of the polynomial. Therefore, you must use sparse input involving only nonzero terms. In particular, an empty list results in the zero polynomial.

Each element of the list must be a list with two elements: the coefficient of the monomial and the exponent (or exponent vector). For a univariate polynomial in the variable x, the list

$$[[c_1, e_1], [c_2, e_2], \ldots]$$

corresponds to $c_1 x^{e_1} + c_2 x^{e_2} + \ldots$. For a multivariate polynomial, the exponent vectors are lists containing the exponents of all indeterminates of the polynomial. The order of

the exponents must be the same as the order given by the list of indeterminates. For a multivariate polynomial in the variables x_1, x_2, the term list

$$[[c_1, [e_{11}, e_{12}]], [c_2, [e_{21}, e_{22}]], \ldots]$$

corresponds to $c_1 x_1^{e_{11}} x_2^{e_{12}} + c_2 x_1^{e_{21}} x_2^{e_{22}} + \ldots$.

The order of the elements of the term list does not affect the resulting polynomial. If you provide multiple entries corresponding to the same term, `poly` adds the coefficients.

This call lets you restore polynomials from the term lists returned by `poly2list`.

The position of the indeterminates in the input list `[x1, x2, ...]` determines their order in the resulting polynomial. If you do not specify indeterminates, MuPAD searches the expression `f` for possible indeterminates and determines their order. See “Example 2” on page 1-1578.

You can perform arithmetical operations on polynomials that have the same indeterminates and the same coefficient ring. Also, you can perform arithmetical operations on polynomials and arithmetical expressions. When you operate on a polynomial and an arithmetical expression, MuPAD internally converts that arithmetical expression to a polynomial and performs the calculation. See “Example 1” on page 1-1577.

The `poly` function does not limit acceptable indeterminates to identifiers or indexed identifiers. You can use any expression (except for rational expressions) as an indeterminate. For example, `poly` accepts the expressions `sin(x)` and `f(x)` as indeterminates. See “Example 5” on page 1-1579.

After creating a polynomial, the `poly` function does not evaluate the coefficients of the polynomial. If the coefficients contain free identifiers, `poly` does not replace these identifiers with their values. See “Example 12” on page 1-1583.

If any domain of type `DOM_DOMAIN` provides arithmetical operations, you can use that domain as a coefficient ring. See the “Background” section for details.

If you specify a coefficient domain, MuPAD accepts only the elements of that domain as coefficients of the polynomial. On input, `poly` tries to convert a polynomial expression `f` to a polynomial over the coefficient ring. For some coefficient rings, you cannot use arithmetical expressions to represent a polynomial. The reason is that multiplication

with the indeterminates can be an invalid operation in the ring. In such cases, you can define the polynomial by using a term list. See “Example 7” on page 1-1580.

Examples

Example 1

The `poly` function creates a polynomial from a polynomial expression:

```
p := poly(2*x*(x + 3))
```

$$\mathrm{poly}(2x^2 + 6x, [x])$$

The operators `*`, `+`, `-` and `^` work on polynomials:

```
p^2 - p*(p + 1)
```

$$\mathrm{poly}(-2x^2 - 6x, [x])$$

You can multiply a polynomial by an arithmetical expression. MuPAD internally converts the expression to a polynomial of the appropriate type, and then multiplies polynomials. For example, multiply the polynomial `p` by the constant 5:

```
p*5
```

$$\mathrm{poly}(10x^2 + 30x, [x])$$

Now, multiply the polynomial `p` by `x - 1`:

```
p*(x - 1)
```

$$\mathrm{poly}(2x^3 + 4x^2 - 6x, [x])$$

If MuPAD cannot convert the expression to a polynomial of the appropriate type, the arithmetical operation between a polynomial and this expression fails:

```
p*(1/x - 1)
```

```
Error: The argument is invalid. [_mult]
```

```
delete p:
```

Example 2

You can create a polynomial with parameters. In the following call, `y` is a parameter (not an indeterminate):

```
poly((x*(y + 1))^2, [x])
```

$$\mathrm{poly}((y+1)^2\,x^2,\,[x])$$

If you do not specify indeterminates, MuPAD tries to find indeterminates automatically. The following call converts a multivariate expression to a multivariate polynomial:

```
poly((x*(y + 1))^2)
```

$$\mathrm{poly}(x^2\,y^2+2\,x^2\,y+x^2,\,[x,\,y])$$

Now, specify the order of the indeterminates explicitly:

```
poly((x*(y + 1))^2, [y, x])
```

$$\mathrm{poly}(y^2\,x^2+2\,y\,x^2+x^2,\,[y,\,x])$$

Example 3

Use the `poly` function to convert the following list of coefficients to a univariate polynomial in `x`. The first entry of the list produces the term with the zero exponent. The last entry produces the term with the highest exponent:

```
p := poly([1, 2, 3, 4, 5], [x])
```

$$\mathrm{poly}(5\,x^4+4\,x^3+3\,x^2+2\,x+1,\,[x])$$

To revert the ordering of the coefficients in a polynomial, use the `revert` function:

```
revert(p)
```

$$\text{poly}(x^4 + 2\,x^3 + 3\,x^2 + 4\,x + 5, [x])$$

Example 4

Create the following polynomials by term lists:

```
poly([[c2, 3], [c1, 7], [c3, 0]], [x])
```

$$\text{poly}(\text{c1}\,x^7 + \text{c2}\,x^3 + \text{c3}, [x])$$

If you provide multiple coefficients corresponding to the same exponent, `poly` adds those coefficients:

```
poly([[c2, 3], [c1, 7], [c3, 0], [a, 3]], [x])
```

$$\text{poly}(\text{c1}\,x^7 + (a + \text{c2})\,x^3 + \text{c3}, [x])$$

For multivariate polynomials, specify exponent vectors by lists:

```
poly([[c1, [2, 2]], [c2, [2, 1]], [c3, [2, 0]]], [x, y])
```

$$\text{poly}(\text{c1}\,x^2\,y^2 + \text{c2}\,x^2\,y + \text{c3}\,x^2, [x, y])$$

Example 5

You can use expressions as indeterminates:

```
poly(f(x)*(f(x) + x^2))
```

$$\text{poly}(x^2\,f(x) + f(x)^2, [x, f(x)])$$

Example 6

The residue class ring `IntMod(7)` is a valid coefficient ring:

```
p := poly(9*x^3 + 4*x - 7, [x], IntMod(7))
```

$$\text{poly}(2x^3 - 3x, [x], \text{IntMod}(7))$$

For computations that involve polynomials over this ring, MuPAD uses modular arithmetic:

```
p^3
```

$$\text{poly}(x^9 - x^7 - 2x^5 + x^3, [x], \text{IntMod}(7))$$

However, MuPAD does not return coefficients as elements of a special domain. Instead, it returns coefficients as plain integers of the type `DOM_INT`:

```
coeff(p)
```

$$2, -3$$

```
delete p:
```

Example 7

To create the following polynomial, combine the input syntax that uses term lists with a specified coefficient ring:

```
poly([[9, 3], [4, 1], [-2, 0]], [x], IntMod(7))
```

$$\text{poly}(2x^3 - 3x - 2, [x], \text{IntMod}(7))$$

MuPAD interprets the input coefficients as elements of the coefficient domain. For example, conversions such as `9 mod 7 = 2 mod 7` occur on input. You also can use the domain `Dom::IntegerMod(7)` to define an equivalent polynomial. If you use `IntMod(7)`, MuPAD uses the symmetric modulo function `mods` and represents the coefficients by the numbers - 3, …, 3. If you use `Dom::IntegerMod(7)`, MuPAD uses the positive modulo function `modp` and represents the coefficients by the numbers 0, …, 6:

```
poly([[9, 3], [4, 1], [-2, 0]], [x], Dom::IntegerMod(7))
```

$$\text{poly}(2x^3 + 4x + 5, [x], \text{Dom::IntegerMod}(7))$$

The domain `Dom::IntegerMod(7)` does not allow multiplication with identifiers:

```
c := Dom::IntegerMod(7)(3)
```

$$3 \bmod 7$$

```
poly(c*x^2, [x], Dom::IntegerMod(7))
```

FAIL

Instead, use the term list to specify the polynomial:

```
poly([[c, 2]], [x], Dom::IntegerMod(7))
```

$$\mathrm{poly}(3\,x^2, [x], \mathrm{Dom::IntegerMod}(7))$$

```
delete c:
```

Example 8

Change the indeterminates in a polynomial:

```
p:= poly(((a + b)*x - a^2)*x, [x]):
p, poly(p, [a, b])
```

$$\mathrm{poly}((a+b)\,x^2+(-a^2)\,x, [x]), \mathrm{poly}((-x)\,a^2+x^2\,a+x^2\,b, [a, b])$$

Example 9

Change the coefficient ring of a polynomial:

```
p := poly(-4*x + 5*y - 5, [x, y], IntMod(7)):
p, poly(p, IntMod(3))
```

$$\mathrm{poly}(3\,x-2\,y+2, [x, y], \mathrm{IntMod}(7)), \mathrm{poly}(y-1, [x, y], \mathrm{IntMod}(3))$$

Example 10

Create a polynomial over the coefficient ring `Dom::Float`:

```
poly(3*x - y, Dom::Float)
```

$$\text{poly}(3.0\,x - 1.0\,y, [x, y], \text{Dom::Float})$$

The identifier `y` cannot appear in coefficients from this ring because it cannot be converted to a floating-point number:

```
poly(3*x - y, [x], Dom::Float)
```

FAIL

Example 11

You can overload `poly` by its first operand. For example, create a domain `polyInX` that represents polynomials in `x`:

```
domain polyInX
  new   := () -> new(dom, poly(args(), [x]));
  print := p -> expr(extop(p, 1));
  poly  := p ->
           if args(0) = 1 then
             print(Unquoted, "polyInX::poly called with 1 argument");
             extop(p, 1);
           else
             print(Unquoted,
                   "polyInX::poly called with more than 1 argument");
             poly(extop(p, 1),
                  args(2..args(0)));
           end;
end_domain:
p := polyInX(3*x^2-2)
```

$$3\,x^2 - 2$$

You can convert the elements of `polyInX` into polynomials of the type `DOM_POLY`. The `poly` function calls the `poly` method of the domain:

```
poly(p)
```

```
polyInX::poly called with 1 argument
```

$$\mathrm{poly}(3\,x^2 - 2, [x])$$

By reacting to additional arguments, the overloading defined above also allows you to create polynomials over other coefficient rings:

```
poly(p, [x], IntMod(2))
```

```
polyInX::poly called with more than 1 argument
```

$$\mathrm{poly}(x^2, [x], \mathrm{IntMod}(2))$$

Example 12

Create a polynomial with coefficients containing the identifier `y`. Although you assign the value 1 to `y`, MuPAD does not substitute the new value into the polynomial:

```
f := poly(x^2 - y, [x]):
y := 1:
eval(f)
```

$$\mathrm{poly}(x^2 - y, [x])$$

You can evaluate the coefficients explicitly. Use the `mapcoeffs` function to apply `eval` to the coefficients of the polynomial:

```
f := mapcoeffs(f, eval)
```

$$\mathrm{poly}(x^2 - 1, [x])$$

Parameters

`f`

A polynomial expression

x_1, x_2, …

The indeterminates of the polynomial: typically, identifiers or indexed identifiers.

ring

The coefficient ring: either `Expr`, or `IntMod(n)` with some integer `n` greater than 1, or a domain of type `DOM_DOMAIN`. The default is the ring `Expr` of arbitrary MuPAD expressions.

p

A polynomial of type `DOM_POLY` generated by `poly`

list

A list containing coefficients and exponents

coeffs

A list containing coefficients of a univariate polynomial

x

The indeterminate of a univariate polynomial

Options

Expr

The default ring `Expr` represents arbitrary MuPAD expressions. Mathematically, this ring coincides with `Dom::ExpressionField()`. However, MuPAD operates differently on the polynomials created over `Expr` and the polynomials created over `Dom::ExpressionField()`. In particular, MuPAD performs arithmetic operations for polynomials over the ring `Expr` faster.

IntMod

The ring `IntMod(n)` represents the residue class ring $\mathbb{Z}_n$, using the symmetrical representation. Here, `n` is an integer greater than 1. Mathematically, this ring coincides with `Dom::IntegerMod(n)`. However, MuPAD operates differently on the polynomials

created over `IntMod(n)` and the polynomials created over `Dom::IntegerMod(n)`. In particular, MuPAD performs arithmetic operations for polynomials over the ring `IntMod` faster. Also, for polynomials over `IntMod`, `coeff` and similar functions return requested coefficients as integers of the type `DOM_INT`. See “Example 6” on page 1-1579, “Example 7” on page 1-1580, and “Example 9” on page 1-1581.

Return Values

Polynomial of the domain type `DOM_POLY`. If conversion to a polynomial is not possible, the return value is `FAIL`.

Overloaded By

f

Algorithms

To use a domain as a coefficient, the domain must contain the following:

- The entry `"zero"` that provides the neutral element with respect to addition.
- The entry `"one"` that provides the neutral element with respect to multiplication.
- The method `"_plus"` that adds domain elements.
- The method `"_negate"` that returns the inverse with respect to addition.
- The method `"_mult"` that multiplies domain elements.
- The method `"_power"` that computes integer powers of a domain element. Call this method with the domain element as the first argument and an integer as the second argument.

In addition, you must define the following methods. Functions (such as `gcd`, `diff`, `divide`, `norm` and so on) call these methods:

- The method `"gcd"` that returns the greatest common divisor of domain elements.
- The method `"diff"` that differentiates a domain element with respect to a variable.
- The method `"_divide"` that divides two domain elements. It must return `FAIL` if division is not possible.

- The method `"norm"` that computes the norm of a domain element and returns it as a number.
- The method `"convert"` that converts an expression to a domain element. The method must return `FAIL` if such conversion is not possible.

  The system calls this method to convert the coefficients of polynomial expressions to coefficients of the specified domain. If this method does not exist, you can specify the coefficients only by using domain elements.
- The method `"expr"` that converts a domain element to an expression.

  The system function `expr` calls this method to convert a polynomial over the coefficient domain to a polynomial expression. If this method does not exist, `expr` inserts domain elements into the expression.

You can convert a polynomial over a certain coefficient domain into a polynomial over the same domain, but a different set of indeterminates. This conversion is much more efficient when the domain has the axiom `Ax::indetElements`. MuPAD implicitly assumes that this axiom holds for the domain `IntMod(n)`, but not for `Expr`.

Internally, MuPAD stores polynomials of the type `DOM_POLY` in a sparse representation and uses machine integers for the exponents. This method implies that in a 32-bit environment, the exponent of each variable in each monomial cannot exceed $2^{31} - 1$.

See Also

MuPAD Domains

`Dom::DistributedPolynomial` | `Dom::MultivariatePolynomial` | `Dom::Polynomial` | `Dom::UnivariatePolynomial`

MuPAD Functions

`coeff` | `collect` | `degree` | `degreevec` | `divide` | `evalp` | `expr` | `factor` | `gcd` | `ground` | `indets` | `lcoeff` | `ldegree` | `lmonomial` | `lterm` | `mapcoeffs` | `monomials` | `nterms` | `nthcoeff` | `nthmonomial` | `nthterm` | `poly2list` | `RootOf` | `tcoeff`

poly2list

Convert a polynomial to a list of terms

Syntax

```
poly2list(p)
```

```
poly2list(f, <vars>)
```

Description

`poly2list(p)` returns a term list containing the coefficients and exponent vectors of the polynomial `p`.

The returned term list is a list where each element represents a monomial of the polynomial with non-zero coefficient. The monomials are also represented as lists, each containing two elements: The first element is the coefficient and the second the exponent or exponent vector of the monomial. If the polynomial is univariate, exponents are returned, otherwise exponent vectors are returned. Exponent vectors have the same form as returned by the function `degreevec`. A zero polynomial results in an empty list.

The elements of the term list are sorted lexicographically according to the exponent vectors. This is also the ordering used internally for the terms of polynomials.

`poly2list(f, vars)` is equivalent to `poly2list(poly(f, vars))`: First, the polynomial expression `f` is converted to a polynomial in the variables `vars` over the expressions. Then that polynomial is converted to a term list. If the variables `vars` are not given, the free identifiers contained in `f` are used as variables. See `poly` about details on how the expression is converted to a polynomial. `FAIL` is returned if the expression cannot be converted to a polynomial.

Examples

Example 1

The following expressions define univariate polynomials. Thus the term lists contain exponents and not exponent vectors:

```
poly2list(2*x^100 + 3*x^10 + 4)
```

$$[[2, 100], [3, 10], [4, 0]]$$

```
poly2list(2*x*(x + 1)^2)
```

$$[[2, 3], [4, 2], [2, 1]]$$

Specification of a list of indeterminates allows to distinguish symbolic parameters from the indeterminates:

```
poly2list(a*x^2 + b*x + c, [x])
```

$$[[a, 2], [b, 1], [c, 0]]$$

Example 2

In this example the polynomial is bivariate, thus exponent vectors are returned:

```
poly2list((x*(y + 1))^2, [x, y])
```

$$[[1, [2, 2]], [2, [2, 1]], [1, [2, 0]]]$$

Example 3

In this example a polynomial of domain type DOM_POLY is given. This form must be used if the polynomial has coefficients that does not consist of expressions:

```
poly2list(poly(-4*x + 5*y - 5, [x, y], IntMod(7)))
```

$$[[3, [1, 0]], [-2, [0, 1]], [2, [0, 0]]]$$

Parameters

p

A polynomial of type DOM_POLY

`f`

A polynomial expression

`vars`

A list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

Return Values

List containing the coefficients and exponent vectors of the polynomial. `FAIL` is returned if a given expression cannot be converted to a polynomial.

See Also

MuPAD Functions

`coeff` | `coerce` | `degree` | `degreevec` | `lcoeff` | `monomials` | `poly` | `tcoeff`

polylog

Polylogarithm function

Syntax

```
polylog(n, x)
```

Description

`polylog(n,x)` represents the polylogarithm function $Li_n(x)$ of index n at the point x.

For a complex number x of modulus $|x| < 1$, the polylogarithm function of index n is defined as

$$\mathrm{Li}_n(x) = \sum_{k=1}^{\infty} \frac{x^k}{k^n}$$

.

This function is extended to the whole complex plane by analytic continuation. Do not confuse the polylogarithms Li_n with the integral logarithm function `Li` which is displayed using the same symbol (without an index).

If `n` is an integer and `x` a floating-point number, then a floating-point result is computed.

If `n` is an integer less or equal to 1, then an explicit expression is returned for any input parameter `x`. If `n` is an integer larger than 1 or if `n` is a symbolic expression, then an unevaluated call of `polylog` is returned, unless `x` is a floating-point number. If `n` is a numerical value, but not an integer, then an error occurs.

Some special values for $n = 2$ are implemented (cf. `dilog`). The values $Li_n(0) = 0$ and $\mathrm{Li}_n(1) = \zeta(n)$ are implemented for any n. Furthermore, $\mathrm{Li}_n(-1) = \left(2^{1-n} - 1\right)\zeta(n)$ for any $n \neq 1$.

$Li_n(x)$ has a singularity at the point $x = 1$ for indices $n \leq 1$. For indices $n \geq 1$, the point $x = 1$ is a branch point. The branch cut is the real interval $[1, \infty)$. A jump occurs when crossing this cut. Cf. “Example 2” on page 1-1592.

Mathematically, `polylog(2,x)` coincides with `dilog(1-x)`.

Environment Interactions

When called with a floating-point argument `x`, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Explicit results are returned for integer indices $n \le 1$:

```
polylog(-5, x), polylog(-1, x), polylog(0, x), polylog(1, x)
```

$$\frac{x^5 + 26\,x^4 + 66\,x^3 + 26\,x^2 + x}{(x-1)^6},\ \frac{x}{(x-1)^2},\ -\frac{x}{x-1},\ -\ln(1-x)$$

An unevaluated call is returned if the index is an integer $n > 1$ or a symbolic expression:

```
polylog(2, x), polylog(n^2 + 1, 2), polylog(n + 1, 2.0)
```

$$\text{polylog}(2, x),\ \text{polylog}(n^2 + 1, 2),\ \text{polylog}(n + 1, 2.0)$$

Floating point values are computed for integer indices n and floating-point arguments x:

```
polylog(-5, -1.2), polylog(10, 100.0 + 3.2*I)
```

$$-0.2326930882,\ 104.9131863 + 11.44600047\,i$$

Some special symbolic values are implemented:

```
polylog(4, 1), polylog(5, -1), polylog(2, I)
```

$$\frac{\pi^4}{90},\ -\frac{15\,\zeta(5)}{16},\ -\frac{\pi^2}{48} + \text{CATALAN}\,i$$

```
assume(n <> 1): polylog(n, -1)
```

$$\zeta(n)\left(2^{1-n}-1\right)$$

```
unassume(n): polylog(n, -1)
```

$$\mathrm{polylog}(n, -1)$$

Example 2

For indices $n \geq 1$, the real interval $[1, \infty)$ is a branch cut. The values returned by `polylog` jump when crossing this cut:

```
polylog(3, 1.2 + I/10^1000) - polylog(3, 1.2 - I/10^1000)
```

$$-1.379393155\ 10^{-18} + 0.1044301529\ \mathrm{i}$$

Example 3

The functions `diff`, `float`, `limit`, and `series` handle expressions involving `polylog`:

```
diff(polylog(n, x), x), float(polylog(4, 3 + I))
```

$$\frac{\mathrm{polylog}(n-1, x)}{x},\ 3.177636803 + 1.859135861\ \mathrm{i}$$

```
series(polylog(4, sin(x)), x = 0)
```

$$x + \frac{x^2}{16} - \frac{25\,x^3}{162} - \frac{13\,x^4}{768} + \frac{1523\,x^5}{405000} + \frac{49\,x^6}{51840} + O(x^7)$$

Parameters

n

An arithmetical expression representing an integer

x

An arithmetical expression

Return Values

Arithmetical expression.

Overloaded By

x

Algorithms

The polylogarithms are characterized by $\frac{d}{dx}\,\mathrm{Li}_n(x) = \frac{1}{x}\mathrm{Li}_{n-1}(x)$ in conjunction with $Li_n(0) = 0$ and $Li_1(x) = -\ln(1 - x)$. $Li_n(x)$ is a rational function in x for $n \le 0$.

Li_n has a branch cut along the real interval $[1, \infty)$ for indices $n \ge 1$. The value at a point x on the cut coincides with the limit "from below":

$$\mathrm{Li}_n(x) = \lim_{\varepsilon \to 0^+} \mathrm{Li}_n(x - \varepsilon\, i) = \left(\lim_{\varepsilon \to 0^+} \mathrm{Li}_n(x + \varepsilon\, i)\right) - \frac{2\,\pi\, i}{(n-1)!}\ln(x)^{n-1}.$$

References

L. Lewin, "Polylogarithms and Related Functions", North Holland (1981). L. Lewin (ed.), "Structural Properties of Polylogarithms", Mathematical Surveys and Monographs Vol. 37, American Mathematical Society, Providence (1991).

See Also

MuPAD Functions

`dilog` | `ln`

potential

The (scalar) potential of a gradient field

Syntax

```
potential(f, [x1, x2, …], <[y1, y2, …]>, <Test>)
```

Description

`potential(f, x)` determines whether the vector field $\vec{f} = \vec{f}(\vec{x})$ is a gradient field $\vec{f}(\vec{x}) = \mathrm{grad}(p(\vec{x}))$ of some scalar potential p with respect to the variables $\vec{x}$, and computes that potential if it exists.

The potential of a vector field $\vec{f}(\vec{x})$ = [$f_1(x_1, x_2, \ldots)$, $f_2(x_1, x_2, \ldots)$, …] exists (locally) if and only if the Jacobian matrix $\left(\frac{\partial}{\partial x_j} f_i\right)$ is symmetric in i and j. In 3 space, this is the condition that $\mathrm{curl}\left(\vec{f}(\vec{x})\right)$ vanishes.

The potential $p(\vec{x})$ with $\vec{f}((\vec{x})) = \mathrm{grad}(p(\vec{x}))$ is uniquely determined up to an additive constant.

An integral representation of the potential is given by

$$p(\vec{x}) = \int_0^1 (\vec{x} - \vec{y})\, \vec{f}(\vec{y} + \lambda\,(\vec{x} - \vec{y}))\, \mathrm{d}\lambda$$

,

where $\vec{y}$ is an arbitrary “base point.” This is the contour integral of $\vec{f}(\vec{x})$ along the straight line from $\vec{y}$ to $\vec{x}$.

If the Jacobian matrix $\left(\frac{\partial}{\partial x_j} f_i\right)$ is not symmetric, the potential of $\vec{f}(\vec{x})$ does not exist.

In this case, `potential` returns `FALSE`.

Note: Note that the answer `FALSE` is not always conclusive. For arbitrary expressions f_i, f_j, there is no algorithm to decide whether $\left(\frac{\partial}{\partial x_j} f_i\right) = \left(\frac{\partial}{\partial x_i} f_j\right)$ holds mathematically:

`potential` may return `FALSE` due to insufficient simplification of the partial derivatives.

The representation of the potential depends on the strength of the symbolic integrator `int`. If `int` does not manage to find a closed form of the potential, symbolic calls of `int` may be returned. See “Example 3” on page 1-1596.

If no base point $\vec{y}$ is specified, the potential is only defined up to some additive constant.

`potential` does not consider irregular points of the vector field and its potential and investigates the potential only locally. The returned potential may be a valid potential only in a neighbourhood of the current point $\vec{x}$!

If `f` is a vector, the component ring of `f` must be a field (i.e., a domain of category `Cat::Field`) which allows integration.

Examples

Example 1

Using the option `Test`, we check whether a vector field is a gradient field:

```
f := [x, y, z*exp(z)]:
potential(f, [x, y, z], Test)
```

```
TRUE
```

Without the option `Test`, the potential is returned:

```
potential(f, [x, y, z])
```

$$\frac{x^2}{2}+\frac{y^2}{2}+e^z\,(z-1)$$

We check the result:

```
normal(gradient(%, [x, y, z]))
```

$$\begin{pmatrix} x \\ y \\ z\,e^z \end{pmatrix}$$

When a `base point' is specified, a suitable constant is added to the potential such that it vanishes at this point:

```
potential(f, [x, y, z], [0, 0, 0])
```

$$\frac{x^2}{2}+\frac{y^2}{2}+e^z\,(z-1)+1$$

```
potential(f, [x, y, z], [x0, y0, z0])
```

$$\frac{x^2}{2}-\frac{x0^2}{2}+\frac{y^2}{2}-\frac{y0^2}{2}+e^z\,(z-1)-e^{z0}\,(z0-1)$$

```
delete f:
```

Example 2

The vector field in this example is not a gradient field and has no potential:

```
potential([x[2], -x[1]], [x[1], x[2]])
```

FALSE

Example 3

The vector field in this example is a gradient field and has a potential. However, the symbolic integrator does not find a closed form of the integral representation for the potential and returns a symbolic definite integral:

```
potential([a + b*x, sin(y^2)*exp(y)], [x, y])
```

$$\frac{b\,x^2}{2} + a\,x + \int_0^y \sin(\mathrm{X42}^2)\,e^{\mathrm{X42}}\,d\,\mathrm{X42}$$

We check the result:

```
gradient(%, [x, y])
```

$$\begin{pmatrix} a + b\,x \\ \sin(y^2)\,e^y \end{pmatrix}$$

Parameters

`f`

The vector field: a list of arithmetical expressions, or a vector of such expressions. A vector is an $n \times 1$ or $1 \times n$ matrix of a domain of category `Cat::Matrix`.

x_1, x_2, …

The variables: identifiers or indexed identifiers

y_1, y_2, …

The components of the "base point:" arithmetical expressions. If a base point $\vec{y}$ is specified, the returned potential p satisfies $p(\vec{y}) = 0$.

Options

`Test`

Check whether the vector field has a potential and return `TRUE` or `FALSE`, respectively.

Return Values

Arithmetical expression or a Boolean value.

See Also

MuPAD Functions

`curl` | `divergence` | `gradient` | `laplacian` | `vectorPotential`

powermod

Compute a modular power of a number or a polynomial

Syntax

```
powermod(b, e, m)
```

Description

`powermod(b, e, m)` computes b^e mod m.

If `b` and `m` are numbers, the modular power b^e mod m can also be computed by the direct call `b^e mod m`. However, `powermod(b, e, m)` avoids the overhead of computing the intermediate result b^e and computes the modular power much more efficiently.

If `b` is a rational number, then the modular inverse of the denominator is calculated and multiplied with the numerator.

If the modulus `m` is an integer, then the base `b` must either be a number, a polynomial expression or a polynomial that is convertible to an `IntMod(m)`-polynomial.

If the modulus `m` is a polynomial expression, then the base `b` must either be a number, a polynomial expression or a polynomial over the coefficient ring of MuPAD expressions.

If the modulus `m` is a polynomial of domain type `DOM_POLY`, then the base `b` must either be a number, or a polynomial of the same type as `m` or a polynomial expression that can be converted to a polynomial of the same type as `m`.

Note that the system function `mod` in charge of modular arithmetic may be changed by the user; see the help page of `mod`. The function `powermod` reacts accordingly. See “Example 5” on page 1-1601.

Internally, polynomials are divided by the function `divide`.

Examples

Example 1

We compute `3^(123456) mod 7`:

```
powermod(3, 123456, 7)
```

$$1$$

If the base is a rational number, the modular inverse of the denominator is computed and multiplied with the numerator:

```
powermod(3/5, 1234567, 7)
```

$$2$$

Example 2

The coefficients of the following polynomial expression are computed modulo 7:

```
powermod(x^2 + 7*x - 3, 10, 7)
```

$$x^{20} - 2\,x^{18} - x^{16} + x^{14} - 3\,x^6 - x^4 + 3\,x^2 - 3$$

Example 3

The power of the following polynomial expression is reduced modulo the polynomial $x^2 + 1$:

```
powermod(x^2 + 7*x - 3, 10, x^2 + 1)
```

$$1029668584\,x - 534842913$$

Example 4

The type of the return value coincides with the type of the base: a polynomial is returned if the base is a polynomial:

```
powermod(poly(x^2 + 7*x - 3), 2, x^2 + 1),
powermod(poly(x^2 + 7*x - 3), 2, poly(x^2 + 1))
```

$$\mathrm{poly}(-56\,x-33,\,[x]),\ \mathrm{poly}(-56\,x-33,\,[x])$$

If the base is a polynomial expression, `powermod` returns a polynomial expression:

```
powermod(x^2 + 7*x - 3, 2, x^2 + 1),
powermod(x^2 + 7*x - 3, 2, poly(x^2 + 1))
```

$$-56\,x-33,\ -56\,x-33$$

Example 5

The following re-definition of `_mod` switches to a symmetric representation of modular numbers:

```
R := Dom::IntegerMod(17):
_mod := mods: powermod(poly(2*x^2, R), 3, poly(3*x + 1, R))
```

$$\mathrm{poly}(-4,\,[x],\,\mathrm{Dom::IntegerMod}(17))$$

The following command restores the default representation:

```
_mod := modp: powermod(poly(2*x^2, R), 3, poly(3*x + 1, R))
```

$$\mathrm{poly}(13,\,[x],\,\mathrm{Dom::IntegerMod}(17))$$

```
unalias(R):
```

Parameters

b

The base: an integer, a rational number, or a polynomial of type `DOM_POLY`, or a polynomial expression

e

The power: a nonnegative integer

m

The modulus: an integer (at least 2), or a polynomial of type DOM_POLY, or a polynomial expression

Return Values

Depending on the type of b, the return value is an integer, a polynomial, or a polynomial expression. FAIL is returned if an expression cannot be converted to a polynomial.

Overloaded By

b

See Also

MuPAD Functions
divide | mod | modp | mods | poly

PRETTYPRINT

Control the formatting of output

Description

The environment variable PRETTYPRINT determines whether the MuPAD results are printed in the one-dimensional or the two-dimensional format.

Possible values: Either TRUE or FALSE

PRETTYPRINT controls the pretty printer, which is responsible for formatted output. If PRETTYPRINT has the value TRUE, then pretty printing is enabled for output.

The default value of PRETTYPRINT is TRUE; PRETTYPRINT has this value after starting or resetting the system via `reset`. Also the command `delete PRETTYPRINT` restores the default value.

In the MuPAD Notebook app, PRETTYPRINT normally has no effect when "typesetting" is activated. An exception occurs for MuPAD output without typesetting defined, where PRETTYPRINT determines the output style even if the typesetting is activated.

Typesetting is activated by default. You can switch it on or off by selecting **Notebook>Typeset Math** or via **View>Configure**.

Examples

Example 1

The following command disables pretty printing:

```
PRETTYPRINT := FALSE
```

```
FALSE
```

Now MuPAD results are printed in a one-dimensional, linearized form:

```
series(sin(x), x = 0, 14)
```

```
x - (1/6)*x^3 + (1/120)*x^5 - (1/5040)*x^7 + (1/362880)*x^9 - (1/39916800)\
*x^11 + (1/6227020800)*x^13 + O(x^15)
```

After setting PRETTYPRINT to TRUE again, the usual two-dimensional output format is used:

```
PRETTYPRINT := TRUE:
series(sin(x), x = 0, 14)
```

```
     3     5     7       9          11           13
    x     x     x       x          x            x           15
x - -- + --- - ---- + ------ - -------- + ---------- + O(x   )
     6   120   5040   362880   39916800   6227020800
```

See Also

MuPAD Functions

print | TEXTWIDTH

prevprime

Next smaller prime

Syntax

```
prevprime(a)
```

Description

`prevprime(a)` returns the greatest prime number less or equal than a. If $a < 2$, then `prevprime(a)` returns `FAIL`.

`prevprime` returns the function call with evaluated argument if the argument is not an integer.

`prevprime` returns an error if the argument evaluates to zero or a negative integer.

Examples

Example 1

Computing the largest prime $p \leq 15485865$:

```
prevprime(15485865)
```

```
    15485863
```

Example 2

There are no primes smaller than 2:

```
prevprime(1)
```

```
    FAIL
```

Parameters

a

A positive integer

Return Values

`prevprime(a)` returns either a natural number or `FAIL`.

Algorithms

`prevprime` uses the probabilistic prime test `isprime` and may therefore return false results with small probability.

See Also

MuPAD Functions
`isprime` | `ithprime` | `nextprime` | `numlib::proveprime`

print

Print objects to the screen

Syntax

```
print(<Unquoted>, <NoNL>, <KeepOrder>, <Plain>, <Typeset>, object1, object2, …)
```

Description

`print(object)` displays `object` on the screen.

At interactive level, the result of a MuPAD command entered at the command prompt is usually displayed on the screen automatically. `print` serves to generate additional output from within loops or procedures.

Apart from some exceptions mentioned below, the output generated by `print` is identical to the usual output of MuPAD results at interactive level.

`print` evaluates its arguments sequentially from left to right (cf. “Example 3” on page 1-1609) and displays the results on the screen. The individual outputs are separated by commas. A new line is started at the end of the output if this is not suppressed by the option `NoNL`.

The output width for `print` with option `Plain` is limited by the environment variable `TEXTWIDTH`. Cf. “Example 4” on page 1-1610.

With option `Plain` the style of the output is determined by the value of the environment variable `PRETTYPRINT`. Cf. “Example 5” on page 1-1611.

`print` descends recursively into the operands of an object. For each subobject `s`, `print` first determines its domain type`T`. If the domain `T` has a `"print"` slot, then `print` issues the call `T::print(s)` to the slot routine. In contrast to the overloading mechanism for most other MuPAD functions, `print` processes the result of this call recursively, and the result of the recursive process is printed at the position of `s` (cf. “Example 6” on page 1-1611).

Note: The result returned by the `"print"` method must not contain the domain element `s` itself as a subobject, since this leads to infinite recursion (cf. “Example 7”

on page 1-1612). The same remark also applies to the output procedures of function environments (see below).

If T is a library domain without a "print" slot and the internal operands of s are op1, op2, ..., then s is printed as new(T, op1, op2, ...). (See "Example 6" on page 1-1611.)

"print" methods may return strings or expressions. Strings are always printed unquoted. Expressions are printed in normal mode. If they contain strings, they will be printed with quotation marks. Cf. "Example 8" on page 1-1612.

The output of an expression is determined by the 0th operand of the expression. If the 0th operand is a function environment, then its second operand handles the output of the expression. See "Example 9" on page 1-1613. Otherwise, the expression is printed in functional notation.

In contrast to the usual output of MuPAD objects at interactive level, print does not perform resubstitution of aliases (see Pref::alias for details). Moreover, the routines defined via Pref::output and Pref::postOutput are not called by print. Cf. "Example 14" on page 1-1615.

The output of floating-point numbers depends on the environment variable DIGITS and the settings of Pref::floatFormat (exponential or floating-point representation) and Pref::trailingZeroes (printing of trailing zeroes). Cf. "Example 16" on page 1-1617.

Environment Interactions

print is sensitive to the environment variables DIGITS, PRETTYPRINT, and TEXTWIDTH, and to the output preferences Pref::floatFormat, Pref::keepOrder, and Pref::trailingZeroes.

Examples

Example 1

This example shows a simple call of print with strings as arguments. They are printed with quotation marks:

```
print("Hello", "You"." !"):
```

"Hello", "You !"

Example 2

On platforms supporting typesetting, `print` can generate typeset output:

```
print(Typeset, int(f(x)/g(x), x = a..b)):
```

$$\int_a^b \frac{f(x)}{g(x)}\,dx$$

`print` uses the `Typeset` option by default:

```
print(int(f(x)/g(x), x = a..b)):
```

$$\int_a^b \frac{f(x)}{g(x)}\,dx$$

ASCII output is available with the option `Plain`:

```
print(Plain, int(f(x)/g(x), x = a..b)):
```

```
  b
  /
 |   f(x)
 |   ---- dx
/    g(x)
  a
```

Example 3

Like most other functions, `print` evaluates its arguments. In the following call, `x` evaluates to `0` and `cos(0)` evaluates to `1`:

```
a := 0: print(cos(a)^2):
```

$$1$$

Use `hold` if you want to print the expression `cos(a)^2` literally:

```
print(hold(cos(a)^2)):
```

$$\cos(a)^2$$

```
delete a:
```

Example 4

`print` with the option `Plain` is sensitive to the current value of `TEXTWIDTH`:

```
print(Plain, expand((a + b)^4)):
```

```
 4      3        2  2        3    4
a  + 4 a  b + 6 a  b  + 4 a b  + b
```

```
TEXTWIDTH := 25:
print(Plain, expand((a + b)^4)):
```

```
 4      3        2  2
a  + 4 a  b + 6 a  b

          3    4
   + 4 a b  + b
```

If you disable the pretty print mode, the `print` function inserts the line continuation character at the line breaks:

```
PRETTYPRINT:=FALSE:
print(Plain, expand((a + b)^4)):
```

```
a^4 + 4*a^3*b + 6*a^2*b^\
2 + 4*a*b^3 + b^4
```

The line continuation character can be invalid for some strings. For example, when you use the code generators, such as `generate::MATLAB` and `generate::Simscape`, the

displayed code containing the line continuation character is not valid. To avoid inserting this character, change the TEXTWIDTH setting or use the fprint function instead of print:

```
fprint(Unquoted, 0, expand((a + b)^4))
```

```
a^4 + 4*a^3*b + 6*a^2*b^2 + 4*a*b^3 + b^4
```

Also, see the Example 4 on the fprint help page.

```
PRETTYPRINT := TRUE:
delete TEXTWIDTH:
```

Example 5

print with option Plain is sensitive to the current value of PRETTYPRINT:

```
print(Plain, a/b):
old := PRETTYPRINT: PRETTYPRINT := FALSE:
print(Plain, a/b):
PRETTYPRINT := old:
```

```
a
-
b
```

```
a/b
```

```
delete old:
```

Example 6

We demonstrate how to achieve formatted output for elements of a user-defined domain. Suppose that we want to write a new domain Complex for complex numbers. Each element of this domain has two operands: the real part r and the imaginary part s:

```
Complex := newDomain("Complex"): z := new(Complex, 1, 3):
z + 1;
print(Plain, z + 1):
```

$$\mathrm{new}(\mathrm{Complex}, 1, 3) + 1$$

```
new(Complex, 1, 3) + 1
```

Now we want a nicer output for elements of this domain, namely in the form `r+s*I`, where `I` denotes the imaginary unit. We implement the slot routine `Complex::print` to handle this. This slot routine will be called by MuPAD with an element of the domain `Complex` as argument whenever such an element is to be printed on the screen:

```
Complex::print := (z -> extop(z, 1) + extop(z, 2)*I):
z + 1;
print(Plain, z + 1):
```

$$1 + 3\,i + 1$$

```
1 + 3 I + 1
```

```
delete Complex, z:
```

Example 7

The result of a `"print"` method must not contain the argument as a subobject; otherwise this leads to infinite recursion. In the following example, the slot routine `T::print` would be called infinitely often. MuPAD tries to trap such infinite recursions and prints `` `????` `` instead:

```
T := newDomain(T): T::print := id:
new(T, 1);
print(Plain, new(T, 1)):
```

```
`????`
```

```
`????`
```

```
delete T:
```

Example 8

If a `"print"` method returns a string, it will be printed unquoted:

```
Example := newDomain("Example"):  e := new(Example, 1):
```

```
Example::print := x -> "elementOfExample":
print(e):
```

elementOfExample

If a `"print"`-method returns an expression, it will be printed in normal mode. If the expression contains strings, they will be printed in the usual way with quotation marks:

```
Example::print := x -> ["elementOfExample", extop(x)]:
print(e):
```

["elementOfExample", 1]

```
delete Example, e:
```

Example 9

Suppose that you have defined a function `f` that may return itself symbolically, and you want such symbolic expressions of the form `f(x,...)` to be printed in a special way. To this end, embed your procedure`f` in a function environment and supply an output procedure as second argument to the corresponding `funcenv` call. Whenever an expression of the form `f(x,...)` is to be printed, the output procedure will be called with the arguments `x,...` of the expression:

```
f := funcenv(f,
         proc(x) begin
            if nops(x) = 2 then
              "f does strange things with its arguments ".
              expr2text(op(x, 1))." and ".expr2text(op(x,2))
            else
              FAIL
            end
         end):

delete a, b:
f(a, b)/2;
f(a, b, c)/2
```

$$\frac{\text{f does strange things with its arguments a and b}}{2}$$

$$\frac{f(a, b, c)}{2}$$

```
delete f:
```

Example 10

With the option `Unquoted`, quotation marks are omitted:

```
print(Unquoted, "Hello", "World"." !"):
```

```
Hello, World !
```

With `Unquoted` the special characters `\t` and `\n` are expanded:

```
print(Unquoted, "As you can see,\n".
                "'\\n' is the newline character\n".
                "\tand '\\t' a tabulator"):
```

```
As you can see,
'\n' is the newline character
  and '\t' a tabulator
```

Example 11

It is useful to construct output strings using `expr2text` and the concatenation operator `.`:

```
d := 5:  print(Unquoted, "d plus 3 = ".expr2text(d + 3)):
```

```
d plus 3 = 8
```

```
delete d:
```

Example 12

With the option `NoNL`, no new line is put at the end of the output and `PRETTYPRINT` is implicitly set to `FALSE`. Apart from that, the behavior is the same as with the option `Unquoted`:

```
print(NoNL, "Hello"):  print(NoNL, ",  You"." !\n"):
print(NoNL, "As you can see, PRETTYPRINT is FALSE: "):
print(NoNL, x^2-1):  print(NoNL, "\n"):
```

```
Hello
```

```
,  You !
```

```
As you can see, PRETTYPRINT is FALSE:
```

```
x^2 - 1
```

Example 13

If the option `KeepOrder` is given, sums are printed in their internal order:

```
print(b - a):  print(KeepOrder, b - a):
```

$$b - a$$

$$-a + b$$

Example 14

Alias resubstitution (see `Pref::alias`) takes place for normal result outputs in an interactive session, but not for outputs generated by `print`:

```
delete a, b: alias(a = b):
a; print(a):
unalias(a):
```

$$a$$

$$b$$

In contrast to the usual result output, `print` does not react to `Pref::output`:

```
old := Pref::output(generate::TeX):
sin(a)^b; print(sin(a)^b):
Pref::output(old):
```

```
"{\sin\!\left(a\right)}^b"
```

$$\sin(a)^b$$

The same is true for `Pref::postOutput`:

```
old := Pref::postOutput("postOutput was called"):
a*b; print(a*b):
Pref::postOutput(old):
```

$$a\,b$$

```
postOutput was called
```

$$a\,b$$

```
delete old:
```

Example 15

The output of summands of a sum depends on the form of these summands. If the summand is a `_mult` expression, only the first and last operand of the product are taken into account for determining the sign of that term in the output. If one of them is a negative number then the "+"-symbol in the sum is replaced by a "-"-symbol:

```
print(hold(a + b*c*(-2)),
      hold(a + b*(-2)*c),
      hold(a + (-2)*b*c)):
```

$$a-2\,b\,c,\ a-b\,2\,c,\ a-2\,b\,c$$

This has to be taken into account when writing "print"-methods for polynomial domains.

Example 16

Floating point numbers are usually printed in fixed-point notation. You can change this to floating-point form with mantissa and exponent via Pref::floatFormat:

```
print(0.000001, 1000.0): old := Pref::floatFormat("e"):
print(0.000001, 1000.0): Pref::floatFormat(old):
```

```
0.000001, 1000.0
```

```
0.000001, 1000.0
```

In the default output of floating-point numbers, trailing zeroes are cut off. This behavior can be changed via Pref::trailingZeroes:

```
0.000001, 1000.0
```

```
0.000001, 1000.0
```

```
print(0.000001, 1000.0): old := Pref::trailingZeroes(TRUE):
print(0.000001, 1000.0): Pref::trailingZeroes(old):
```

```
0.000001, 1000.0
```

```
0.000001000000000, 1000.000000
```

The number of digits of floating-point numbers in output depends on the environment variable DIGITS:

```
print(float(PI)):
DIGITS := 20:  print(float(PI)):
DIGITS := 30:  print(float(PI)):
```

```
3.141592654
```

```
3.14159265358979323850
```

$$3.14159265358979323846264338328$$

```
delete old, DIGITS:
```

Example 17

The output order of `sets` differs from the internal order of sets, which is returned by `op`:

```
s := {a, b, 1}:
s;
print(Plain, s):
op(s)
```

$$\{1, a, b\}$$

```
{1, a, b}
```

$$a, b, 1$$

The index operator `[]` can be used to access the elements of a set with respect to the output order:

```
s[1], s[2], s[3]
```

$$1, a, b$$

```
delete s:
```

Example 18

The output of a domain is determined by its `"Name"` slot if it exists, and otherwise by its *key*:

```
T := newDomain("T"):
T;
print(Plain, T):
```

$$T$$

```
T
```

```
T::Name := "domain T":
T;
print(Plain, T):
```


```
domain T
```

```
delete T:
```

Example 19

It is sometimes desirable to combine strings with “pretty” expressions in an output. This is not possible via `expr2text`. On the other hand, an output with commas as separators is usually regarded as ugly. The following dummy expression sequence may be used to achieve the desired result. It uses the MuPAD internal function for standard operator output `builtin(1100,...)`, with priority `20`—the priority of `_exprseq`—and with an empty operator symbol `""`:

```
myexprseq := funcenv(myexprseq,
                     builtin(1100, 20, "", "myexprseq")):
print(Unquoted,
      myexprseq("String and pretty expression ", a^b, ".")):
```

```
                                b
String and pretty expression a .
```

```
delete myexprseq:
```

Example 20

If the option `Typeset` is combined with `Unquoted` or `NoNL`, a warning is given and `Typeset` is ignored:

```
print(Typeset, Unquoted, "1"):
```

```
Warning: Conflicting options, ignoring 'Typeset' [print]
```

Example 21

For more elaborate constructions, you may want to combine multi-line strings with MuPAD expressions. A first attempt might look like the following:

```
myexprseq := funcenv(myexprseq,
                     builtin(1100, 20, "", "myexprseq")):
Example := newDomain("Example"):
Example::print :=
  x -> myexprseq("--- \n--\n-\n--\n---", op(x)):
e := new(Example, 1):
print(Plain, e):
```

```
"--- \n--\n-\n--\n---"1
```

Obviously, this approach doesn't work. The return value of the `"print"` method defined above is not a string, it's a (special) sequence, so the special rules for printing a string do not apply. We would need another domain that simply takes a string and returns exactly this string from its `"print"` slot. Fortunately, MuPAD already has such a domain, `stdlib::Exposed`:

```
Example::print :=
  x -> myexprseq(stdlib::Exposed("--- \n--\n-\n--\n---"),
                 op(x)):
print(e):
```

```
---
--
-   1
--
---
```

For expressions with a higher output, you see that the alignment of the string is constant:

```
new(Example,x^(1/n));
new(Example,x/y)
```

```
---
--   1/n
-   x
```

```
--
---


---
--  x
-   -
--  y
---
```

To change this alignment, replace a `\n` by `\b`, thereby making the line it terminates the "baseline" of the string:

```
Example::print :=
  x -> myexprseq(stdlib::Exposed("--- \b--\n-\n--\n---"),
                 op(x)):
print(e+2):


--- 1 + 2
--
-
--
---
```

When multiple `\b` appear in a string, the first one is taken as defining the base line:

```
Example::print :=
  x -> myexprseq(stdlib::Exposed("--- \n--\n-\b--\b---"),
                 op(x)):
print(e+2):


---
--
-   1 + 2
--
---
```

Parameters

`object1, object2, …`

Any MuPAD objects

Options

Unquoted

With this option, character strings are displayed without quotation marks. Moreover, the control characters `\n`, `\t`, and `\\` in strings are expanded into a new line, a tabulator skip, and a single backslash `\`, respectively. Cf. “Example 10” on page 1-1614.

The control character `\t` is expanded with tab-size 8. The following character is placed in the next column `i` with `i mod 8 = 0`.

`\b` is expanded into a newline, too, but when combining multiple strings, the last line with `\b` at its end is regarded as the “baseline”. Cf. “Example 21” on page 1-1620.

Note: The option `Unquoted` implicitly sets the option `Plain`. If the option `Typeset` is used together with `Unquoted`, a warning is given and `Typeset` is ignored. Cf. “Example 20” on page 1-1619.

NoNL

This option has the same functionality as `Unquoted`. In addition, the new line at the end of the output is suppressed. Cf. “Example 12” on page 1-1614.

Moreover, this option implicitly sets `PRETTYPRINT` to `FALSE`.

Note: The option `NoNL` implicitly sets the option `Plain`. If the option `Typeset` is used together with `NoNL`, a warning is given and `Typeset` is ignored. Cf. “Example 20” on page 1-1619.

KeepOrder

This option determines the order of terms in sums. Normally, the system sorts the terms of a `sum` such that a positive term is in the first position of the output. If `KeepOrder` is given, no such re-ordering takes place and sums are printed in the internal order. Cf. “Example 13” on page 1-1615.

This behavior can also be controlled via `Pref::keepOrder`. More precisely, the call `print(KeepOrder, ...)` generates the same output as the following command:

```
Pref::keepOrder(Always):
print(...):
Pref::keepOrder(%2):
```

`Plain`

The output is in plain text mode. This is the default behavior in the terminal version. In text mode the value of `PRETTYPRINT` determines if the output is linear or in a more readable 2D form.

`Typeset`

The output is in typesetting mode. This is the default print behavior in the notebook, if no other options are given. The option is only kept for backward compatibility.

In typesetting mode the value of `PRETTYPRINT` is ignored.

Return Values

`print` returns the void object `null()` of type `DOM_NULL`.

Overloaded By

object1, object2

Algorithms

The output order of `sets` differs from the internal order of sets, which can be obtained via `op`. For this reordering in the output, the kernel calls the method `DOM_SET::sort`, which takes the set as argument and returns a sorted list. The elements of the set are then printed in the order given by this list.

See Also

MuPAD Domains
`DOM_FUNC_ENV`

MuPAD Functions

DIGITS | doprint | expose | expr2text | finput | fprint | fread | funcenv | input | Pref::floatFormat | Pref::keepOrder | Pref::trailingZeroes | PRETTYPRINT | protocol | read | strprint | TEXTWIDTH | write

->, -->, proc, name, option, local, begin, end_proc, procname

Define a procedure

Syntax

```
( x1, x2, … ) -> body

proc(
     x1 <= default1> <: type1>,
     x2 <= default2> <: type2>,
     ...
    ) <: returntype>
  <name pname;>
  <option option1, option2, …>
  <local local1, local2, …>
  <save global1, global2, …>
begin
   body
end_proc

( x1, x2, … ) --> body

_procdef(, …)
```

Description

`proc - end_proc` defines a procedure.

Procedures `f := proc(x1, x2, ...) ... end_proc` may be called like a system function in the form `f(x1, x2, ...)`. The return value of this call is the value of the last command executed in the procedure body (or the value returned by the body via the function `return`).

The procedure declaration `(x1, x2, ...) -> body` is equivalent to `proc(x1, x2, ...) begin body end_proc`. It is useful for defining *simple* procedures that

do not need local variables. E.g., `f := x -> x^2` defines the mathematical function $f: x \to x^2$. If the procedure uses more than one parameter, use brackets as in `f := (x, y) -> x^2 + y^2`. Cf. “Example 1” on page 1-1629.

The procedure declaration `(x1, x2, ...) --> body` is equivalent to `fp::unapply(body, x1, x2, ...)`. The difference from the other definitions is that `body` is *evaluated* before defining the procedure. Cf. “Example 2” on page 1-1629.

Note: The evaluation of `body` must not contain references to parameters or local variables of an outer procedure.

A MuPAD procedure may have an arbitrary number of parameters. For each parameter, a default value may be specified. This value is used if no actual value is passed when the procedure is called. E.g.,

```
f := proc(x = 42) begin body end_proc
```

defines the default value of the parameter `x` to be `42`. The call `f()` is equivalent to `f(42)`. Cf. “Example 3” on page 1-1630.

For each parameter, a type may be specified. This invokes an automatic type checking when the procedure is called. E.g.,

```
f := proc(x : DOM_INT) begin body end_proc
```

restricts the argument `x` to integer values. If the procedure is called with an argument of a wrong data type, the evaluation is aborted with an error message. Cf. “Example 4” on page 1-1630. Checking the input parameters should be a standard feature of every procedure. See Testing Arguments.

Also an automatic type checking for the return value may be implemented specifying `returntype`. Cf. “Example 4” on page 1-1630.

With the keyword `name`, a name may be defined for the procedure, e.g.,

```
f := proc(...) name myName; begin body end_proc.
```

There is a special variable `procname` associated with a procedure which stores its name. When the body returns a symbolic call `procname(args())`, the actual name is

substituted. This is the name defined by the optional `name` entry. If no `name` entry is specified, the first identifier the procedure has been assigned to is used as the name, i.e., `f` in this case. Cf. “Example 5” on page 1-1631.

With the keyword `option`, special features may be specified for a procedure:

- `escape`

  Must be used if the procedure creates and returns a new procedure which accesses local values of the enclosing procedure. Cf. “Example 6” on page 1-1632. This option should only be used if necessary. Also refer to `Pref::warnDeadProcEnv`.

- `hold`

  Prevents the procedure from evaluating the actual parameters it is called with. See “Example 7” on page 1-1633.

- `noDebug`

  Prevents the MuPAD source code debugger from entering this procedure. Also refer to `Pref::ignoreNoDebug`. Cf. “Example 8” on page 1-1634.

- `noFlatten`

  Prevents flattening of sequences passed as arguments of the procedure. See “Example 9” on page 1-1634.

- `remember`

  Instructs the procedure to store each computed result in a so-called remember table. When this procedure is called later with the same input parameters, the result is read from this table and needs not be computed again.

  This may speed up, e.g., recursive procedures drastically. Cf. “Example 10” on page 1-1635. However, the remember table may grow large and use a lot of memory. Furthermore, the usefulness of this function is very limited in the light of properties—identification of “the same input parameters” does not depend on assumptions on identifiers or global variables such as `DIGITS` and `ORDER`, so the returned result may not be compatible with new assumptions. Use of `prog::remember` instead of this option is highly recommended for any function accepting symbolic input.

- `noExpose`

  Instructs MuPAD to hide the procedure body from the user. Note that this prevents debugging the procedure, too. Cf. “Example 15” on page 1-1640.

With the keyword `local`, the local variables of the procedure are specified, e.g.,

```
f := proc(...) local x, y; begin body end_proc.
```

Cf. “Example 11” on page 1-1637.

Local variables cannot be used as “symbolic variables” (identifiers). They must be assigned values before they can be used in computations.

Note that the names of global MuPAD variables such `DIGITS`, `READPATH` etc. should not be used as local variables. Also refer to the keyword `save`.

With the keyword `save`, a local context for global MuPAD variables is created, e.g.,

```
f := proc(...) save DIGITS; begin DIGITS := newValue; ... end_proc.
```

This means that the values these variables have on entering the procedure are restored on exiting the procedure. This is true even if the procedure is exited because of an error. Cf. “Example 12” on page 1-1637.

One can define procedures that accept a variable number of arguments. E.g., one may declare the procedure without any formal parameters. Inside the body, the actual parameters the procedure is called with may be accessed via the function `args`. Cf. “Example 13” on page 1-1639.

Calling a procedure name `f`, say, usually does not print the source code of the body to the screen. Use `expose(f)` to see the body. Cf. “Example 14” on page 1-1639.

The environment variable `MAXDEPTH` limits the “nesting depth” of recursive procedure calls. The default value is `MAXDEPTH` = 500. Cf. “Example 10” on page 1-1635.

If a procedure is a domain slot, the special variable `dom` contains the name of the domain the slot belongs to. If the procedure is not a domain slot, the value of `dom` is `NIL`.

Instead of `end_proc`, also the keyword `end` can be used.

The imperative declaration `proc - end_proc` internally results in a call of the kernel function `_procdef`. There is no need to call `_procdef` directly.

When evaluating a procedure, MuPAD parses the entire procedure first, and only then executes it. If you want to introduce a new syntax (for example, define a new `operator`), do it outside a procedure. See “Example 16” on page 1-1641.

Examples

Example 1

Simple procedures can be generated with the "arrow operator" `->`:

```
f := x -> x^2 + 2*x + 1:
f(x), f(y), f(a + b), f(1.5)
```

$$x^2+2\,x+1,\ y^2+2\,y+1,\ 2\,a+2\,b+(a+b)^2+1,\ 6.25$$

```
f := n -> isprime(n) and isprime(n + 2):
f(i) $ i = 11..18
```

TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE, FALSE

The following command maps an "anonymous" procedure to the elements of a list:

```
map([1, 2, 3, 4, 5, 6], x -> x^2)
```

[1, 4, 9, 16, 25, 36]

```
delete f:
```

Example 2

The declaration of procedures with the "arrow operator" is a powerful tool. In some situations, however, it results in potentially unexpected results:

```
f := x -> sin(x^2)
```

$$x \to \sin(x^2)$$

```
g := x -> f'(x)
```

$$x \to f'(x)$$

The reason is simple: The body of a procedure definition is not evaluated at the time of definition. For those occasions where evaluation is desired, the long version of the arrow operator should be used:

```
g := x --> f'(x)
```

$$x \to 2\,x\,\cos(x^2)$$

Of course, in this example, there is an even shorter way:

```
g := f'
```

$$x \to 2\,x\,\cos(x^2)$$

Example 3

The declaration of default values is demonstrated. The following procedure uses the default values if the procedure call does not provide all parameter values:

```
f := proc(x, y = 1, z = 2) begin [x, y, z] end_proc:
f(x, y, z), f(x, y), f(x)
```

$$[x, y, z],\ [x, y, 2],\ [x, 1, 2]$$

No default value was declared for the first argument. A warning is issued if this argument is missing:

```
f()
```

```
Warning: Uninitialized variable 'x' is used.
  Evaluating: f
```

$$[\mathrm{NIL}, 1, 2]$$

```
delete f:
```

Example 4

The automatic type checking of procedure arguments and return values is demonstrated. The following procedure accepts only positive integers as argument:

```
f := proc(n : Type::PosInt) begin n! end_proc:
```

An error is raised if an unsuitable parameter is passed:

```
f(-1)
```

```
Error: The object '-1' is incorrect. The type of argument number 1 must be 'Type::PosIn
  Evaluating: f

Error: Wrong type of 1. argument (type 'Type::PosInt'
expected,        got argument '-1');   Evaluating: f
```

In the following procedure, automatic type checking of the return value is invoked:

```
f := proc(n : Type::PosInt) : Type::Integer
begin
  n/2
end_proc:
```

An error is raised if the return value is not an integer:

```
f(3)
```

```
Error: The return value '3/2' is incorrect. The type of the return value must be 'Type
  Evaluating: f

Error: Wrong type of return value (type 'Type::Integer'
expected,        value is '3/2');   Evaluating: f
```

```
delete f:
```

Example 5

The `name` entry of procedures is demonstrated. A procedure returns a symbolic call to itself by using the variable `procname` that contains the current procedure name:

```
f := proc(x)
begin
  if testtype(x,Type::Numeric)
    then return(float(1/x))
    else return(procname(args()))
  end_if
end_proc:
```

```
f(x), f(x + 1), f(3), f(2*I)
```

$$f(x),\ f(x+1),\ 0.3333333333,\ -0.5\,i$$

Also error messages use this name:

```
f(0)
```

```
Error: Division by zero. [_invert]
  Evaluating: f
```

If the procedure has a name entry, this entry is used:

```
f := proc(x)
name myName;
begin
  if testtype(x,Type::Numeric)
    then return(float(1/x))
    else return(procname(args()))
  end_if
end_proc:
f(x), f(x + 1), f(3), f(2*I)
```

$$\mathrm{myName}(x),\ \mathrm{myName}(x+1),\ 0.3333333333,\ -0.5\,i$$

```
f(0)
```

```
Error: Division by zero. [_invert]
  Evaluating: myName
```

```
delete f:
```

Example 6

The option `escape` is demonstrated. This option must be used if the procedure returns another procedure that references a formal parameter or a local variable of the generating procedure:

```
f := proc(n)
begin
```

```
  proc(x) begin x^n end_proc
end_proc:
```

Without the option `escape`, the formal parameter `n` of `f` leaves its scope: `g := f(3)` references `n` internally. When `g` is called, it cannot evaluate `n` to the value `3` that `n` had inside the scope of the function `f`:

```
g := f(3): g(x)
```

```
Warning: Uninitialized variable 'unknown' is used.
  Evaluating: g
```

```
Error: The operand is invalid. [_power]
  Evaluating: g
```

The option `escape` instructs the procedure `f` to deal with variables escaping the local scope. Now, the procedure `g := f(3)` references the value `3` rather than the formal parameter `n` of `f`, and `g` can be executed correctly:

```
f := proc(n)
option escape;
begin
  proc(x) begin x^n end_proc
end_proc:
g := f(3): g(x), g(y), g(10)
```

$$x^3,\ y^3,\ 1000$$

```
delete f, g:
```

Example 7

The option `hold` is demonstrated. With `hold`, the procedure sees the actual parameter in the form that was used in the procedure call. Without `hold`, the function only sees the value of the parameter:

```
f := proc(x) option hold; begin x end_proc:
g := proc(x) begin x end_proc:
x := PI/2:
f(sin(x) + 2) = g(sin(x) + 2), f(1/2 + 1/3) = g(1/2 + 1/3)
```

$$\sin(x) + 2 = 3,\ \frac{1}{2} + \frac{1}{3} = \frac{5}{6}$$

Procedures using `option hold` can evaluate the arguments with the function `context`:

```
f := proc(x) option hold; begin x = context(x) end_proc:
f(sin(x) + 2), f(1/2 + 1/3)
```

$$\sin(x) + 2 = 3,\ \frac{1}{2} + \frac{1}{3} = \frac{5}{6}$$

```
delete f, g, x:
```

Example 8

The option `noDebug` is demonstrated. The `debug` command starts the debugger which steps inside the procedure `f`. After entering the debugger command `c` (continue), the debugger continues the evaluation:

```
f := proc(x) begin x end_proc: debug(f(42))
```

```
Activating debugger...   #0 in f($1=42) at
/tmp/debug0.556:4 mdx> c Execution completed.
               42
```

With the option `noDebug`, the debugger does not step into the procedure:

```
f := proc(x) option noDebug; begin x end_proc: debug(f(42))
```

```
Execution completed.
          42
```

```
delete f:
```

Example 9

Create a procedure that accepts two arguments and returns a table containing the arguments:

```
f := proc(x, y) begin table(x = y) end_proc:
```

The parameters `x, y` of the procedure `f` form a sequence. If you call this procedure for the sequence `(a, b)` and a variable `c`, MuPAD flattens the nested sequence `((a, b),`

`c)` into `(a, b, c)`. The procedure `f` accepts only two arguments. Thus, it uses `a` and `b`, and ignores `c`:

```
f((a, b), c)
```

$$\overline{a \mid b}$$

When you use the `noFlatten` option, MuPAD does not flatten the arguments of the procedure:

```
g := proc(x, y) option noFlatten; begin table(x = y) end_proc:
g((a, b), c)
```

$$\overline{a, b \mid c}$$

For further computations, delete `f` and `g`:

```
delete f, g:
```

Example 10

The option `remember` is demonstrated. The `print` command inside the following procedure indicates if the procedure body is executed:

```
f:= proc(n : Type::PosInt)
option remember;
begin
   print("computing ".expr2text(n)."!");
   n!
end_proc:
f(5), f(10)
```

"computing 5!"

"computing 10!"

120, 3628800

When calling the procedure again, all values that were computed before are taken from the internal “remember table” without executing the procedure body again:

```
f(5)*f(10) + f(15)
```

```
"computing 15!"
```

```
1308109824000
```

`option remember` is used in the following procedure which computes the Fibonacci numbers $F(0) = 0$, $F(1) = 1$, $F(n) = F(n - 1) + F(n - 2)$ recursively:

```
f := proc(n : Type::NonNegInt)
option remember;
begin
   if n = 0 or n = 1 then return(n) end_if;
   f(n - 1) + f(n - 2)
end_proc:

f(123)
```

```
22698374052006863956975682
```

Due to the recursive nature of `f`, the arguments are restricted by the maximal recursive depth (see `MAXDEPTH`):

```
f(1000)
```

```
Error: Recursive definition: the maximal depth for nested procedure calls is reached.
  Evaluating: f
```

Without `optionremember`, the recursion is rather slow:

```
f := proc(n : Type::NonNegInt)
begin
   if n = 0 or n = 1 then return(n) end_if;
   f(n - 1) + f(n - 2)
end_proc:

f(28)
```

```
     317811

delete f:
```

Example 11

We demonstrate the use of local variables:

```
f := proc(a)
local x, y;
begin
  x := a^2;
  y := a^3;
  print("x, y" = (x, y));
  x + y
end_proc:
```

The local variables `x` and `y` do not coincide with the global variables `x`, `y` outside the procedure. The call to `f` does not change the global values:

```
x := 0: y := 0: f(123), x, y
```

```
     "x, y" = (15129, 1860867)

     1875996, 0, 0

delete f, x, y:
```

Example 12

The `save` declaration is demonstrated. The following procedure changes the environment variable `DIGITS` internally. Because of `save DIGITS`, the original value of `DIGITS` is restored after return from the procedure:

```
myfloat := proc(x, digits)
save DIGITS;
begin
  DIGITS := digits;
  float(x);
```

```
end_proc:
```

The current value of DIGITS is:

```
DIGITS
```

$$10$$

With the default setting DIGITS = 10, the following float conversion suffers from numerical cancellation. Due to the higher internal precision, myfloat produces a more accurate result:

```
x := 10^20*(PI - 21053343141/6701487259):
float(x), myfloat(x, 20)
```

$$0.0,\ 0.02616405487$$

The value of DIGITS was not changed by the call to myfloat:

```
DIGITS
```

$$10$$

The following procedure needs a global identifier, because local variables cannot be used as integration variables in the int function. Internally, the global identifier x is deleted to make sure that x does not have a value:

```
f := proc(n)
save x;
begin
  delete x;
  int(x^n*exp(-x), x = 0..1)
end_proc:
```

```
x := 3: f(1), f(2), f(3)
```

$$1 - 2\,e^{-1},\ 2 - 5\,e^{-1},\ 6 - 16\,e^{-1}$$

Because of save x, the previously assigned value of x is restored after the integration:

```
x
```

```
    3
```

```
delete myfloat, x, f:
```

Example 13

The following procedure accepts an arbitrary number of arguments. It accesses the actual parameters via `args`, puts them into a list, reverses the list via `revert`, and returns its arguments in reverse order:

```
f := proc()
local arguments;
begin
   arguments := [args()];
   op(revert(arguments))
end_proc:
```

```
f(a, b, c)
```

```
    c, b, a
```

```
f(1, 2, 3, 4, 5, 6, 7)
```

```
    7, 6, 5, 4, 3, 2, 1
```

```
delete f:
```

Example 14

Use `expose` to see the source code of a procedure:

```
f := proc(x = 0, n : DOM_INT)
begin
  sourceCode;
end_proc
```

```
    proc f(x, n) ... end
```

```
expose(f)
```

```
proc(x = 0, n : DOM_INT)
  name f;
begin
  sourceCode
end_proc

delete f:
```

Example 15

The option `noExpose` prevents users from reading the definition of a procedure:

```
f := proc(a)
  option noExpose;
begin
  print(sin(a));
  if is(a>1)=TRUE then
    cos(a)
  else
    cos(a + 2)
  end_if
end_proc
```

proc f(a) ... end

```
f(x), f(0), f(3)
```

$\sin(x)$

0

$\sin(3)$

$\cos(x+2), \cos(2), \cos(3)$

```
expose(f)
```

```
proc(a)
  name f;
  option noDebug, noExpose;
begin
  /* Hidden */
end_proc
```

As you can see, setting option `noExpose` implicitly sets the option `noDebug`, too.

For more information on the intended use of this option, refer to the documentation of `write`.

Example 16

When you evaluate a procedure, MuPAD parses the entire procedure, and only then executes it. Thus, you cannot define and use a new operator inside a procedure. For example, when MuPAD parses this procedure, it does not recognize the new operator <<. The reason is that the procedure is not executed yet, and therefore, the new operator is not defined:

```
f := proc(A, B)
begin
  bitshiftleft := (a, b) -> a * 2^b:
  operator("<<", bitshiftleft, Binary, 950):

  C := A<<B;

end_proc:


Error: Invalid input. 'expression' is expected. [line 6, col 10]
```

Define the operator << on the interactive level:

```
bitshiftleft := (a, b) -> a * 2^b:
operator("<<", bitshiftleft, Binary, 950):
```

Now you can use << inside procedures on the interactive level:

```
f := proc(A, B)
begin
  C := A<<B;
end_proc:
f(2, 1)
```

```
     4
```

```
m<<n
```

$$2^n\, m$$

Parameters

x_1, x_2, …

The formal parameters of the procedure: identifiers

$default_1$, $default_2$, …

Default values for the parameters: arbitrary MuPAD objects

$type_1$, $type_2$, …

Admissible types for the parameters: type objects as accepted by the function `testtype`

returntype

Admissible type for the return value: a type object as accepted by the function `testtype`

pname

The name of the procedure: an expression

$option_1$, $option_2$, …

Available options are: `escape`, `hold`, `noDebug`, `noExpose`, `noFlatten`, `remember`

$local_1$, $local_2$, …

The local variables: identifiers

global1, global2, …

Global variables: identifiers

`body`

The body of the procedure: an arbitrary sequence of statements

Return Values

Procedure of type `DOM_PROC`.

See Also

MuPAD Functions

`args` | `context` | `debug` | `expose` | `fp::unapply` | `hold` | `MAXDEPTH` | `newDomain` | `Pref::ignoreNoDebug` | `Pref::typeCheck` | `Pref::warnDeadProcEnv` | `return` | `save` | `testargs`

product

Definite and indefinite products

Syntax

```
product(f, i)
product(f, i = a .. b)
product(f, i = RootOf(p, <x>))
product(f, i in RootOf(p, <x>))
product(f, i in {x1, x2, …})
```

Description

`product(f, i)` computes the indefinite product of $f(i)$ with respect to i, i.e., a closed form g such that $\frac{g(i+1)}{g(i)} = f(i)$.

`product(f, i = a..b)` tries to find a closed form representation of the product $\prod_{i=a}^{b} f(i)$.

`product (f, i = RootOf(p, x))` computes the product of $f(i)$ over the roots of the polynomial p.

`product(f, i in { x1, x2, …})` computes the product $\prod_{i \in \{x_1, x_2, \ldots\}} f(i)$.

`product` serves for simplifying *symbolic* products. It should *not* be used for multiplying a finite number of terms: if `a` and `b` are integers of type `DOM_INT`, the call `_mult(f $ i = a..b)` is more efficient than `product(f, i = a..b)`.

`product(f, i)` computes the indefinite product of `f` with respect to `i`. This is an expression `g` such that $f(i) = \frac{g(i+1)}{g(i)}$.

It is implicitly assumed that `i` runs through integers only.

`product(f, i = a..b)` computes the definite product with `i` running from `a` to `b`. It is implicitly assumed that $a \le b$; it is an error if this is inconsistent.

`a` and `b` must not be numbers other than integers.

If `b-a` is an integer, the explicit product $f(a)\, f(a+1) \ldots f(b)$ is returned if it has no more than 1000 factors.

`product (f, i = RootOf(p, x))` computes the definite product with `i` running through the roots of the polynomial `p` in `x` according to their multiplicity, i.e., the number of factors is equal to the degree of `p`.

The calls `product (f, i = RootOf(p, x))` and `product (f, i in RootOf(p, x))` are equivalent.

The system returns a symbolic `product` call if it cannot compute a closed form representation of the product.

Examples

Example 1

Each of the following two calls computes the product 1 2 3 4 5:

```
product(i, i = 1..5) = _mult(i $ i = 1..5)
```

$$120 = 120$$

However, using `_mult` is usually more efficient when the boundaries are integers of type `DOM_INT`.

There is a closed form of this definite product from 1 to n:

```
product(i, i = 1..n)
```

$$n!$$

Since the upper boundary is a symbolic identifier, `_mult` cannot handle this product:

```
_mult(i $ i = 1..n)
```

$$\_\mathrm{mult}(i\ \$\ i = 1..n)$$

The corresponding indefinite product is:

```
product(i, i)
```

$$\begin{cases} \Gamma(i) & \text{if } 1 \le i \\ \frac{(-1)^i}{\Gamma(1-i)} & \text{if } i \le -1 \end{cases}$$

The indefinite and the definite product of $2\,i + 1$ are:

```
product(2*i + 1, i)
```

$$2^i\,\Gamma\left(i + \frac{1}{2}\right)$$

```
product(2*i + 1, i = 1..n)
```

$$\frac{\frac{1}{2^{n+1}}\,(2\,n+2)!}{(n+1)!}$$

The boundaries may be symbolic expressions or $\pm\infty$ as well:

```
product(i^2/(i^2 - 1), i = 2..infinity)
```

$$2$$

The system cannot find closed forms of the following two products and returns symbolic `product` calls:

```
delete f: product(f(i), i)
```

$$\prod_i f(i)$$

```
product((1 + 2^(-i)), i = 1..infinity)
```

$$\prod_{i=1}^{\infty} \frac{2^i+1}{2^i}$$

An approximation can be computed numerically via `float`:

```
float(%)
```

$$2.384231029$$

Alternatively, you can call `numeric::product` directly. This is usually more efficient, since it skips the symbolic computations performed by `product`:

```
numeric::product((1 + 2^(-i)), i = 1..infinity)
```

$$2.384231029$$

Example 2

Some products over the roots of a polynomial:

```
product(1 + 1/x, x = RootOf(x^2 - 5*x + 6))
```

$$2$$

```
product(r+c, r = RootOf(x^3 + a*x^2 + b*x + c, x))
```

$$b\,c - c - a\,c^2 + c^3$$

The multiplicity of roots is taken into account:

```
product(x+2, x in RootOf(x^5))
```

$$32$$

MuPAD finds closed forms for products of rational expressions. In other cases, a symbolic call to `product` is returned:

```
product(sin(r), r = RootOf(x^2 - PI^2/4, x))
```

$$\prod_{r=\mathrm{RootOf}\left(x^2-\frac{\pi^2}{4},x\right)} \sin(r)$$

An approximation can be computed numerically via `float`:

```
float(%)
```

$$-1.0$$

Example 3

Some products over elements of a set:

```
product(x+2, x in {2,4,8})
```

$$240$$

```
product(a*x, x in {3, b, 5})
```

$$15\,a^3\,b$$

Identical objects appear only once in a set. Therefore, the second `a` in the following example has no effect on the result:

```
product(-x, x in {a,a,7,b})
```

$$-7\,a\,b$$

Parameters

`f`

An arithmetical expression depending on `i`

`i`

The product index: an identifier or indexed indentifier

`a`, `b`

The boundaries: arithmetical expressions

`p`

A polynomial expression in `x`

`x`

Indeterminate

Return Values

arithmetical expression.

Algorithms

The product over the roots of a polynomial is computed via `polylib::resultant`.

See Also

MuPAD Functions

`*` | `_mult` | `numeric::product` | `sum`

protect

Protect an identifier

Syntax

```
protect(x, <ProtectLevelError | ProtectLevelWarning | ProtectLevelNone>)
```

Description

`protect(x)` protects the identifier `x`.

`protect(x, ProtectLevelError)` sets full write-protection for the identifier. Any subsequent attempt to assign a value to the identifier will lead to an error.

`protect(x, ProtectLevelWarning)` sets a "soft" protection. Any subsequent assignment to the identifier results in a warning message. However, the identifier will be assigned a value, anyway.

`protect(x)` is equivalent to `protect(x, ProtectLevelWarning)`.

`protect(x, ProtectLevelNone)` removes any protection from the identifier. This call is equivalent to `unprotect(x)`.

Note: Overwriting protected identifiers such as the names of MuPAD functions may damage your current session.

Examples

Example 1

The following call protects the identifier `important` with the protection level "`ProtectLevelWarning`":

```
protect(important, ProtectLevelWarning)
```

ProtectLevelNone

The identifier can still be overwritten:

```
important := 1
```

```
Warning: The protected variable 'important' is overwritten. [_assign]
```

1

We protect the identifier with the level "`ProtectLevelError`":

```
protect(important, ProtectLevelError)
```

ProtectLevelWarning

Now, it is no longer possible to overwrite `important`:

```
important := 2
```

```
Error: The identifier 'important' is protected. [_assign]
```

The identifier keeps its previous value:

```
important
```

1

In order to overwrite this value, we must unprotect `important`:

```
protect(important, ProtectLevelNone)
```

ProtectLevelError

```
important := 2
```

2

The identifier is protected again with the default level "`ProtectLevelWarning`":

```
protect(important)
```

```
     ProtectLevelNone
```

```
important := 1
```

```
Warning: The protected variable 'important' is overwritten. [_assign]
```

```
     1
```

```
unprotect(important): delete important:
```

Example 2

`protect` does not evaluate its first argument. Here the identifier `x` can still be overwritten, while its value – which is the identifier `y` – remains write protected:

```
protect(y, ProtectLevelError):  x := y:  protect(x):  x := 1
```

```
Warning: The protected variable 'x' is overwritten. [_assign]
```

```
     1
```

```
y := 2
```

```
Error: The identifier 'y' is protected. [_assign]
```

```
unprotect(x):  unprotect(y):  delete x, y:
```

Parameters

x

An identifier

Options

`ProtectLevelError`, `ProtectLevelNone`, `ProtectLevelWarning`

The level of protection to set. The default value is `ProtectLevelWarning`.

Return Values

Previous protection level of `x`: either `ProtectLevelError` or `ProtectLevelWarning` or `ProtectLevelNone`.

Algorithms

`protect` does not evaluate its first argument. This way identifiers can be protected that have been assigned a value.

Identifiers starting with a # are implicitly protected and cannot be assigned a value nor receive assumptions.

See Also

MuPAD Functions

`:=` | `unprotect`

protocol

Create a protocol of a session

Syntax

```
protocol(filename | n, <InputOnly | OutputOnly>)
```

```
protocol()
```

Description

`protocol(file)` starts a protocol of the current MuPAD terminal session.

`protocol()` stops the protocol.

`protocol` writes a protocol of input commands and corresponding MuPAD output of a terminal session to a text file.

When used from the MuPAD Notebook app, `protocol` is disabled and raises an error.

The file may be specified directly by its name. This either creates a new file or overwrites an existing file. `protocol` opens and closes the file automatically.

If the filename ends in "`.gz`", MuPAD will write a `gzip`-compressed text file.

If `WRITEPATH` does not have a value, `protocol` interprets the file name as a pathname relative to the "working directory."

Note that the meaning of "working directory" depends on the operating system. On Microsoft Windows systems and on Mac OS X systems, the "working directory" is the folder where MuPAD is installed. On UNIX systems, it is the current working directory in which MuPAD was started; when started from a menu or desktop item, this is typically the user's home directory.

Also absolute path names are processed by `protocol`.

Alternatively, the file may be specified by a file descriptor `n`. In this case, the file must have been opened via `fopen(Text, filename, Write)` or `fopen(Text, filename,`

`Append)`. This returns the file descriptor as an integer `n`. Note that `fopen(filename)` opens the file in read-only mode. A subsequent `protocol` command to this file causes an error.

The file is not closed automatically by `protocol()` and must be closed by a subsequent call to `fclose`.

A call of `protocol` without arguments terminates a running protocol and closes the corresponding file if it has been opened by `protocol`. Closing the protocol file with `fclose` also terminates the protocol.

If a new protocol is started while a protocol is running, then the old one is terminated and the corresponding file is closed.

Environment Interactions

The function is sensitive to the environment variable `WRITEPATH`. If this variable has a value, then the protocol file is created in the corresponding directory. Otherwise, the file is created in the "current working directory."

Examples

Example 1

We open a text file `test` in write mode with `fopen`:

```
n := fopen(Text, "test", Write):
```

A protocol is written into this file:

```
protocol(n):
1 + 1, a/b;
solve(x^2 = 2);
protocol():
fclose(n):
```

The file now has the following content:

```
1 + 1, a/b;

     a
  2, -
     b
solve(x^2 = 2);

          1/2            1/2
  {[x = 2   ], [x = - 2   ]}
protocol():
```

Example 2

The protocol file is opened directly by `protocol`. Only input is protocolled:

```
protocol("test", InputOnly):
1 + 1, a/b;
solve(x^2 = 2);
protocol():
```

The file now has the following content:

```
1 + 1, a/b;
solve(x^2 = 2);
protocol():
```

Example 3

The protocol file is opened directly by `protocol`. Only output is protocolled:

```
protocol("test", OutputOnly):
1 + 1, a/b;
solve(x^2 = 2);
protocol():
```

The file now has the following content:

```
     a
  2, -
     b
```

```
       1/2           1/2
{[x = 2   ], [x = - 2   ]}
```

Parameters

`filename`

The name of a file: a character string

`n`

A file descriptor provided by `fopen`: a positive integer

Options

`InputOnly`

Only input is protocolled

The protocol file only contains the input lines. All output is omitted.

`OutputOnly`

Only output is protocolled

The protocol file only contains the output lines. All input is omitted.

Return Values

Void object of type `DOM_NULL`.

See Also

MuPAD Functions

`fclose` | `finput` | `fname` | `fopen` | `fprint` | `fread` | `ftextinput` | `pathname` | `print` | `read` | `READPATH` | `write` | `WRITEPATH`

psi

Digamma/polygamma function

Syntax

```
psi(x)
```

```
psi(x, n)
```

Description

`psi(x)` represents the digamma function, i.e., the logarithmic derivative $\psi(x) = \frac{\Gamma'(x)}{\Gamma(x)}$ of the `gamma` function.

`psi(x, n)` represents the n-th polygamma function, i.e., the n-th derivative $\psi^{(n)}(x)$.

`psi(x, 0)` is equivalent to `psi(x)`.

The digamma/polygamma function is defined for all complex arguments `x` apart from the singular points 0, - 1, - 2, ….

If `x` is a floating-point value, then a floating point value is returned.

Simplifications are implemented for rational numbers x. In particular, if x = `numer(x)/k` with denominators k = 1, 2, 3, 4 or 6, explicit results expressed in terms of EULER, PI, and `ln` are returned. In general, for any rational `x` with `|x| (n + 1) ≤ 6 Pref::autoExpansionLimit() = 6000` (see `Pref::autoExpansionLimit`), the functional equation

$$\psi^{(n)}(x+1) = \psi^{(n)}(x) + \frac{(-1)^n n!}{x^{n+1}},$$

is used to obtain a result with an argument `x` from the interval $(0, 1]$. Use `expand(psi(x, n))` to obtain such a shift of the argument for larger values of `x`.

Some explicit formulas are implemented including

$$\psi(1) = -\mathrm{EULER},$$

$$\psi^{(n)}(1) = (-1)^{n+1}\, n!\, \zeta(n+1),\ n > 1,$$

$$\psi\left(\frac{1}{2}\right) = -2\ln(2) - \mathrm{EULER},$$

$$\psi^{(n)}\left(\frac{1}{2}\right) = (-1)^{n+1}\, n!\, \left(2^{n+1}-1\right)\, \zeta(n+1),\ n > 1.$$

The special values $\psi(\infty) = \infty$ and $\psi^{(n)}(\infty) = 0$ for $n > 0$ are implemented.

For all other arguments, a symbolic function call of `psi` is returned.

The float attribute of the digamma function `psi(x)` is a kernel function, i.e., floating-point evaluation is fast. The float attribute of the polygamma function `psi(x, n)` with `n > 0` is a library function. Note that `psi(float(x))` and `psi(float(x), n)` rather than `float(psi(x))` and `float(psi(x, n))` should be used for float evaluation because, for rational values of `x`, the computation of the symbolic result `psi(x)`, `psi(x, n)` may be costly. Further, the float evaluation of the symbolic result may be numerically unstable.

The `expand` attribute uses the functional equation

$$\psi^{(n)}(x+1) = \psi^{(n)}(x) + \frac{(-1)^n\, n!}{x^{n+1}},$$

the nth derivative of the reflection formula

$$\psi(-x) = \psi(x) + \frac{1}{x} + \pi\, \cot(\pi\, x),$$

and the Gauß multiplication formula for $\psi^{(n)}(k\,x)$ when k is a positive integer, to rewrite `psi(x, n)`. For numerical `x`, the functional equation is used to shift the argument to the range `0 < x < 1`. Cf. examples “Example 3” on page 1-1661 and “Example 4” on page 1-1661.

Environment Interactions

When called with a floating-point value `x`, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
psi(-3/2), psi(4, 1), psi(3/2, 2)
```

$$\frac{8}{3} - 2\ln(2) - \text{EULER},\ \frac{\pi^2}{6} - \frac{49}{36},\ 16 - 14\,\zeta(3)$$

```
psi(x + sqrt(2), 4), psi(infinity, 5)
```

$$\psi^{(4)}(x + \sqrt{2}),\ 0$$

Floating point values are computed for floating-point arguments:

```
psi(-5.2), psi(1.0, 3), psi(2.0 + 3.0*I, 10)
```

$$6.065773152,\ 6.493939402,\ 0.7526409593 - 2.299472238\,i$$

Example 2

`psi` is singular for nonpositive integers:

```
psi(-2)
```

```
Error: Singularity. [psi]
```

Example 3

For positive integers and rational numbers x with denominators 2, 3, 4 and 6, respectively, the result is expressed in terms of EULER, PI, `ln`, and `zeta` if `|x| (n + 1) ≤ 6 Pref::autoExpansionLimit() = 6000`:

```
Pref::autoExpansionLimit()
```

$$1000$$

```
psi(-5/2), psi(-3/2, 1), psi(13/3, 2), psi(11/6, 4)
```

$$\frac{46}{15} - 2\ln(2) - \text{EULER},\ \frac{\pi^2}{2} + \frac{40}{9},\ \frac{75535713}{1372000} - \frac{4\sqrt{3}\,\pi^3}{9} - 26\,\zeta(3),$$
$$176\sqrt{3}\,\pi^5 - 90024\,\zeta(5) + \frac{186624}{3125}$$

For larger arguments, use `expand` to obtain such expressions:

```
psi(1001, 5)
```

$$\psi^{(5)}(1001)$$

```
expand(%)
```

```
     6 8 PI ----- -   63 13353338895557883398775 68.../109380825689800611313229 6...
```

Example 4

The functions `diff`, `expand`, `float`, `limit`, and `series` handle expressions involving `psi`:

```
diff(psi(x^2 + 1, 3), x), float(ln(3 + psi(sqrt(PI))))
```

$$2\,x\,\psi^{(4)}(x^2+1),\ 1.183103343$$

```
expand(psi(2*x + 3, 2))
```

$$\frac{\psi''\left(x+\frac{1}{2}\right)}{8}+\frac{2}{(2x+1)^3}+\frac{2}{(2x+2)^3}+\frac{1}{4x^3}+\frac{\psi''(x)}{8}$$

```
limit(x*psi(x), x = 0), limit(psi(x, 3), x = infinity)
```

$$-1,\ 0$$

```
series(psi(x), x = 0), series(psi(x, 3), x = infinity, 3)
```

$$-\frac{1}{x}-\mathrm{EULER}+\frac{\pi^2 x}{6}-x^2\,\zeta(3)+\frac{\pi^4 x^3}{90}-x^4\,\zeta(5)+O(x^5),\ \frac{2}{x^3}+\frac{3}{x^4}+\frac{2}{x^5}+O\left(\frac{1}{x^6}\right)$$

Parameters

x

An arithmetical expression

n

A nonnegative integer

Return Values

Arithmetical expression.

Overloaded By

x

See Also

MuPAD Functions

`beta` | `binomial` | `fact` | `gamma` | `harmonic` | `lngamma` | `zeta`

radsimp

Simplify radicals in arithmetical expressions

Syntax

```
radsimp(z)
```

Description

`radsimp(z)` tries to simplify the radicals in the expression `z`. The result is mathematically equivalent to `z`.

`radsimp` and `simplifyRadical` are equivalent.

Examples

Example 1

Simplify these constant expressions with square roots and higher order radicals:

```
radsimp(3*sqrt(7)/(sqrt(7) - 2)),
radsimp(sqrt(5 + 2*sqrt(6)));
radsimp(sqrt(5*sqrt(3) + 6*sqrt(2))),
radsimp(sqrt(3 + 2*sqrt(2)))
```

$$2\sqrt{7}+7,\ \sqrt{2}+\sqrt{3}$$

$$\sqrt{2}\,3^{1/4}+3^{3/4},\ \sqrt{2}+1$$

```
radsimp((1/2 + 1/4*3^(1/2))^(1/2))
```

$$\frac{\sqrt{2}}{4}+\frac{\sqrt{6}}{4}$$

```
radsimp((5^(1/3) - 4^(1/3))^(1/2))
```

$$\frac{2^{2/3}\,5^{1/3}}{3} + \frac{2^{1/3}}{3} - \frac{5^{2/3}}{3}$$

```
radsimp(sqrt(3*sqrt(3 + 2*sqrt(5 - 12*sqrt(3 - 2*sqrt(2))))
             + 14))
```

$$\sqrt{2} + 3$$

```
radsimp(2*2^(1/4) + 2^(3/4) - (6*2^(1/2) + 8)^(1/2))
```

$$0$$

```
radsimp(sqrt(1 + sqrt(3)) + sqrt(3 + 3*sqrt(3))
             - sqrt(10 + 6*sqrt(3)))
```

$$0$$

Example 2

Create the following expression and then simplify it using `radsimp`:

```
x := sqrt(3)*I/2 + 1/2: y := x^(1/3) + x^(-1/3): z := y^3 - 3*y
```

$$-\frac{3}{\left(\frac{1}{2}+\frac{\sqrt{3}\,i}{2}\right)^{1/3}} - 3\left(\frac{1}{2}+\frac{\sqrt{3}\,i}{2}\right)^{1/3} + \left(\frac{1}{\left(\frac{1}{2}+\frac{\sqrt{3}\,i}{2}\right)^{1/3}} + \left(\frac{1}{2}+\frac{\sqrt{3}\,i}{2}\right)^{1/3}\right)^3$$

```
radsimp(z)
```

$$1$$

```
delete x, y, z:
```

Example 3

Use `radsimp` to simplify these arithmetical expressions containing variables:

```
z := x/(sqrt(3) - 1) - x/2
```

$$\frac{x}{\sqrt{3}-1}-\frac{x}{2}$$

```
radsimp(z) = expand(radsimp(z))
```

$$x\left(\frac{\sqrt{3}}{2}+\frac{1}{2}\right)-\frac{x}{2}=\frac{\sqrt{3}\,x}{2}$$

```
delete z:
```

Example 4

Use `radsimp` to simplify nested radicals. When simplifying nested radicals, `radsimp` tries to reduce the nesting depth:

```
radsimp((6*2^(1/2) + 8)^(1/2));
radsimp(((32/5)^(1/5) - (27/5)^(1/5))^(1/3));
radsimp(sqrt((3+2^(1/3))^(1/2) * (4-2^(1/3))^(1/2)))
```

$$2\,2^{1/4}+2^{3/4}$$

$$\frac{5^{3/5}\left(3^{1/5}-3^{2/5}+1\right)}{5}$$

$$\left(2^{1/3}-2^{2/3}+12\right)^{1/4}$$

Parameters

z

An arithmetical expression

Return Values

Arithmetical expression.

References

Borodin A., Fagin R., Hopcroft J.E., and Tompa M.: Decreasing the Nesting Depth of Expressions Involving Square Roots, JSC 1, 1985, pp. 169-188.

See Also

MuPAD Functions

`combine` | `ifactor` | `normal` | `rectform` | `simplify` | `simplifyRadical`

More About

- “Manipulate Expressions”
- “Choose Simplification Functions”

simplifyRadical

Simplify radicals in arithmetical expressions

Syntax

```
simplifyRadical(z)
```

Description

`simplifyRadical(z)` tries to simplify the radicals in the expression `z`. The result is mathematically equivalent to `z`.

`radsimp` and `simplifyRadical` are equivalent.

Examples

Example 1

Simplify these constant expressions with square roots and higher order radicals:

```
simplifyRadical(3*sqrt(7)/(sqrt(7) - 2)),
simplifyRadical(sqrt(5 + 2*sqrt(6)));
simplifyRadical(sqrt(5*sqrt(3) + 6*sqrt(2))),
simplifyRadical(sqrt(3 + 2*sqrt(2)))
```

$$2\sqrt{7}+7,\ \sqrt{2}+\sqrt{3}$$

$$\sqrt{2}\,3^{1/4}+3^{3/4},\ \sqrt{2}+1$$

```
simplifyRadical((1/2 + 1/4*3^(1/2))^(1/2))
```

$$\frac{\sqrt{2}}{4}+\frac{\sqrt{6}}{4}$$

```
simplifyRadical((5^(1/3) - 4^(1/3))^(1/2))
```

$$\frac{2^{2/3}\,5^{1/3}}{3} + \frac{2^{1/3}}{3} - \frac{5^{2/3}}{3}$$

```
simplifyRadical(sqrt(3*sqrt(3 + 2*sqrt(5 - 12*sqrt(3 - 2*sqrt(2))))
            + 14))
```

$$\sqrt{2} + 3$$

```
simplifyRadical(2*2^(1/4) + 2^(3/4) - (6*2^(1/2) + 8)^(1/2))
```

$$0$$

```
simplifyRadical(sqrt(1 + sqrt(3)) + sqrt(3 + 3*sqrt(3))
            - sqrt(10 + 6*sqrt(3)))
```

$$0$$

Example 2

Create the following expression and then simplify it using `simplifyRadical`:

```
x := sqrt(3)*I/2 + 1/2: y := x^(1/3) + x^(-1/3): z := y^3 - 3*y
```

$$-\frac{3}{\left(\frac{1}{2}+\frac{\sqrt{3}\,i}{2}\right)^{1/3}} - 3\left(\frac{1}{2}+\frac{\sqrt{3}\,i}{2}\right)^{1/3} + \left(\frac{1}{\left(\frac{1}{2}+\frac{\sqrt{3}\,i}{2}\right)^{1/3}} + \left(\frac{1}{2}+\frac{\sqrt{3}\,i}{2}\right)^{1/3}\right)^3$$

```
simplifyRadical(z)
```

$$1$$

```
delete x, y, z:
```

Example 3

Use `simplifyRadical` to simplify these arithmetical expressions containing variables:

```
z := x/(sqrt(3) - 1) - x/2
```

$$\frac{x}{\sqrt{3}-1}-\frac{x}{2}$$

```
simplifyRadical(z) = expand(radsimp(z))
```

$$x\left(\frac{\sqrt{3}}{2}+\frac{1}{2}\right)-\frac{x}{2}=\frac{\sqrt{3}\,x}{2}$$

```
delete z:
```

Example 4

Use `simplifyRadical` to simplify nested radicals. When simplifying nested radicals, `simplifyRadical` tries to reduce the nesting depth:

```
simplifyRadical((6*2^(1/2) + 8)^(1/2));
simplifyRadical(((32/5)^(1/5) - (27/5)^(1/5))^(1/3));
simplifyRadical(sqrt((3+2^(1/3))^(1/2) * (4-2^(1/3))^(1/2)))
```

$$2\,2^{1/4}+2^{3/4}$$

$$\frac{5^{3/5}\left(3^{1/5}-3^{2/5}+1\right)}{5}$$

$$\left(2^{1/3}-2^{2/3}+12\right)^{1/4}$$

Parameters

z

An arithmetical expression

Return Values

Arithmetical expression.

Algorithms

For constant algebraic expressions, `simplifyRadical` constructs a tower of algebraic extensions of $\mathbb{Q}$ using the domain `Dom::AlgebraicExtension`. It tries to return the simplest possible form.

This function is based on an algorithm described in Borodin, Fagin, Hopcroft and Tompa, "Decreasing the Nesting Depth of Expressions Involving Square Roots", JSC 1, 1985, pp. 169-188.In some special cases, an algorithm based on Landau, "How to tangle with a nested radical", The Mathematical Intelligencer 16, 1994, no. 2, pp. 49-55, is used.

See Also

MuPAD Functions

`combine` | `ifactor` | `normal` | `radsimp` | `rectform` | `simplify`

More About

- "Manipulate Expressions"
- "Choose Simplification Functions"

random

Generate random integer numbers

Syntax

```
random()
```

```
random(n1 .. n2)
```

```
random(n)
```

Description

`random()` returns a random integer number between 0 and 10^{12}.

`random(n1..n2)` returns a procedure that generates random integers between `n1` and `n2`.

The calls `random()` return uniformly distributed random integers between 0 and 999999999988 (approximately 10^{12}).

`r := random(n1..n2)` produces a random number generator `r`. Subsequent calls `r()` generate uniformly distributed random integers between `n1` and `n2`.

`random(n)` is equivalent to `random(0 .. n - 1)`.

The global variable `SEED` is used for initializing or changing the sequence of random numbers. It may be assigned any *non-zero* integer. The value of `SEED` fixes the sequence of random numbers. This may be used to reset random generators and reproduce random sequences.

`SEED` is set to a default value when MuPAD is initialized. Thus, each time MuPAD is started or re-initialized with the `reset` function, the random generators produce the same sequence of numbers.

To get a non-predictable initial value, make it dependent on the current time. See “Example 5” on page 1-1674.

Several random generators produced by `random` may run simultaneously. All generators make use of the same global variable `SEED`.

For producing uniformly distributed floating-points numbers, it is recommended to use the faster function `frandom` instead. The stats library provides random generators with various other distributions. Cf. “Example 4” on page 1-1673.

Environment Interactions

`random` as well as the random number generators created by it are sensitive to the environment variable `SEED`.

`random` and the random number generators created by it change the environment variable `SEED` on each call.

Examples

Example 1

The following call produces a sequence of random integers. Note that an index variable `i` must be used in the construction of the sequence. A call such as `random() $8` would produce 8 copies of the same random value:

```
random() $ i = 1..8
```

```
427419669081, 321110693270, 343633073697, 474256143563, 558458718976, 746753830538,
  32062222085, 722974121768
```

The following call produces a “die” that is rolled 20 times:

```
die := random(1..6): die() $ i = 1..20
```

```
     2, 2, 2, 4, 4, 3, 3, 2, 1, 4, 4, 6, 1, 1, 1, 2, 4, 2, 1, 3
```

The following call produces a “coin” that produces “head” or “tail”:

```
coin := random(2): coin() $ i = 1..10
```

```
     1, 0, 1, 1, 0, 1, 0, 1, 0, 0
```

```
subs(%, [0 = head, 1 = tail])
```

```
     tail, head, tail, tail, head, tail, head, tail, head, head
```

```
delete dice, coin:
```

Example 2

`random` is sensitive to the global variable `SEED` which is set and reset when MuPAD is (re-)initialized. The seed may also be set by the user. Random sequences can be reproduced by starting with a fixed `SEED`:

```
SEED := 1: random() $ i = 1..4
```

```
     427419669081, 321110693270, 343633073697, 474256143563
```

```
SEED := 1: random() $ i = 1..4
```

```
     427419669081, 321110693270, 343633073697, 474256143563
```

Example 3

`random` allows you to create several random number generators for different ranges of numbers, and to use them simultaneously:

```
r1 := random(0..4): r2 := random(2..9): [r1(), r2()] $ i = 1..6
```

```
     [1, 4], [0, 2], [1, 3], [0, 5], [2, 2], [4, 7]
```

```
delete r1, r2:
```

Example 4

`random` can be used to build a random generator for uniformly distributed floating-point numbers. The following generator produces such numbers between `-1.0` and `1.0`:

```
r := float@random(-10^DIGITS..10^DIGITS)/10^DIGITS:
r() $ i = 1..12;
```

0.2920457876, 0.3747019439, −0.5968604725, −0.9375052697, 0.1053530039, −0.3513692809,
0.5590763459, −0.0607326312, −0.4571489053, 0.2600608968, 0.9760099364, 0.5982933733

However, it is strongly recommended to use the much more efficient functions `frandom` or `stats::uniformRandom` instead:

```
r := stats::uniformRandom(-1, 1, Seed = 10^10):
r() $ i = 1..12
```

−0.5438091778, 0.1842867446, −0.9859463167, −0.6071964914, −0.8190627066,
−0.4262015812, 0.978028969, 0.4404626935, 0.05402948609,
0.3740704365, −0.2952265339, −0.06597080227

```
delete r:
```

Example 5

Usually, `random` is used to generate experimental input or "random" examples. In these cases, reproducibility is a good thing. However, on occasion a "more random" sequence is desirable. The usual way to get a random seed in a program is to use the current system time:

```
SEED := round(1e10*frandom(CurrentTime()))
```

```
1035804049
```

```
random(), random()
```

```
861209862222, 269921735546
```

Parameters

n_1, n_2

Integers with $n_1 < n_2$

n

A positive integer

Return Values

`random()` returns a nonnegative integer. The calls `random(n1..n2)` and `random(n)` return a procedure of type `DOM_PROC`.

Algorithms

`random` implements a linear congruence generator. The sequence of pseudo-random numbers generated by calling `random()` over and over again is $f(x)$, $f(f(x))$, …., where x is the initial value of `SEED` and f is the function mapping x to $a\,x \bmod m$ with suitable integer constants a and m.

See Also

MuPAD Functions

`frandom` | `stats::uniformRandom`

rationalize

Transform an expression into a rational expression

Syntax

```
rationalize(object, options)
```

Description

`rationalize(object)` transforms the expression `object` into an equivalent rational expression by replacing non-rational subexpressions by newly generated variables.

By default, a rational expression is an expression that contains only sums, products, powers with integer exponents, integers, rational numbers, and identifiers as subexpressions.

The `rationalize` function returns a sequence `(rat, subsSet)`. The rationalized expression `rat` contains new variables. The set of substitutions `subsSet` expresses the new variables by the old ones.

If the original expression contains subexpressions, the `rationalize` function can rationalize or replace subexpressions or keep them in their original form. Use the options `DescendInto`, `ReplaceType`, and `StopOn` to control the action `rationalize` takes for particular types of subexpressions.

If `FindRelations = ["exp", "_power", "sin"]`, the `rationalize` function detects maximal number of algebraic dependencies.

If you call `rationalize` with any combination of the following three contradicting options, the function chooses the option using the following priorities: `ReplaceType`, `StopOn`, `DescendInto`. For example, if you specify the same type of subexpression with `StopOn` and `DescendInto`, the `rationalize` function uses only the `StopOn` option for subexpressions of the specified type. If you combine any of these options with the `ReplaceType` option, `rationalize` uses only the `ReplaceType` option.

Examples

Example 1

`rationalize` operates on single arithmetical expressions, lists, and sets of expressions:

```
rationalize(2*sqrt(3) + 0.5*x^3)
```

$$X4\,x^3 + 2\,X3,\ \{X3 = \sqrt{3},\ X4 = 0.5\}$$

```
rationalize([(x - sqrt(2))*(x^2 + sqrt(3)),
             (x - sqrt(2))*(x - sqrt(3))])
```

$$[-(X5 - x)\,(x^2 + X6),\ (X5 - x)\,(X6 - x)],\ \{X5 = \sqrt{2},\ X6 = \sqrt{3}\}$$

Example 2

Use the `ApproximateFloats` option to replace all floating-point numbers with rational numbers:

```
rationalize([0.4, 0.333, 0.74], ApproximateFloats)
```

$$\left[\frac{2}{5},\ \frac{333}{1000},\ \frac{37}{50}\right],\ \varnothing$$

If you use both `ApproximateFloats` and `ReplaceTypes` options, `ApproximateFloats` does not apply to the types of subexpressions specified in `ReplaceTypes`:

```
rationalize(0.4*x^2 + sin(0.33/x),
                ApproximateFloats,
                ReplaceTypes={DOM_FLOAT})
```

$$X8\,x^2 + X9,\ \left\{X7 = 0.33,\ X8 = 0.4,\ X9 = \sin\left(\frac{0.33}{x}\right)\right\}$$

Instead of specifying the value of `ReplaceTypes` as a sequence of types, you can specify it as a function. The function must return `TRUE` or `FALSE` as a result. For example, rationalize the same expression $0.4\,x^2 + \sin\left(\frac{0.33}{x}\right)$. This time, use the function `F` to specify the type of subexpressions which you want to replace by variables:

```
F := X -> testtype(X, DOM_FLOAT):
rationalize(0.4*x^2 + sin(0.33/x),
                ApproximateFloats,
                ReplaceTypes = F)
```

$$X11\, x^2 + X12,\ \left\{X10 = 0.33,\ X11 = 0.4,\ X12 = \sin\left(\frac{0.33}{x}\right)\right\}$$

Example 3

By default, `rationalize` rationalizes sums, products, bases of integer powers, lists, and sets:

```
rationalize(ln(sin(x)^2 + cos(x)*exp(x)))
```

$$X13,\ \{X13 = \ln(\sin(x)^2 + e^x \cos(x))\}$$

The `DescendInto` option lets you specify the types of subexpressions that you want to rationalize. Each type can be a domain type, a string as returned by the function `type` or a Type object. Note that `DescendInto` overwrites the default types with the types that you specify:

```
rationalize(ln(sin(x)^2 + cos(x)*exp(x)), DescendInto = {"ln"})
```

$$\ln(X14),\ \{X14 = \sin(x)^2 + e^x \cos(x)\}$$

If you want to add new types of subexpressions to the default ones, define the value of `DescendInto` as a procedure that specifies all required types explicitly. The procedure must return `TRUE` or `FALSE`:

```
F := proc(X)
begin
  hastype(X, {"_plus", "_mult", DOM_SET, DOM_LIST, "ln"}) or
  (hastype(X, "_power") and hastype(op(X, 2), DOM_INT))
end:

rationalize(ln(sin(x)^2 + cos(x)*exp(x)), DescendInto = F)
```

$$\ln(X17^2 + X15\, X16),\ \{X16 = \cos(x),\ X15 = e^x,\ X17 = \sin(x)\}$$

Example 4

Use the `MinimalPolynomials` option to find minimal polynomials of irrational expressions:

```
rationalize(x^(7/6) + x^(3/2), MinimalPolynomials)
```

$$\mathrm{X18} + \mathrm{X19},\ \left\{\mathrm{X18} = x^{3/2},\ \mathrm{X19} = x^{7/6}\right\},\ \left\{\mathrm{X18}^2 - x^3,\ \mathrm{X19}^6 - x^7\right\}$$

Example 5

Use `Prefix = s`, where `s` is a string, to specify the prefix for generated variables (the default prefix is `X`):

```
rationalize(x^(7/6) + x^(3/2), Prefix = "ABC")
```

$$\mathrm{ABC1} + \mathrm{ABC2},\ \left\{\mathrm{ABC1} = x^{3/2},\ \mathrm{ABC2} = x^{7/6}\right\}$$

Example 6

Use the `ReplaceHardToEval` option to replace limits, sums, and integrals with generated variables. Expressions with limits, sums, and integrals tend to be the most computationally expensive:

```
rationalize(sum(exp(x)/(x^2 + 1), x) +
            limit(sin(cos(1/x))*cos(1/x), x),
                          ReplaceHardToEval)
```

$$\mathrm{X48} + \mathrm{X49},\ \left\{\mathrm{X48} = \lim_{x \to 0} \cos\left(\frac{1}{x}\right) \sin\left(\cos\left(\frac{1}{x}\right)\right),\ \mathrm{X49} = \sum_x \frac{e^x}{x^2 + 1}\right\}$$

Example 7

By default, `rationalize` avoids rationalization of integers, rational numbers, and identifiers:

```
rationalize(2*sqrt(3) + 0.5*x^3)
```

$$\mathrm{X51}\,x^3 + 2\,\mathrm{X50},\ \{\mathrm{X50} = \sqrt{3},\ \mathrm{X51} = 0.5\}$$

The `DescendInto` option lets you avoid rationalization of particular types of subexpressions. Each type can be specified as a domain type, a string as returned by the function `type`, or a Type object. For example, rationalize the same expression leaving the subexpression `x^3` (of the type `"_power"`) unchanged:

```
rationalize(2*sqrt(3) + 0.5*x^3, StopOn = {"_power"})
```

$$\mathrm{X53}\,x^3 + \sqrt{3}\,\mathrm{X52},\ \{\mathrm{X52} = 2,\ \mathrm{X53} = 0.5\}$$

Rationalize the same expression including all subexpressions. Keep floating-point numbers, integers, and identifiers (do not replace them with generated variables):

```
rationalize(2*sqrt(3) + 0.5*x^3,
            StopOn = {DOM_FLOAT, DOM_INT, DOM_IDENT})
```

$$0.5\,x^3 + 2\,\mathrm{X54},\ \{\mathrm{X54} = \sqrt{3}\}$$

Note that `StopOn` overwrites the default types with the types that you specify. If you want to add new types of subexpressions to the default ones, specify all the types explicitly:

```
rationalize(2*sqrt(3) + 0.5*x^3,
            StopOn = {DOM_INT, DOM_IDENT, DOM_RAT, DOM_FLOAT})
```

$$0.5\,x^3 + 2\,\mathrm{X55},\ \{\mathrm{X55} = \sqrt{3}\}$$

```
rationalize(2*sqrt(3) + 0.5*x^3,
            StopOn = {DOM_INT, DOM_IDENT,
                      DOM_RAT, DOM_FLOAT,
                               "_power"})
```

$$0.5\,x^3 + 2\sqrt{3},\ \varnothing$$

The `StopOn` option also can accept a function as its value. The function must return `TRUE` or `FALSE`. For example, use generated variables to replace only subexpressions that contain `sin`. Keep all other subexpressions intact:

```
F := X -> not hastype(X, "sin"):
rationalize(sin(x^2) + x^3 + exp(x) + 1/x, StopOn = F)
```

$$X56 + e^x + \frac{1}{x} + x^3, \{X56 = \sin(x^2)\}$$

Example 8

Use the `FindRelations` option to detect algebraic dependencies between exponentials:

```
rationalize(exp(x/2) + exp(x/3), FindRelations = ["exp"])
```

$$X63^3 + X63^2, \left\{X63 = e^{\frac{x}{6}}\right\}$$

Detect algebraic dependencies for different powers of the same base by specifying the type `"_power"`:

```
rationalize(x^(3/2) + x^(7/4), FindRelations = ["_power"])
```

$$X67^7 + X67^6, \left\{X67 = x^{1/4}\right\}$$

Detect algebraic dependencies for trigonometric functions by specifying the type `"sin"` or `"cos"`:

```
rationalize(sin(x) + cos(x), FindRelations = ["sin"]);
rationalize(sin(x)^3 + cos(x)^3, FindRelations = ["cos"])
```

$$\frac{2\,X70}{X70^2+1} - \frac{X70^2-1}{X70^2+1}, \left\{X70 = \tan\left(\frac{x}{2}\right)\right\}$$

$$\frac{8\,X73^3}{(X73^2+1)^3} - \frac{(X73^2-1)^3}{(X73^2+1)^3}, \left\{X73 = \tan\left(\frac{x}{2}\right)\right\}$$

Example 9

For nested exponentials, use the `Recursive` option to obtain a list of substitutions:

```
rationalize(exp(exp(x)), FindRelations = ["exp"], Recursive)
```

$$X74,\ [X74 = e^{X77},\ X77 = e^{x}]$$

The option also works for trigonometric functions:

```
rationalize(sin(sin(x)), FindRelations = ["sin"], Recursive)
```

$$X78,\ [X78 = \sin(X79),\ X79 = \sin(x)]$$

Example 10

The `ShowDependencies` option shows all original variables upon which each generated variable depends:

```
rationalize(sin(x)^3, ShowDependencies)
```

$$X80(x)^3,\ \{X80(x) = \sin(x)\}$$

Parameters

`object`

Any MuPAD object

Options

`ApproximateFloats`

When you use the `ApproximateFloats` option, the `rationalize` function replaces floating-point numbers with rational numbers. By default, `ApproximateFloats=FALSE`: the `rationalize` function replaces all floating-point numbers with the

new variables. If you rationalize an expression using both `ApproximateFloats` and `StopOn` options, `StopOn` does not prevent rationalization of floating-point numbers in the specified subexpressions. If you rationalize an expression using both `ApproximateFloats` and `ReplaceTypes` options, `ApproximateFloats` does not apply to the types of subexpressions specified in `ReplaceTypes`. See "Example 2" on page 1-1677.

`DescendInto`

When you use the `DescendInto` option, the `rationalize` function rationalizes all subexpressions of the specified types. You can specify the value of this option as a set (even if there is only one type) or a procedure that returns `TRUE` or `FALSE`. Each type can be

- A domain type (such as `DOM_INT`, `DOM_EXPR`, and so on)
- A string as returned by the function `type` (such as `"_plus"`, `"_mult"`, `"sin"`, and so on)
- A Type object (`Type::Boolean`, `Type::Equation`, and so on)

By default, the `rationalize` function rationalizes the following types of subexpresssions: sums, products, bases of integer powers, lists, and sets. When you specify other types of subexpressions, `rationalize` uses them instead of the default types. (`DescendInto` overwrites the default types with the types that you specify.) If you want to extend the set of types of subexpressions retaining the default types, define the value of `DescendInto` as a procedure that specifies all default and additional types explicitly. See "Example 3" on page 1-1678.

`FindRelations`

When you use the `FindRelations` option, the `rationalize` function detects algebraic dependencies for subexpressions of specified types. This option accepts the types of subexpressions in the form of a list. The following types are available: `"sin"`, `"cos"`, `"exp"`, and `"_power"`. By default, `rationalize` does not look for dependencies for irrational subexpressions: `FindRelations= []`.

`MinimalPolynomials`

When you use the `MinimalPolynomials` option, the `rationalize` function returns the minimal polynomials of irrational expressions. The function returns the rationalized expression, the set of substitution equations, and minimal polynomials. By default, `MinimalPolynomials= FALSE`. See "Example 4" on page 1-1679.

Prefix

Use the `Prefix` option to specify the prefix for new variables generated by the `rationalize` function. The value of this option must be a string. By default, `Prefix= "X"`. See “Example 5” on page 1-1679.

Recursive

When you use the `Recursive` option, the `rationalize` function recursively rationalizes nested subexpressions, and returns a list of substitution equations. Each generated variable in the returned list can depend on other variables in the list. By default, `Recursive= FALSE`. See “Example 9” on page 1-1682.

ReplaceHardToEval

When you use the `ReplaceHardToEval` option, the `rationalize` function replaces all limits, sums, and integrals by generated variables. Generally, this option allows you to avoid most expensive rationalizations of sums, limits, and integrals. By default, `ReplaceHardToEval= FALSE`. See “Example 6” on page 1-1679.

ReplaceTypes

When you use the `ReplaceTypes` option, the `rationalize` function replaces all subexpressions of the specified types with generated variables. You can specify the value of this option as a set (even if there is only one type) or a procedure that returns `TRUE` or `FALSE`. Each type can be

- A domain type (such as `DOM_INT`, `DOM_EXPR`, and so on)
- A string as returned by the function `type` (such as `"_plus"`, `"_mult"`, `"sin"`, and so on)
- A Type object (`Type::Boolean`, `Type::Equation`, and so on)

This option allows you to specify and avoid most expensive rationalizations for your particular expression. If you use this option in combination with `ReplaceHardToEval`, the `rationalize` function uses generated variables to replace all limits, sums, integrals, and the types that you specify. If `ReplaceTypes` specifies the same type of subexpression as `DescendInto`, the `ReplaceTypes` option prevails. By default, `ReplaceTypes= {}`.

Alternatively, specify the value of this option as a function that returns `TRUE` or `FALSE`. See “Example 2” on page 1-1677.

ShowDependencies

When you use the ShowDependencies option, the rationalize function replaces any irrational subexpression containing the identifiers vars with an expression of the form newvar(vars), showing the dependencies of the generated variables on the original variables. By default, ShowDependencies= FALSE.

StopOn

When you use the StopOn option, the rationalize function does not rationalize the specified types of subexpressions. You can specify the value of this option as a set (even if there is only one type) or a function that returns TRUE or FALSE. Each type can be

- A domain type (such as DOM_INT, DOM_EXPR, and so on)
- A string as returned by the function type (such as "_plus", "_mult", "sin", and so on)
- A Type object (Type::Boolean, Type::Equation, and so on)

By default, the rationalize function does not rationalize or replace integers, rational numbers, and identifiers. When you specify other types of subexpressions, rationalize uses them instead of the default types. (StopOn overwrites the default types with the types that you specify.) If you want to extend the set of types of subexpressions retaining the default types, specify StopOn = {DOM_INT, DOM_IDENT, DOM_RAT, extra types}, where extra types are the additional types of subexpressions that you do not want to rationalize. See “Example 7” on page 1-1679.

If StopOn specifies the same type of subexpression as DescendInto, the StopOn option prevails.

StopOnConstants

When you use the StopOnConstants option, the rationalize function does not rationalize the object of the type Type::Constant: numbers, strings, Boolean constants, NIL, FAIL, PI, EULER, and CATALAN in the set Type::ConstantIdents. By default, StopOnConstants= FALSE.

Return Values

Sequence consisting of the rationalized object and a set of substitution equations. If you use the Recursive option, the rationalize function returns a list of substitution

equations instead of a set. If you use the `MinimalPolynomials` option, the returned value has a third argument: the minimal polynomials.

See Also

MuPAD Functions

`indets` | `maprat` | `normal` | `rewrite` | `simplify` | `subs`

Re

Real part of an arithmetical expression

Syntax

```
Re(z)
```

```
Re(L)
```

Description

`Re(z)` returns the real part of *z*.

The intended use of `Re` is for constant arithmetical expressions. Especially for numbers, of type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT`, or `DOM_COMPLEX`, the real part is computed directly and very efficiently.

`Re` can handle symbolic expressions. Properties of identifiers are taken into account (see `assume`). An identifier without any property is assumed to be complex. See “Example 2” on page 1-1688.

However, for arbitrary symbolic expressions, `Re` may be unable to extract the real part of `z`. You may then use the function `rectform` (see “Example 3” on page 1-1688). Note, however, that using `rectform` is computationally expensive.

If `Re` cannot extract the whole real part of `z`, then the returned expression contains symbolic `Re` and `Im` calls. The same is true for `Im`. See “Example 2” on page 1-1688.

The `Re` function is automatically mapped to all entries of container objects such as arrays, lists, matrices, polynomials, sets, and tables.

Environment Interactions

These functions are sensitive to properties of identifiers set via `assume`. See “Example 2” on page 1-1688.

Examples

Example 1

The real and the imaginary part of $2\,e^{1+i}$ are:

```
Re(2*exp(1 + I)), Im(2*exp(1 + I))
```

$$2\cos(1)\,e,\ 2\,e\,\sin(1)$$

Example 2

`Re` and `Im` are not able to extract the whole real and imaginary part, respectively, of symbolic expressions containing identifiers without a value. However, in some cases they can still simplify the input expression, as in the following two examples:

```
delete u, v: Re(u + v*I), Im(u + v*I)
```

$$\Re(u) - \Im(v),\ \Im(u) + \Re(v)$$

```
delete z: Re(z + 2), Im(z + 2)
```

$$\Re(z) + 2,\ \Im(z)$$

By default, identifiers without a value are assumed to represent arbitrary complex numbers. You can use `assume` to change this. The following command tells the system that `z` represents only real numbers:

```
assume(z, Type::Real): Re(z + 2), Im(z + 2)
```

$$z + 2,\ 0$$

Example 3

Here is another example of a symbolic expression for which `Re` and `Im` fail to extract its real and imaginary part, respectively:

```
delete z: Re(exp(I*sin(z))), Im(exp(I*sin(z)))
```

$$\Re\left(e^{\sin(z)\,i}\right),\ \Im\left(e^{\sin(z)\,i}\right)$$

You may use the function `rectform`, which splits a complex expression *z* into its real and imaginary part and is more powerful than `Re` and `Im`:

```
r := rectform(exp(I*sin(z)))
```

$$\cos(\cosh(\Im(z))\,\sin(\Re(z)))\,e^{-\cos(\Re(z))\,\sinh(\Im(z))} + \left(\sin(\cosh(\Im(z))\,\sin(\Re(z)))\,e^{-\cos(\Re(z))\,\sinh(\Im(z))}\right) i$$

Then `Re(r)` and `Im(r)` give the real and the imaginary part of `r`, respectively:

```
Re(r)
```

$$\cos(\cosh(\Im(z))\,\sin(\Re(z)))\,e^{-\cos(\Re(z))\,\sinh(\Im(z))}$$

```
Im(r)
```

$$\sin(\cosh(\Im(z))\,\sin(\Re(z)))\,e^{-\cos(\Re(z))\,\sinh(\Im(z))}$$

Example 4

Symbolic expressions of type `"Re"` and `"Im"` always have the property `Type::Real`, even if no identifier of the symbolic expression has a property:

```
is(Re(sin(2*x)), Type::Real)
```

TRUE

Example 5

Advanced users can extend the functions `Re` and `Im` to their own special mathematical functions (see section “Backgrounds” below). To this end, embed your mathematical function into a function environment`f` and implement the behavior of the functions `Re` and `Im` for this function as the slots `"Re"` and `"Im"` of the function environment.

If a subexpression of the form `f(u,..)` occurs in `z`, then `Re` and `Im` issue the call `f::Re(u,..)` and `f::Im(u,..)`, respectively, to the slot routine to determine the real and the imaginary part of `f(u,..)`, respectively.

For illustration, we show how this works for the sine function and the slot `"Re"`. Of course, the function environment `sin` already has a `"Re"` slot. We call our function environment `Sin` in order not to overwrite the existing system function `sin`:

```
Sin := funcenv(Sin):
Sin::Re := proc(u) // compute Re(Sin(u))
  local r, s;
begin
  r := Re(u);
  if r = u then
    return(Sin(u))
  elif not has(r, {hold(Im), hold(Re)}) then
    s := Im(u);
    if not has(s, {hold(Im), hold(Re)}) then
      return(Sin(r)*cosh(s))
    end_if
  end_if;
  return(FAIL)
end:

Re(Sin(2)), Re(Sin(2 + 3*I))
```

$$\mathrm{Sin}(2),\ \mathrm{Sin}(2)\cosh(3)$$

The return value `FAIL` tells the function `Re` that `Sin::Re` was unable to determine the real part of the input expression. The result is then a symbolic `Re` call:

```
delete f, z: Re(2 + Sin(f(z)))
```

$$\Re(\mathrm{Sin}(f(z))) + 2$$

Parameters

z

An arithmetical expression

`L`

A container object: an array, an hfarray, a list, a matrix, a polynomial, a set, or a table.

Return Values

arithmetical expression or a container object containing such expressions

Overloaded By

z

Algorithms

If a subexpression of the form `f(u,..)` occurs in `z` and `f` is a function environment, then `Re` attempts to call the slot `"Re"` of `f` to determine the real part of `f(u,..)`. In this way, you can extend the functionality of `Re` to your own special mathematical functions.

The slot `"Re"` is called with the arguments `u,..` of `f`. If the slot routine `f::Re` is not able to determine the real part of `f(u,..)`, then it must return `FAIL`.

If `f` does not have a slot `"Re"`, or if the slot routine `f::Re` returns `FAIL`, then `f(u,..)` is replaced by the symbolic call `Re(f(u...))` in the returned expression.

The same holds for the function `Im`, which attempts to call the corresponding slot `"Im"` of `f`.

See “Example 5” on page 1-1689.

Similarly, if an element `d` of a library domain`T` occurs as a subexpression of `z`, then `Re` attempts to call the slot `"Re"` of that domain with `d` as argument to compute the real part of `d`.

If the slot routine `T::Re` is not able to determine the real part of `d`, then it must return `FAIL`.

If `T` does not have a slot `"Re"`, or if the slot routine `T::Re` returns `FAIL`, then `d` is replaced by the symbolic call `Re(d)` in the returned expression.

The same holds for the function `Im`, which attempts to call the corresponding slot `"Im"` of the `T`.

See Also

MuPAD Functions

`abs` | `assume` | `conjugate` | `Im` | `rectform` | `sign`

Im

Imaginary part of an arithmetical expression

Syntax

```
Im(z)
```

```
Im(L)
```

Description

`Im(z)` returns the imaginary part of *z*.

The intended use of `Im` is for constant arithmetical expressions. Especially for numbers, of type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT`, or `DOM_COMPLEX`, the imaginary part is computed directly and very efficiently.

`Im` can handle symbolic expressions. Properties of identifiers are taken into account (see `assume`). An identifier without any property is assumed to be complex. See “Example 2” on page 1-1694.

However, for arbitrary symbolic expressions, `Im` may be unable to extract the imaginary part of `z`. You may then use the function `rectform` (see “Example 3” on page 1-1694). Note, however, that using `rectform` is computationally expensive.

If `Re` cannot extract the whole real part of `z`, then the returned expression contains symbolic `Re` and `Im` calls. The same is true for `Im`. See “Example 2” on page 1-1694.

The `Im` function is automatically mapped to all entries of container objects such as arrays, lists, matrices, polynomials, sets, and tables.

Environment Interactions

These functions are sensitive to properties of identifiers set via `assume`. See “Example 2” on page 1-1694.

Examples

Example 1

The real and the imaginary part of $2\,e^{1+i}$ are:

```
Re(2*exp(1 + I)), Im(2*exp(1 + I))
```

$$2\cos(1)\,e,\ 2\,e\,\sin(1)$$

Example 2

`Re` and `Im` are not able to extract the whole real and imaginary part, respectively, of symbolic expressions containing identifiers without a value. However, in some cases they can still simplify the input expression, as in the following two examples:

```
delete u, v: Re(u + v*I), Im(u + v*I)
```

$$\Re(u) - \Im(v),\ \Im(u) + \Re(v)$$

```
delete z: Re(z + 2), Im(z + 2)
```

$$\Re(z) + 2,\ \Im(z)$$

By default, identifiers without a value are assumed to represent arbitrary complex numbers. You can use `assume` to change this. The following command tells the system that `z` represents only real numbers:

```
assume(z, Type::Real): Re(z + 2), Im(z + 2)
```

$$z + 2,\ 0$$

Example 3

Here is another example of a symbolic expression for which `Re` and `Im` fail to extract its real and imaginary part, respectively:

```
delete z: Re(exp(I*sin(z))), Im(exp(I*sin(z)))
```

$$\Re\left(e^{\sin(z)\,i}\right),\ \Im\left(e^{\sin(z)\,i}\right)$$

You may use the function `rectform`, which splits a complex expression *z* into its real and imaginary part and is more powerful than `Re` and `Im`:

```
r := rectform(exp(I*sin(z)))
```

$$\cos(\cosh(\Im(z))\,\sin(\Re(z)))\,e^{-\cos(\Re(z))\,\sinh(\Im(z))} + \left(\sin(\cosh(\Im(z))\,\sin(\Re(z)))\,e^{-\cos(\Re(z))\,\sinh(\Im(z))}\right) i$$

Then `Re(r)` and `Im(r)` give the real and the imaginary part of `r`, respectively:

```
Re(r)
```

$$\cos(\cosh(\Im(z))\,\sin(\Re(z)))\,e^{-\cos(\Re(z))\,\sinh(\Im(z))}$$

```
Im(r)
```

$$\sin(\cosh(\Im(z))\,\sin(\Re(z)))\,e^{-\cos(\Re(z))\,\sinh(\Im(z))}$$

Example 4

Symbolic expressions of type `"Re"` and `"Im"` always have the property `Type::Real`, even if no identifier of the symbolic expression has a property:

```
is(Re(sin(2*x)), Type::Real)
```

$$\text{TRUE}$$

Example 5

Advanced users can extend the functions `Re` and `Im` to their own special mathematical functions (see section “Backgrounds” below). To this end, embed your mathematical function into a function environment`f` and implement the behavior of the functions `Re` and `Im` for this function as the slots `"Re"` and `"Im"` of the function environment.

If a subexpression of the form `f(u,..)` occurs in `z`, then `Re` and `Im` issue the call `f::Re(u,..)` and `f::Im(u,..)`, respectively, to the slot routine to determine the real and the imaginary part of `f(u,..)`, respectively.

For illustration, we show how this works for the sine function and the slot `"Re"`. Of course, the function environment `sin` already has a `"Re"` slot. We call our function environment `Sin` in order not to overwrite the existing system function `sin`:

```
Sin := funcenv(Sin):
Sin::Re := proc(u) // compute Re(Sin(u))
  local r, s;
begin
  r := Re(u);
  if r = u then
    return(Sin(u))
  elif not has(r, {hold(Im), hold(Re)}) then
    s := Im(u);
    if not has(s, {hold(Im), hold(Re)}) then
      return(Sin(r)*cosh(s))
    end_if
  end_if;
  return(FAIL)
end:

Re(Sin(2)), Re(Sin(2 + 3*I))
```

$$\mathrm{Sin}(2),\ \mathrm{Sin}(2)\cosh(3)$$

The return value `FAIL` tells the function `Re` that `Sin::Re` was unable to determine the real part of the input expression. The result is then a symbolic `Re` call:

```
delete f, z: Re(2 + Sin(f(z)))
```

$$\Re(\mathrm{Sin}(f(z))) + 2$$

Parameters

z

An arithmetical expression

`L`

A container object: an array, an hfarray, a list, a matrix, a polynomial, a set, or a table.

Return Values

arithmetical expression or a container object containing such expressions

Overloaded By

z

Algorithms

If a subexpression of the form `f(u,..)` occurs in `z` and `f` is a function environment, then `Re` attempts to call the slot `"Re"` of `f` to determine the real part of `f(u,..)`. In this way, you can extend the functionality of `Re` to your own special mathematical functions.

The slot `"Re"` is called with the arguments `u,..` of `f`. If the slot routine `f::Re` is not able to determine the real part of `f(u,..)`, then it must return `FAIL`.

If `f` does not have a slot `"Re"`, or if the slot routine `f::Re` returns `FAIL`, then `f(u,..)` is replaced by the symbolic call `Re(f(u...))` in the returned expression.

The same holds for the function `Im`, which attempts to call the corresponding slot `"Im"` of `f`.

See “Example 5” on page 1-1695.

Similarly, if an element `d` of a library domain`T` occurs as a subexpression of `z`, then `Re` attempts to call the slot `"Re"` of that domain with `d` as argument to compute the real part of `d`.

If the slot routine `T::Re` is not able to determine the real part of `d`, then it must return `FAIL`.

If `T` does not have a slot `"Re"`, or if the slot routine `T::Re` returns `FAIL`, then `d` is replaced by the symbolic call `Re(d)` in the returned expression.

The same holds for the function `Im`, which attempts to call the corresponding slot `"Im"` of the `T`.

See Also

MuPAD Functions

`abs` | `assume` | `conjugate` | `Re` | `rectform` | `sign`

read

Search, read, and execute file

Syntax

```
read(filename | n, <Quiet>, <Plain>, <Encoding = "encodingValue">)
```

Description

`read(filename)` searches for the file in various folders:

- First, `filename` is concatenated to each folder given by the environment variable `READPATH`.
- Then the file name is interpreted as an absolute path name.
- Then the file name is interpreted as a relative pathname, i.e., relative to the "working folder."
- Last, the file name is concatenated to the library path.

If a file can be opened with one of these names, then the file is read and executed with `fread`.

`read(filename, Encoding = "encodingValue")` uses the specified encoding for text files. For supported encodings, see "Options" on page 1-1703.

If the file is in `gzip`-compressed format and its name ends in "`.gz`", it will be transparently uncompressed upon reading.

Note that the meaning of "working folder" depends on the operating system. On Microsoft Windows systems and on Mac OS X systems, the "working folder" is the folder where MATLAB is installed. On UNIX systems, it is the current working folder in which MATLAB was started. When started from a menu or desktop item, this is typically the user's home folder.

A path separator ("/") is inserted as necessary when concatenating a given path and `filename`.

`read(n)` with a file descriptor `n` as returned by `fopen` is equivalent to the call `fread(n)`. When called with a file description, `read` does not automatically open and close the file. Use `fopen` and `fclose` to open and close the file. The Encoding option does not work with this syntax.

When you use the `read` command to read a file, the command evaluates all the statements in that file with the maximal substitution depth defined by `LEVEL`. The default value of `LEVEL` for interactive computations is 100. See “Example 3” on page 1-1701.

See the function `fread` for details about reading and executing the file's content and for a detailed description of the options `Plain` and `Quiet`.

When a file is read with `read`, the variable `FILEPATH` contains the path of the file.

Examples

Example 1

Create a new file in the system's temporary folder. The name of the temporary folder varies for different platforms. The `fopen` command with the `TempFile` option creates a file in any system's temporary folder (if such folder exists):

```
a := 3:
b := 5:
fid := fopen(TempFile, Write, Text):
```

Use the `write` command to store values `a` and `b` in the temporary file:

```
write(fid, a, b):
```

Use `fname` to return the name of the temporary file you created:

```
file := fname(fid):
```

After reading the file, the values of `a` and `b` are restored:

```
delete a, b:
read(file):
a, b
```

```
3, 5
```

Alternatively, use `fopen` to open the file and read its content:

```
delete a, b:
n := fopen(file):
read(n):
fclose(n):
a, b
```

```
3, 5
```

```
delete a, b, READPATH, n:
```

Example 2

You can explicitly specify the folder and file names. The following example only works on systems like UNIX. To make it work on other operating systems, change the path names accordingly. First, use `write` to store values in the file "`testfile.mb`" in the "`/tmp`" folder:

```
a := 3:
b := 5:
write("/tmp/testfile.mb", a, b):
```

Now, define "`/tmp`" as the search folder and provide a path name relative to it. Note that the path separator "`/`" is inserted by `read`:

```
delete a, b:
READPATH := "/tmp":
read("testfile.mb"):
a, b
```

```
3, 5
```

Example 3

The `read` command evaluates all the statements in a file it reads with the maximal substitution depth defined by `LEVEL`. For example, create and read a file that specifies

the value of the variable `a` by using another variable `b`. Use the `fopen` command with the `TempFile` option to create a new file in the system's temporary folder:

```
fid := fopen(TempFile, Write, Text):
```

Write the following statements to the file:

```
fprint(Unquoted, fid, "a := b^2:  b := 5: c := a/3: delete a, b:"):
```

Use `fname` to return the name of the temporary file you created. Use `fclose` to close the file:

```
file := fname(fid):
fclose(fid)
```

Read the file. The `read` command evaluates the statements in the file recursively:

```
read(file):
c
```

$$\frac{25}{3}$$

To supress recursive evaluations, change the maximal substitution depth to 1:

```
delete c:
LEVEL := 1:
read(file):
c
```

$$\frac{b^2}{3}$$

Restore the default value of `LEVEL` for further computations:

```
delete LEVEL
```

Example 4

To specify the encoding to read data, use Encoding. The Encoding option applies only to text files that are opened using a file name and not a file descriptor. Open a file and write the statement `"str = abcäöü"` in the encoding “`UTF-8`”:

```
fprint(Unquoted, Text, Encoding="UTF-8",
                       "read_test",
                       "str := \"abcäöü\"")
```

Specify the encoding to read the file. `read` returns the correct output:

```
read("read_test",Encoding="UTF-8"):

"abcäöü"
```

If you do not specify an encoding, the default system encoding is used. Thus, your output might vary from that shown next. Characters unrecognized by the default system encoding are replaced by the default substitution character for that encoding:

```
read("read_test"):

"abc??????"
```

Parameters

`filename`

The name of a file: a character string

`n`

A file descriptor provided by `fopen`: a positive integer

Options

`Plain`

Makes `read` use its own parser context

`Quiet`

Suppresses output during execution of `read`

`Encoding`

This option lets you specify the character encoding to use. The allowed encodings are:

"Big5"	"ISO-8859-1"	"windows-932"
"EUC-JP"	"ISO-8859-2"	"windows-936"
"GBK"	"ISO-8859-3"	"windows-949"
"KSC_5601"	"ISO-8859-4"	"windows-950"
"Macintosh"	"ISO-8859-9"	"windows-1250"
"Shift_JIS"	"ISO-8859-13"	"windows-1251"
"US-ASCII"	"ISO-8859-15"	"windows-1252"
"UTF-8"		"windows-1253"
		"windows-1254"
		"windows-1257"

The default encoding is system dependent. If you specify the encoding incorrectly, characters might read incorrectly. Characters unrecognized by the encoding are replaced by the default substitution character for the specified encoding.

Encodings not listed here can be specified but might not produce correct results.

Return Values

Return value of the last statement of the file.

See Also

MuPAD Functions

`fclose` | `FILEPATH` | `finput` | `fname` | `fopen` | `fprint` | `fread` | `ftextinput` | `import::readbitmap` | `import::readdata` | `input` | `pathname` | `print` | `protocol` | `readbytes` | `READPATH` | `textinput` | `write` | `writebytes` | `WRITEPATH`

readbytes

Read binary data from a file

Syntax

```
readbytes(filename | n, <m>, <format>, <BigEndian | LittleEndian>, <ReturnType = DOM_HF
```

Description

`readbytes(file)` reads binary data from a file.

`readbytes` enables the user to read arbitrary files and interpret their contents as a sequence of numbers.

This function is particularly useful to work on data provided by or destined for external programs. You can use it, for example, to implement encryption or compression algorithms in MuPAD. Cf. “Example 2” on page 1-1708.

The results of `readbytes` depend on the interpretation of the binary data set by the `format` option. When reading a file, you can interpret it as a stream of `Byte`, `SignedByte`, `Short`, `SignedShort`, `Word`, `SignedWord`, `Float` or `Double`. These are standard formats that are used by many program packages to read data. Cf. “Example 1” on page 1-1707.

Be sure to read the data in the appropriate way. You need to know the format used by the program which created the file.

If a hardware float array with complex numbers is written to a file, then first the real parts of the elements are written and then the complex parts are written to the file. Because `readbytes` can only read real values, first one have to create the real and then the complex part to reconstruct the complex array. Cf. “Example 9” on page 1-1712.

The file may be specified directly by its name.

If a file name is specified, `readbytes` opens and closes the file automatically.

If READPATH has no value, readbytes interprets the file name as a pathname relative to the "working directory."

Note that the meaning of "working directory" depends on the operating system. On Microsoft Windows systems and on Mac OS X systems, the "working directory" is the folder where MuPAD is installed. On UNIX systems, it is the current working directory in which MuPAD was started; when started from a menu or desktop item, this is typically the user's home directory.

Absolute path names are processed by readbytes, too.

If a file name is specified, each call to readbytes opens the file at the beginning. If the file was opened via fopen, subsequent calls of readbytes with the corresponding file descriptor start at the point in the file that was reached by the last readbytescommand.

Hence, if you want to read a file by portions, you must open it with fopen and use the returned file descriptor instead of the filename. Cf. "Example 3" on page 1-1708.

Note: If the file is to be opened via fopen, be sure to pass the flag Raw to fopen. Otherwise, readbytes raises an error.

Note: If the number of bytes in the file in a readbytes call is not a multiple of units of the specified format, the data are read up to the last complete number. The remaining bytes are ignored. Cf. "Example 5" on page 1-1709.

Environment Interactions

The function readbytes is sensitive to the environment variable READPATH. First, the file is searched in the "working directory." If it cannot be found there, all paths in READPATH are searched.

The function writebytes is sensitive to the environment variable WRITEPATH. If this variable has a value, the file is created in the corresponding directory. Otherwise, the file is created in the "working directory."

Examples

Example 1

In this example, we write a sequence of numbers to the file `test.tst` with the default settings. Then, we load them back in:

```
writebytes("test.tst", [42, 17, 1, 3, 5, 7, 127, 250]):
readbytes("test.tst")
```

```
[42, 17, 1, 3, 5, 7, 127, 250]
```

We now read the above data with some other option: `SignedByte` interprets all values from 0 to 127 exactly as `Byte` does. Higher values x, however, are interpreted as x - 256. For example, 250 - 256 = - 6:

```
readbytes("test.tst", SignedByte)
```

```
[42, 17, 1, 3, 5, 7, 127, −6]
```

`Short` interprets two bytes to be one number. Therefore, the eight written bytes are interpreted as four numbers. For example, the first 2 bytes yield $42\,2^8 + 17 = 10769$:

```
readbytes("test.tst", Short)
```

```
[10769, 259, 1287, 32762]
```

With the flag `LittleEndian`, the byte order is reversed. For example, the first 2 bytes now yield $17\,2^8 + 42 = 4394$:

```
readbytes("test.tst", Short, LittleEndian)
```

```
[4394, 769, 1797, 64127]
```

`Word` interprets four bytes to be one number. Therefore, the eight written bytes give two numbers. The first 4 bytes yield $10769\,2^{16} + 259 = 705757443$:

```
readbytes("test.tst", Word)
```

```
[705757443, 84377594]
```

`Double` interprets eight bytes to represent one floating-point number. The interpretation is machine dependent and may be different for you:

```
readbytes("test.tst", Double)
```

$$\left[4.633737352\ 10^{-106}\right]$$

Example 2

We use `readbytes` and `writebytes` to encrypt the file created in the previous example with a simple "Caesar type encoding": Any integer x (a byte) is replaced by $x + 13 \bmod 256$:

```
L := readbytes("test.tst"):
L := map(L, x -> (x + 13 mod 256)):
writebytes("test.tst", L):
```

Knowing the encryption and its key, we can successfully decrypt the file:

```
L := readbytes("test.tst")
```

```
[55, 30, 14, 16, 18, 20, 140, 7]
```

```
map(L, x -> (x - 13 mod 256))
```

```
[42, 17, 1, 3, 5, 7, 127, 250]
```

```
delete L:
```

Example 3

In this example, we use `fopen` to write and read a file in portions:

```
n := fopen("test.tst", Write, Raw):
for i from 1 to 10 do writebytes(n, [i]) end_for:
```

```
fclose(n):
```

Equivalently, we could have written all data in one go:

```
n := fopen("test.tst", Write, Raw):
writebytes(n, [i $ i = 1..10]):
fclose(n):
```

We read the data byte by byte:

```
n := fopen("test.tst", Read, Raw):
readbytes(n, 1), readbytes(n, 1), readbytes(n, 1);
fclose(n):
```

```
[1], [2], [3]
```

The next command reads in portions of 5 bytes each:

```
n := fopen("test.tst", Read, Raw):
readbytes(n, 5), readbytes(n, 5);
fclose(n):
```

```
[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]
```

```
delete n, i:
```

Example 4

An error is raised if the data do not match the specified format. Here, `-5` does not match `Byte`. This format does not include negative numbers:

```
writebytes("test.tst", [42, 17, -5, 7], Byte)
```

```
Error: The argument is invalid. [writebytes]
```

Example 5

Here we demonstrate what happens if the number of bytes in the file does not match a multiple of units of the specified format. Since both `SignedShort` and `Float` consist of an even number of bytes, the trailing 5-th byte corresponding to `11` is ignored:

```
writebytes("test.tst", [42, 17, 7, 9, 11], Byte):
readbytes("test.tst", SignedShort),
readbytes("test.tst", Float)
```

$$[10769, 1801], \left[1.28810279\ 10^{-13}\right]$$

Example 6

Here we show the effects of `BigEndian` and `LittleEndian`:

```
writebytes("test.tst", [129, 255, 145, 171, 191, 253], Byte):
L1 := readbytes("test.tst", Short, BigEndian)
```

$$[33279, 37291, 49149]$$

```
L2 := readbytes("test.tst", Short, LittleEndian)
```

$$[65409, 43921, 64959]$$

We look at the data in a binary representation (see `numlib::g_adic` for details). The effect of using `LittleEndian` instead of `BigEndian` is to exchange the first 8 bits and the last 8 bits of each number:

```
map(L1, numlib::g_adic, 2)
```

$$[[1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1], [1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1], [1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1]]$$

```
map(L2, numlib::g_adic, 2)
```

$$[[1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1], [1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1]]$$

```
delete L1, L2:
```

Example 7

We are writing the elements of a DOM_HFARRAY to a file. All the elements are double values and `writebytes` does not allow to write the elements of the array in another format than `Double`.

```
A:=hfarray(1..2,1..6,
   [   0.2703,   12.8317, -33.1531, 9999.9948, 0.2662,  -14.3421,
    1000.1801,    0.4521, -34.6787,  -67.3549, 0.6818,   13]):
writebytes("test.tst", A):
```

But if we try to write the elements as bytes we will get an error.

```
writebytes("test.tst", A, Byte);
```

```
Error: The argument is invalid. [writebytes]
```

```
delete A:
```

Example 8

Now we are reading data from a file and we are creating a DOM_HFARRAY with the data using the option `ReturnType`.

```
writebytes("test.tst",
  [    0.2703,   12.8317, -33.1531, 9999.9948, 0.2662,  -14.3421,
    1000.1801,    0.4521, -34.6787,  -67.3549, 0.6818,   13], Double):
readbytes("test.tst", ReturnType=[DOM_HFARRAY,2,6]);
readbytes("test.tst", ReturnType=[DOM_HFARRAY,2,3,2]);
```

$$\begin{pmatrix} 0.2703 & 12.8317 & -33.1531 & 9999.9948 & 0.2662 & -14.3421 \\ 1000.1801 & 0.4521 & -34.6787 & -67.3549 & 0.6818 & 13.0 \end{pmatrix}$$

```
hfarray(1..2, 1..3, 1..2, [0.2703, 12.8317, -33.1531, 9999.9948, 0.2662, -\
14.3421, 1000.1801, 0.4521, -34.6787, -67.3549, 0.6818, 13.0])
```

If we try to read more elements, exactly the elements of the array are read.

```
readbytes("test.tst", ReturnType=[DOM_HFARRAY,2,4]);
readbytes("test.tst", 12, ReturnType=[DOM_HFARRAY,2,3]);
```

$$\begin{pmatrix} 0.2703 & 12.8317 & -33.1531 & 9999.9948 \\ 0.2662 & -14.3421 & 1000.1801 & 0.4521 \end{pmatrix}$$

$$\begin{pmatrix} 0.2703 & 12.8317 & -33.1531 \\ 9999.9948 & 0.2662 & -14.3421 \end{pmatrix}$$

If we read just a part of the array, the other elements are initialized with 0.0.

```
readbytes("test.tst", ReturnType=[DOM_HFARRAY,2,7]);
readbytes("test.tst", 4, ReturnType=[DOM_HFARRAY,2,6]);
```

$$\begin{pmatrix} 0.2703 & 12.8317 & -33.1531 & 9999.9948 & 0.2662 & -14.3421 & 1000.1801 \\ 0.4521 & -34.6787 & -67.3549 & 0.6818 & 13.0 & 0.0 & 0.0 \end{pmatrix}$$

$$\begin{pmatrix} 0.2703 & 12.8317 & -33.1531 & 9999.9948 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

If we just try to read all the data from the file using the option `ReturnType` without a dimension for the `DOM_HFARRAY` a one dimensional array of the right size is created.

```
readbytes("test.tst", ReturnType=DOM_HFARRAY)
```

$$[[0.2703, 12.8317, -33.1531, 9999.9948, 0.2662, -14.3421, 1000.1801, 0.4521, -34.6787, -67.3549, 0.6818, 13.0]]$$

Example 9

We write a `DOM_HFARRAY` with complex numbers to a file and try to reconstruct it by reading the data.

```
A := hfarray(1..2, 1..3,
             [[2342.133 + 56*I, -342.56, PI + I],
              [           -3*E, I^2 + I,     13]]);
writebytes("test.tst", A);
fd := fopen("test.tst", Read, Raw):
B := readbytes(fd, ReturnType = [DOM_HFARRAY, 2, 3]);
```

```
C := readbytes(fd, ReturnType = [DOM_HFARRAY, 2, 3]);
bool(A = B + C*I);
flose(fd):
```

$$\begin{pmatrix} 2342.133 + 56.0\,i & -342.56 & 3.141592654 + 1.0\,i \\ -8.154845485 & -1.0 + 1.0\,i & 13.0 \end{pmatrix}$$

$$\begin{pmatrix} 2342.133 & -342.56 & 3.141592654 \\ -8.154845485 & -1.0 & 13.0 \end{pmatrix}$$

$$\begin{pmatrix} 56.0 & 0.0 & 1.0 \\ 0.0 & 1.0 & 0.0 \end{pmatrix}$$

TRUE

```
delete A, B, C, fd:
```

Example 10

Lets assume we have a DOM_HFARRAY with entries which are integer numbers between -32768 and 32767 and we want to write this data as SignedShort to a file. If we try it without the option Force we will get an error, because a floating-point number of type DOM_FLOAT cannot be written as a SignedShort. With the option Force writebytes tries to convert the floating-point number to a signed word and writes it in any case to the file.

```
A:=hfarray( 1..2,1..3, [[234,-32768,1],[32767,-12111,-3]]);
writebytes("test.tst", SignedShort, A):
```

$$\begin{pmatrix} 234.0 & -32768.0 & 1.0 \\ 32767.0 & -12111.0 & -3.0 \end{pmatrix}$$

```
Error: The argument is invalid. [writebytes]
```

```
writebytes("test.tst", SignedShort, Force, A):
l:= readbytes("test.tst", SignedShort);
op(A,i)-l[i] $i=1..6;
```

```
[234, -32768, 1, 32767, -12111, -3]

0.0, 0.0, 0.0, 0.0, 0.0, 0.0

delete A, 1:
```

Parameters

`filename`

The name of a file: a character string

`n`

A file descriptor provided by `fopen`: a positive integer. The file must have been be opened using the `fopen`-flag `Raw`.

`m`

The number of values to be read or written: a positive integer.

`format`

The format of binary data. Permissible values are `Byte`, `SignedByte`, `Short`, `SignedShort`, `Word`, `SignedWord`, `Float`, and `Double`.

Options

`Byte`, `SignedByte`, `Short`, `SignedShort`, `SignedWord`, `Word`, `Double`, `Float`

The format of the binary data. The default format is `Byte`.

A byte is an 8-bit binary number. Therefore, a byte can have 2^8 different values. For `Byte`, these are the integers from 0 to 255. For `SignedByte`, they are the integers from -128 to 127.

With `Byte`, the data are read/written in 8-bit blocks, interpreted as unsigned bytes. When writing, the numbers are checked for being in the range from 0 to 255.

With `SignedByte`, the data are read or written using the 2-complement.

`Byte` is the default format.

A “short” is a 16-bit binary number (2 bytes). Therefore, a “short” can have 2^{16} different values. For `Short`, these are the integers from 0 to 65536. For `SignedShort`, they are the integers from - 32768 to 32767.

The semantics of `Short` or `SignedShort` is analogous to that of `Byte` or `SignedByte`, respectively.

A “word” is a 32-bit binary number (4 bytes). Therefore, a “word” can have 2^{32} different values. For `Word`, these are the integers from 0 to 4294967296. For `SignedWord`, they are the integers from - 2147483648 to 2147483647.

The semantics of `Word` or `SignedWord` is analogous to that of `Byte` or `SignedByte`, respectively.

A “float” is a 32-bit representation of a real number (4 bytes). A “double” is a 64-bit representation of a real number (8 bytes).

Note: Floats and doubles are read/written in the format of the machine/operating system MuPAD is currently running on. Therefore, the results may differ between different platforms.

Binary files containing floating-point numbers are, in general, not portable to other platforms.

See the flags `BigEndian` and `LittleEndian` for details on the byte ordering.

See “Example 1” on page 1-1707 for an overview over the different format options.

`BigEndian`, `LittleEndian`

The byte ordering: either `BigEndian` or `LittleEndian`. The default odering is `BigEndian`.

`BigEndian` and `LittleEndian` specify the order in which the bytes are arranged for `Short`, `SignedShort`, `Word`, `SignedWord`, `Float`, and `Double`.

For all formats, the data are written in 8-bit blocks (bytes). This also includes the formats where a unit is longer than one byte (all formats but `Byte` and `SignedByte`). With `BigEndian`, the bytes with the most significant bits ("high bits") are written first. With `LittleEndian`, the bytes with the least significant bits are written first.

If, for example, `Short` is selected, there are 16 bits that are to be written. If you pass `BigEndian`, first the byte with the bits for 2^{15} to 2^{8} and then the byte with the bits for 2^{7} to 2^{0} are written. If you specify `LittleEndian`, the order of the bytes is reversed.

`BigEndian` and `LittleEndian` have no effect if the formats `Byte` or `SignedByte` are specified.

`BigEndian` is the default byte order.

Cf. "Example 6" on page 1-1710 for the effects of `BigEndian` and `LittleEndian`.

`Force`

Write the binary data in any case even if the numbers does not match the given format.

If the option `Force` is set, data are written in the given format, e.g. `Byte` even if they does not have the right format. E.g. 100.00 is a `DOM_FLOAT` and normally `writebytes` only writes this data if the format is `Float` or `Double`. With the option `Force` the value is written as a `Byte`. Cf. "Example 10" on page 1-1713.

If the given value does not fit the given data format, the written value is not specified. E.g. 53425.00 written as a `Byte` can be 177 which is `53425.00 mod 256` or just 0. But for sure 100.00 is written as 100.

`ReturnType`

Option, specified as `ReturnType = DOM_HFARRAY | DOM_LIST | [DOM_HFARRAY] | [DOM_HFARRAY, dim1, dim2, …]`that sets the type of the return value.

If set to `DOM_LIST`, the return value is a list which contains the read data.

If set to `DOM_HFARRAY`, the return value is a one dimensional array which contains the read data.

If set to [`DOM_HFARRAY`, $\texttt{dim}_1$, $\texttt{dim}_2$, …], the return value is a (multidimensional) array and $\texttt{dim}_1$, $\texttt{dim}_2$, … are positive integers which specifies the size of the dimensions of the array.

Return Values

`readbytes` returns a list of MuPAD numbers (either integers or floating-point numbers) or an array of hardware floats of type `DOM_HFARRAY`. Its type depends on the setting of the option `ReturnType`; `writebytes` returns the void object `null()` of type `DOM_NULL`.

See Also

MuPAD Functions

`fclose` | `FILEPATH` | `finput` | `fname` | `fopen` | `fprint` | `fread` | `ftextinput` | `import::readbitmap` | `import::readdata` | `pathname` | `print` | `protocol` | `read` | `READPATH` | `write` | `writebytes` | `WRITEPATH`

writebytes

Write binary data to a file

Syntax

```
writebytes(filename | n, list | hfarray, <format>, <BigEndian | LittleEndian>, <Force>)
```

Description

`writebytes(file, list)` writes binary data to a file.

`writebytes(file, hfarray)` writes binary data to a file.

`writebytes` enables the user to write arbitrary files and interpret their contents as a sequence of numbers.

This function is particularly useful to work on data provided by or destined for external programs. You can use it, for example, to implement encryption or compression algorithms in MuPAD. Cf. “Example 2” on page 1-1721.

The results of `writebytes` depend on the interpretation of the binary data set by the `format` option. When writing a file, you can interpret it as a stream of `Byte`, `SignedByte`, `Short`, `SignedShort`, `Word`, `SignedWord`, `Float` or `Double`. These are standard formats that are used by many program packages to write data. Cf. “Example 1” on page 1-1720.

Be sure to write the data in the appropriate way. You need to know the format used by the program which is supposed to read the file.

When writing data via `writebytes`, each entry in the list is checked for whether it can be converted to the specified format. If this is not the case, `writebytes` raises an error. Cf. “Example 4” on page 1-1722.

When writing an array of hardware floats of type `DOM_HFARRAY` only `Double` is allowed as the binary format. If no format option is given hardware floats arrays are written as doubles. Cf. “Example 7” on page 1-1724.

If a hardware float array with complex numbers is written to a file, then first the real parts of the elements are written and then the complex parts are written to the file. Because `readbytes` can only read real values, first one have to create the real and then the complex part to reconstruct the complex array. Cf. “Example 9” on page 1-1725.

The file may be specified directly by its name. In this case, `writebytes` creates a new file or overwrites an existing file.

If a file name is specified, `writebytes` opens and closes the file automatically.

If `WRITEPATH` has no value `writebytes` interprets the file name as a pathname relative to the “working directory.”

Note that the meaning of “working directory” depends on the operating system. On Microsoft Windows systems and on Mac OS X systems, the “working directory” is the folder where MuPAD is installed. On UNIX systems, it is the current working directory in which MuPAD was started; when started from a menu or desktop item, this is typically the user's home directory.

Absolute path names are processed by `writebytes`, too.

If a file name is specified, each call to `writebytes` opens the file at the beginning. If the file was opened via `fopen`, subsequent calls of `writebytes` with the corresponding file descriptor start at the point in the file that was reached by the last `writebytes` command.

Hence, if you want to write a file by portions, you must open it with `fopen` and use the returned file descriptor instead of the filename. Cf. “Example 3” on page 1-1722.

Note: If the file is to be opened via `fopen`, be sure to pass the flag `Raw` to `fopen`. Otherwise, `writebytes` raises an error.

If `writebytes` is used with the option `ReturnType` = [`DOM_HFARRAY`, $\texttt{dim}_1$, $\texttt{dim}_2$, …]the return value is a `DOM_HFARRAY` of the appropriate size. Here $\texttt{dim}_1$, $\texttt{dim}_2$, … and positive integers which specifies the size of the dimensions of the array. If the file contains lesser values or the number of values to be read is limited, the not read elements of the array are initialized to 0.0. In other cases exactly the elements of the array are read. Cf. “Example 8” on page 1-1724.

Environment Interactions

The function `readbytes` is sensitive to the environment variable `READPATH`. First, the file is searched in the "working directory." If it cannot be found there, all paths in `READPATH` are searched.

The function `writebytes` is sensitive to the environment variable `WRITEPATH`. If this variable has a value, the file is created in the corresponding directory. Otherwise, the file is created in the "working directory."

Examples

Example 1

In this example, we write a sequence of numbers to the file `test.tst` with the default settings. Then, we load them back in:

```
writebytes("test.tst", [42, 17, 1, 3, 5, 7, 127, 250]):

readbytes("test.tst")
```

```
[42, 17, 1, 3, 5, 7, 127, 250]
```

We now read the above data with some other option: `SignedByte` interprets all values from 0 to 127 exactly as `Byte` does. Higher values x, however, are interpreted as x - 256. For example, 250 - 256 = - 6:

```
readbytes("test.tst", SignedByte)
```

```
[42, 17, 1, 3, 5, 7, 127, −6]
```

`Short` interprets two bytes to be one number. Therefore, the eight written bytes are interpreted as four numbers. For example, the first 2 bytes yield $42\,2^8 + 17 = 10769$:

```
readbytes("test.tst", Short)
```

```
[10769, 259, 1287, 32762]
```

With the flag `LittleEndian`, the byte order is reversed. For example, the first 2 bytes now yield $17\,2^8 + 42 = 4394$:

```
readbytes("test.tst", Short, LittleEndian)
```

```
[4394, 769, 1797, 64127]
```

`Word` interprets four bytes to be one number. Therefore, the eight written bytes give two numbers. The first 4 bytes yield $10769\,2^{16} + 259 = 705757443$:

```
readbytes("test.tst", Word)
```

```
[705757443, 84377594]
```

`Double` interprets eight bytes to represent one floating-point number. The interpretation is machine dependent and may be different for you:

```
readbytes("test.tst", Double)
```

$$\left[4.633737352\ 10^{-106}\right]$$

Example 2

We use `readbytes` and `writebytes` to encrypt the file created in the previous example with a simple "Caesar type encoding": Any integer x (a byte) is replaced by $x + 13 \bmod 256$:

```
L := readbytes("test.tst"):
L := map(L, x -> (x + 13 mod 256)):
writebytes("test.tst", L):
```

Knowing the encryption and its key, we can successfully decrypt the file:

```
L := readbytes("test.tst")
```

```
[55, 30, 14, 16, 18, 20, 140, 7]
```

```
map(L, x -> (x - 13 mod 256))
```

```
[42, 17, 1, 3, 5, 7, 127, 250]
```

```
delete L:
```

Example 3

In this example, we use `fopen` to write and read a file in portions:

```
n := fopen("test.tst", Write, Raw):
for i from 1 to 10 do writebytes(n, [i]) end_for:
fclose(n):
```

Equivalently, we could have written all data in one go:

```
n := fopen("test.tst", Write, Raw):
writebytes(n, [i $ i = 1..10]):
fclose(n):
```

We read the data byte by byte:

```
n := fopen("test.tst", Read, Raw):
readbytes(n, 1), readbytes(n, 1), readbytes(n, 1);
fclose(n):
```

```
[1], [2], [3]
```

The next command reads in portions of 5 bytes each:

```
n := fopen("test.tst", Read, Raw):
readbytes(n, 5), readbytes(n, 5);
fclose(n):
```

```
[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]
```

```
delete n, i:
```

Example 4

An error is raised if the data do not match the specified format. Here, `-5` does not match `Byte`. This format does not include negative numbers:

```
writebytes("test.tst", [42, 17, -5, 7], Byte)
```

```
Error: The argument is invalid. [writebytes]
```

Example 5

Here we demonstrate what happens if the number of bytes in the file does not match a multiple of units of the specified format. Since both `SignedShort` and `Float` consist of an even number of bytes, the trailing 5-th byte corresponding to `11` is ignored:

```
writebytes("test.tst", [42, 17, 7, 9, 11], Byte):
readbytes("test.tst", SignedShort),
readbytes("test.tst", Float)
```

$$[10769, 1801], \left[1.28810279\ 10^{-13}\right]$$

Example 6

Here we show the effects of `BigEndian` and `LittleEndian`:

```
writebytes("test.tst", [129, 255, 145, 171, 191, 253], Byte):
L1 := readbytes("test.tst", Short, BigEndian)
```

$$[33279, 37291, 49149]$$

```
L2 := readbytes("test.tst", Short, LittleEndian)
```

$$[65409, 43921, 64959]$$

We look at the data in a binary representation (see `numlib::g_adic` for details). The effect of using `LittleEndian` instead of `BigEndian` is to exchange the first 8 bits and the last 8 bits of each number:

```
map(L1, numlib::g_adic, 2)
```

$$[[1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1], [1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1], [1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1]]$$

```
map(L2, numlib::g_adic, 2)
```

```
[[1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1],
  [1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1]]
```

```
delete L1, L2:
```

Example 7

We are writing the elements of a DOM_HFARRAY to a file. All the elements are double values and writebytes does not allow to write the elements of the array in another format than Double.

```
A:=hfarray(1..2,1..6,
   [   0.2703,   12.8317, -33.1531, 9999.9948, 0.2662,  -14.3421,
    1000.1801,    0.4521, -34.6787,  -67.3549, 0.6818,   13]):
writebytes("test.tst", A):
```

But if we try to write the elements as bytes we will get an error.

```
writebytes("test.tst", A, Byte);
```

```
Error: The argument is invalid. [writebytes]
```

```
delete A:
```

Example 8

Now we are reading data from a file and we are creating a DOM_HFARRAY with the data using the option ReturnType.

```
writebytes("test.tst",
  [   0.2703,   12.8317, -33.1531, 9999.9948, 0.2662,  -14.3421,
    1000.1801,    0.4521, -34.6787,  -67.3549, 0.6818,   13], Double):
readbytes("test.tst", ReturnType=[DOM_HFARRAY,2,6]);
readbytes("test.tst", ReturnType=[DOM_HFARRAY,2,3,2]);
```

$$\begin{pmatrix} 0.2703 & 12.8317 & -33.1531 & 9999.9948 & 0.2662 & -14.3421 \\ 1000.1801 & 0.4521 & -34.6787 & -67.3549 & 0.6818 & 13.0 \end{pmatrix}$$

```
hfarray(1..2, 1..3, 1..2, [0.2703, 12.8317, -33.1531, 9999.9948, 0.2662, -\
14.3421, 1000.1801, 0.4521, -34.6787, -67.3549, 0.6818, 13.0])
```

If we try to read more elements, exactly the elements of the array are read.

```
readbytes("test.tst", ReturnType=[DOM_HFARRAY,2,4]);
readbytes("test.tst", 12, ReturnType=[DOM_HFARRAY,2,3]);
```

$$\begin{pmatrix} 0.2703 & 12.8317 & -33.1531 & 9999.9948 \\ 0.2662 & -14.3421 & 1000.1801 & 0.4521 \end{pmatrix}$$

$$\begin{pmatrix} 0.2703 & 12.8317 & -33.1531 \\ 9999.9948 & 0.2662 & -14.3421 \end{pmatrix}$$

If we read just a part of the array, the other elements are initialized with 0.0.

```
readbytes("test.tst", ReturnType=[DOM_HFARRAY,2,7]);
readbytes("test.tst", 4, ReturnType=[DOM_HFARRAY,2,6]);
```

$$\begin{pmatrix} 0.2703 & 12.8317 & -33.1531 & 9999.9948 & 0.2662 & -14.3421 & 1000.1801 \\ 0.4521 & -34.6787 & -67.3549 & 0.6818 & 13.0 & 0.0 & 0.0 \end{pmatrix}$$

$$\begin{pmatrix} 0.2703 & 12.8317 & -33.1531 & 9999.9948 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

If we just try to read all the data from the file using the option `ReturnType` without a dimension for the `DOM_HFARRAY` a one dimensional array of the right size is created.

```
readbytes("test.tst", ReturnType=DOM_HFARRAY)
```

$$[[0.2703, 12.8317, -33.1531, 9999.9948, 0.2662, -14.3421, 1000.1801, 0.4521, -34.6787, -67.3549, 0.6818, 13.0]]$$

Example 9

We write a `DOM_HFARRAY` with complex numbers to a file and try to reconstruct it by reading the data.

```
A := hfarray(1..2, 1..3,
             [[2342.133 + 56*I, -342.56, PI + I],
```

```
              [           -3*E, I^2 + I,     13]]);
writebytes("test.tst", A);
fd := fopen("test.tst", Read, Raw):
B := readbytes(fd, ReturnType = [DOM_HFARRAY, 2, 3]);
C := readbytes(fd, ReturnType = [DOM_HFARRAY, 2, 3]);
bool(A = B + C*I);
flose(fd):
```

$$\begin{pmatrix} 2342.133 + 56.0\,i & -342.56 & 3.141592654 + 1.0\,i \\ -8.154845485 & -1.0 + 1.0\,i & 13.0 \end{pmatrix}$$

$$\begin{pmatrix} 2342.133 & -342.56 & 3.141592654 \\ -8.154845485 & -1.0 & 13.0 \end{pmatrix}$$

$$\begin{pmatrix} 56.0 & 0.0 & 1.0 \\ 0.0 & 1.0 & 0.0 \end{pmatrix}$$

TRUE

```
delete A, B, C, fd:
```

Example 10

Lets assume we have a DOM_HFARRAY with entries which are integer numbers between -32768 and 32767 and we want to write this data as SignedShort to a file. If we try it without the option Force we will get an error, because a floating-point number of type DOM_FLOAT cannot be written as a SignedShort. With the option Force writebytes tries to convert the floating-point number to a signed word and writes it in any case to the file.

```
A:=hfarray( 1..2,1..3, [[234,-32768,1],[32767,-12111,-3]]);
writebytes("test.tst", SignedShort, A):
```

$$\begin{pmatrix} 234.0 & -32768.0 & 1.0 \\ 32767.0 & -12111.0 & -3.0 \end{pmatrix}$$

```
Error: The argument is invalid. [writebytes]
```

```
writebytes("test.tst", SignedShort, Force, A):
l:= readbytes("test.tst", SignedShort);
op(A,i)-l[i] $i=1..6;
```

$$[234, -32768, 1, 32767, -12111, -3]$$

$$0.0, 0.0, 0.0, 0.0, 0.0, 0.0$$

```
delete A, l:
```

Parameters

`filename`

The name of a file: a character string

`n`

A file descriptor provided by `fopen`: a positive integer. The file must have been be opened using the `fopen`-flag `Raw`.

`list`

A list of MuPAD numbers that are to be written to the file. The entries must match the specified `format`.

`hfarray`

An array of hardware floats of type `DOM_HFARRAY`.

`format`

The format of binary data. Permissible values are `Byte`, `SignedByte`, `Short`, `SignedShort`, `Word`, `SignedWord`, `Float`, and `Double`.

Options

`Byte`, `SignedByte`, `Short`, `SignedShort`, `SignedWord`, `Word`, `Double`, `Float`

The format of the binary data. The default format is `Byte`.

A byte is an 8-bit binary number. Therefore, a byte can have 2^8 different values. For `Byte`, these are the integers from 0 to 255. For `SignedByte`, they are the integers from -128 to 127.

With `Byte`, the data are read/written in 8-bit blocks, interpreted as unsigned bytes. When writing, the numbers are checked for being in the range from 0 to 255.

With `SignedByte`, the data are read or written using the 2-complement.

`Byte` is the default format.

A "short" is a 16-bit binary number (2 bytes). Therefore, a "short" can have 2^{16} different values. For `Short`, these are the integers from 0 to 65536. For `SignedShort`, they are the integers from - 32768 to 32767.

The semantics of `Short` or `SignedShort` is analogous to that of `Byte` or `SignedByte`, respectively.

A "word" is a 32-bit binary number (4 bytes). Therefore, a "word" can have 2^{32} different values. For `Word`, these are the integers from 0 to 4294967296. For `SignedWord`, they are the integers from - 2147483648 to 2147483647.

The semantics of `Word` or `SignedWord` is analogous to that of `Byte` or `SignedByte`, respectively.

The format of the binary data. The default format is `Byte`.

A "float" is a 32-bit representation of a real number (4 bytes). A "double" is a 64-bit representation of a real number (8 bytes).

Note: Floats and doubles are read/written in the format of the machine/operating system MuPAD is currently running on. Therefore, the results may differ between different platforms.

Binary files containing floating-point numbers are, in general, not portable to other platforms.

See the flags `BigEndian` and `LittleEndian` for details on the byte ordering.

See "Example 1" on page 1-1720 for an overview over the different format options.

BigEndian, LittleEndian

The byte ordering: either `BigEndian` or `LittleEndian`. The default odering is `BigEndian`.

`BigEndian` and `LittleEndian` specify the order in which the bytes are arranged for `Short`, `SignedShort`, `Word`, `SignedWord`, `Float`, and `Double`.

For all formats, the data are written in 8-bit blocks (bytes). This also includes the formats where a unit is longer than one byte (all formats but `Byte` and `SignedByte`). With `BigEndian`, the bytes with the most significant bits ("high bits") are written first. With `LittleEndian`, the bytes with the least significant bits are written first.

If, for example, `Short` is selected, there are 16 bits that are to be written. If you pass `BigEndian`, first the byte with the bits for 2^{15} to 2^{8} and then the byte with the bits for 2^{7} to 2^{0} are written. If you specify `LittleEndian`, the order of the bytes is reversed.

`BigEndian` and `LittleEndian` have no effect if the formats `Byte` or `SignedByte` are specified.

`BigEndian` is the default byte order.

See "Example 6" on page 1-1723 for the effects of `BigEndian` and `LittleEndian`.

Force

Write the binary data in any case even if the numbers does not match the given format.

If the option `Force` is set, data are written in the given format, e.g. `Byte` even if they does not have the right format. E.g. 100.00 is a `DOM_FLOAT` and normally `writebytes` only writes this data if the format is `Float` or `Double`. With the option `Force` the value is written as a `Byte`. Cf. "Example 10" on page 1-1726.

If the given value does not fit the given data format, the written value is not specified. E.g. 53425.00 written as a `Byte` can be 177 which is `53425.00 mod 256` or just 0. But for sure 100.00 is written as 100.

ReturnType

Option, specified as `ReturnType = DOM_HFARRAY | DOM_LIST | [DOM_HFARRAY] | [DOM_HFARRAY, dim1, dim2, …]` that sets the type of the return value.

If set to DOM_LIST, the return value is a list which contains the read data.

If set to DOM_HFARRAY, the return value is a one dimensional array which contains the read data.

If set to [DOM_HFARRAY, dim_1, dim_2, ...], the return value is a (multidimensional) array and dim_1, dim_2, ... are positive integers which specifies the size of the dimensions of the array.

Return Values

readbytes returns a list of MuPAD numbers (either integers or floating-point numbers) or an array of hardware floats of type DOM_HFARRAY. Its type depends on the setting of the option ReturnType; writebytes returns the void object null() of type DOM_NULL.

See Also

MuPAD Functions

fclose | FILEPATH | finput | fname | fopen | fprint | fread | ftextinput | import::readbitmap | import::readdata | pathname | print | protocol | read | readbytes | READPATH | write | WRITEPATH

repeat, until, end_repeat, _repeat

"repeat" loop

Syntax

```
repeat
  body
until condition end_repeat

_repeat(body, condition)
```

Description

`repeat - end_repeat` is a loop that evaluates its body until a specified stopping criterion is satisfied.

In a `repeat` loop, first `body` and then `condition` are evaluated until `condition` evaluates to `TRUE`.

In contrast to the `while` loop, the body of a `repeat` loop is always evaluated at least once.

The body may consist of any number of statements which must be separated either by a colon `:` or a semicolon `;`. Only the last evaluated result inside the body (the return value of the loop) is printed on the screen. Use `print` to see intermediate results.

The Boolean expression `condition` must be reducible to either `TRUE` or `FALSE`. Internally, the condition is evaluated in the lazy evaluation context of the functions `_lazy_and` and `_lazy_or`.

The statements `next` and `break` can be used in `repeat` loops in the same way as in `for` loops.

The keyword `end_repeat` may be replaced by the keyword `end`.

The imperative form `repeat - end_repeat` is equivalent to corresponding call of the function `_repeat`. In most cases, the imperative form should be preferred because it leads to simpler code.

The $-operator is often a more elegant notation for loops.

`_repeat` is a function of the system kernel.

Examples

Example 1

Intermediate results of statements within a `repeat` and `while` loop are not printed to the screen:

```
i := 1:
s := 0:
while i < 3 do
  s := s + i;
  i := i + 1;
end_while
```

```
3
```

Above, only the return value of the loop is displayed. Use `print` to see intermediate results:

```
i := 1:
s := 0:
while i < 3 do
  print("intermediate sum" = s);
  s := s + i;
  i := i + 1;
  s
end_while
```

```
"intermediate sum" = 0

"intermediate sum" = 1

3
```

```
delete i, s:
```

Example 2

A simple example is given, how a `repeat` loop can be expressed via an equivalent `while` loop. For other examples, this may be more complicated and additional initializations of variables may be needed:

```
i := 1:
repeat
  print(i);
  i := i + 1;
until i = 3 end:
```

```
    1

    2
```

```
i := 1:
while i < 3 do
  print(i);
  i := i + 1;
end:
```

```
    1

    2
```

```
delete i:
```

Example 3

The Boolean expression `condition` must evaluate to `TRUE` or `FALSE`:

```
condition := UNKNOWN:
while not condition do
  print(Condition = condition);
  condition := TRUE;
end_while:
```

```
Error: The Boolean 'TRUE' or 'FALSE' is expected. [while]
```

To avoid this error, change the stopping criterion to `condition <> TRUE`:

```
condition := UNKNOWN:
while condition <> TRUE do
  print(Condition = condition);
  condition := TRUE;
end_while:
```

```
Condition = UNKNOWN
```

```
delete condition:
```

Example 4

We demonstrate the correspondence between the functional and the imperative form of the `repeat` and `while` loop, respectively:

```
hold(_repeat((statement1; statement2), condition))
```

```
repeat
  statement1;
  statement2
until condition end_repeat
```

```
hold(_while(condition, (statement1; statement2)))
```

```
while condition do
  statement1;
  statement2
end_while
```

Parameters

`body`

The body of the loop: an arbitrary sequence of statements

`condition`

A Boolean expression

Return Values

Value of the last command executed in the body of the loop. If no command was executed, the value `NIL` is returned. If the body of a `while` loop is not evaluated due to a false condition, the void object of type `DOM_NULL` is returned.

See Also

MuPAD Functions

`$` | `_lazy_and` | `_lazy_or` | `break` | `for` | `next` | `while`

More About

- "Loops"

while, end_while, _while

"while" loop

Syntax

```
while condition do
  body
end_while
```

```
_while(condition, body)
```

Description

`while - end_while` represents a loop that evaluates its body while a specified condition holds true.

In a `while` loop, `condition` is evaluated before the body is executed for the first time. If `condition` evaluates to `TRUE`, the loop is entered and `body` and `condition` are evaluated until `condition` evaluates to `FALSE`.

In contrast to the `while` loop, the body of a `repeat` loop is always evaluated at least once.

The body may consist of any number of statements which must be separated either by a colon `:` or a semicolon `;`. Only the last evaluated result inside the body (the return value of the loop) is printed on the screen. Use `print` to see intermediate results.

The Boolean expression `condition` must be reducible to either `TRUE` or `FALSE`. Internally, the condition is evaluated in the lazy evaluation context of the functions `_lazy_and` and `_lazy_or`.

The statements `next` and `break` can be used in `while` loops in the same way as in `for` loops.

The keyword `end_while` may be replaced by the keyword `end`.

The imperative form `while - end_while` is equivalent to corresponding call of the function `_while`. In most cases, the imperative form should be preferred because it leads to simpler code.

The $-operator is often a more elegant notation for loops.

_while is a function of the system kernel.

Examples

Example 1

Intermediate results of statements within a repeat and while loop are not printed to the screen:

```
i := 1:
s := 0:
while i < 3 do
  s := s + i;
  i := i + 1;
end_while
```

```
     3
```

Above, only the return value of the loop is displayed. Use print to see intermediate results:

```
i := 1:
s := 0:
while i < 3 do
  print("intermediate sum" = s);
  s := s + i;
  i := i + 1;
  s
end_while
```

```
     "intermediate sum" = 0

     "intermediate sum" = 1

     3
```

```
delete i, s:
```

Example 2

A simple example is given, how a `repeat` loop can be expressed via an equivalent `while` loop. For other examples, this may be more complicated and additional initializations of variables may be needed:

```
i := 1:
repeat
  print(i);
  i := i + 1;
until i = 3 end:
```

```
    1
```

```
    2
```

```
i := 1:
while i < 3 do
  print(i);
  i := i + 1;
end:
```

```
    1
```

```
    2
```

```
delete i:
```

Example 3

The Boolean expression `condition` must evaluate to `TRUE` or `FALSE`:

```
condition := UNKNOWN:
while not condition do
  print(Condition = condition);
  condition := TRUE;
end_while:
```

```
Error: The Boolean 'TRUE' or 'FALSE' is expected. [while]
```

To avoid this error, change the stopping criterion to `condition <> TRUE`:

```
condition := UNKNOWN:
while condition <> TRUE do
  print(Condition = condition);
  condition := TRUE;
end_while:
```

```
Condition = UNKNOWN
```

```
delete condition:
```

Example 4

We demonstrate the correspondence between the functional and the imperative form of the `repeat` and `while` loop, respectively:

```
hold(_repeat((statement1; statement2), condition))
```

```
repeat
  statement1;
  statement2
until condition end_repeat
```

```
hold(_while(condition, (statement1; statement2)))
```

```
while condition do
  statement1;
  statement2
end_while
```

Parameters

`body`

The body of the loop: an arbitrary sequence of statements

condition

A Boolean expression

Return Values

Value of the last command executed in the body of the loop. If no command was executed, the value `NIL` is returned. If the body of a `while` loop is not evaluated due to a false condition, the void object of type `DOM_NULL` is returned.

See Also

MuPAD Functions

`$` | `_lazy_and` | `_lazy_or` | `break` | `for` | `next` | `repeat`

More About

- "Loops"

rec

Domain of recurrence equations

Syntax

```
rec(eq, y(n), <cond>)
```

Description

`rec(eq, y(n))` creates an object of type `rec` representing a recurrence equation for the sequence `y(n)`.

The equation `eq` must involve only shifts `y(n + i)` with integer values of `i`; at least one such expression must be present in `eq`. An arithmetical expression`eq` is equivalent to the equation `eq = 0`.

Initial or boundary conditions `cond` must be specified as sets of equations of the form `{y(n0) = y0, y(n1) = y1, ...}` with arithmetical expressions `n0, n1, ...` that must not contain the identifier `n`, and arithmetical expressions `y0, y1, ...` that must not contain the identifier `y`.

The main purpose of the `rec` domain is to provide an environment for overloading the function `solve`. For a recurrence `r` of type `rec`, the call `solve(r)` returns a set representing an affine subspace of the complete solution space. Its only entry is an expression in `n` that may contain free parameters such as `C1`, `C2`, etc. See“Example 1” on page 1-1742, “Example 4” on page 1-1743, and “Example 5” on page 1-1743.

Currently only linear recurrences with coefficients that are rational functions of `n` can be solved. `solve` handles recurrences with constant coefficients, it finds hypergeometric solutions of first order recurrences, and polynomial solutions of higher order recurrences with non-constant coefficients.

`solve` is not always able to find the complete solution space. Cf. “Example 5” on page 1-1743. If `solve` cannot find a solution, then the `solve` call is returned symbolically. For parametric recurrences, the output of `solve` may be a conditionally defined set of type `piecewise`. Cf. “Example 6” on page 1-1743.

Examples

Example 1

The first command defines the homogeneous first order recurrence equation $y(n+1) = \frac{2\,(n+1)\,y(n)}{n}$ for the sequence $y(n)$. It is solved by a call to the `solve` function:

```
rec(y(n + 1) = 2*y(n)*(n + 1)/n, y(n))
```

$$\mathrm{rec}\left(y(n+1) - \frac{2\,y(n)\,(n+1)}{n},\ y(n),\ \varnothing\right)$$

```
solve(%)
```

$$\{2^n\ C1\ n\}$$

Thus, the general solution of the recurrence equation is $y(n) = C_1\, n\, 2^n$, where C_1 is an arbitrary constant.

Example 2

In the next example, the homogeneous first order recurrence $y(n + 1) = 3\,(n + 1)\,y(n)$ with the initial condition $y(0) = 1$ is solved for the unknown sequence $y(n)$:

```
solve(rec(y(n + 1) = 3*(n + 1)*y(n), y(n), {y(0) = 1}))
```

$$\{3^n\,\Gamma(n+1)\}$$

Thus, the solution is $y(n) = 3^n\,\Gamma(n+1) = 3^n\,n!$ for all integers $n \geq 0$ (*gamma* is the gamma function).

Example 3

In the following example, the inhomogeneous second order recurrence $y(n + 2) - 2\,y(n + 1) + y(n) = 2$ is solved for the unknown sequence $y(n)$. The initial conditions $y(0) = -1$ and $y(1) = m$ with some parameter m are taken into account by `solve`:

```
solve(rec(y(n + 2) - 2*y(n + 1) + y(n) = 2, y(n),
          {y(0) = -1, y(1) = m}))
```

$$\{n^2 + m\,n - 1\}$$

Example 4

We compute the general solution of the homogeneous second order recurrence $y(n + 2) + 3\,y(n + 1) + 2\,y(n) = 0$:

```
solve(rec(y(n + 2) + 3*y(n + 1) + 2*y(n), y(n)))
```

$$\{(-1)^n\,C7 + (-2)^n\,C6\}$$

Here, `C6` and `C7` are arbitrary constants.

Example 5

For the following homogeneous third order recurrence with non-constant coefficients, the system only finds the polynomial solutions:

```
solve(rec(n*y(n + 3) = (n + 3)*y(n), y(n)))
```

$$\{C9\,n\}$$

Example 6

The following homogeneous second order recurrence with constant coefficients involves a parameter `a`. The solution set depends on the value of this parameter, and `solve` returns a `piecewise` object:

```
solve(rec(a*y(n + 2) = y(n), y(n)))
```

$$\begin{cases} \{0\} & \text{if } a = 0 \\ \left\{C11\left(\frac{1}{\sqrt{a}}\right)^n + C10\left(-\frac{1}{\sqrt{a}}\right)^n\right\} & \text{if } a \neq 0 \end{cases}$$

Example 7

The following homogeneous second order recurrence with non-constant coefficients involves a parameter `a`. Although it has a polynomial solution for $a = 2$, the system does not recognize this:

```
solve(rec(n*y(n + 2) = (n + a)*y(n), y(n)))
```

```
{0}
```

Parameters

`eq`

An equation or an arithmetical expression

`y`

The unknown function: an identifier

`n`

The index: an identifier

`cond`

A set of initial or boundary conditions

Return Values

Object of type `rec`.

Algorithms

For homogeneous recurrences with constant coefficients, `solve` computes the roots of the characteristic polynomial. If some of them cannot be given in explicit form, i.e., only

by means of `RootOf`, then `solve` does not return a solution. Otherwise, the complete solution space is returned.

For first order homogeneous recurrences with nonconstant coefficients, `solve` returns the complete solution space if the coefficients of the recurrence can be factored into at most quadratic polynomials. Otherwise, `solve` does not return a solution.

For homogeneous recurrences of order at least two with nonconstant coefficients, `solve` finds the complete space of all *polynomial* solutions.

Currently, inhomogeneous recurrences can only be solved if they have a polynomial solution. The previous remarks apply.

For parametric recurrences, the system may not find solutions that are valid only for special values of the parameters. Cf. “Example 7” on page 1-1744.

See Also

MuPAD Functions

`ode` | `solve` | `sum`

rectform

Rectangular form of a complex expression

Syntax

```
rectform(z)
```

Description

`rectform(z)` computes the rectangular form of the complex expression z, i.e., it splits z into $z = \Re(z) + i\,\Im(z)$.

`rectform(z)` tries to split `z` into its real and imaginary part and to return `z` in the form $z = \Re(z) + i\,\Im(z)$.

`rectform` works recursively, i.e., it first tries to split each subexpression of `z` into its real and imaginary part and then tackles `z` as a whole.

Use `Re` and `Im` to extract the real and imaginary parts, respectively, from the result of `rectform`. See “Example 1” on page 1-1747.

`rectform` is more powerful than a direct application of `Re` and `Im` to `z`. However, usually it is much slower. For constant arithmetical expressions, it is therefore recommended to use the functions `Re` and `Im` directly. See “Example 2” on page 1-1748.

The main use of `rectform` is for symbolic expressions, and properties of identifiers are taken into account (see `assume`). An identifier without any property is assumed to be complex valued. See “Example 3” on page 1-1749.

If `z` is an array, a list, or a set, then `rectform` is applied to each entry of `z`.

If `z` is an hfarray, then `rectform` returns `z` unchanged.

If `z` is a polynomial or a series expansion, of type `Series::Puiseux` or `Series::gseries`, then `rectform` is applied to each coefficient of `z`.

See “Example 5” on page 1-1751.

The result `r := rectform(z)` is an element of the domain `rectform`. Such a domain element consists of three operands, satisfying the following equality: `z = op(r, 1) + I*op(r, 2) + op(r, 3)`. The first two operands are real arithmetical expressions, and the third operand is an expression that cannot be split into its real and imaginary part.

Sometimes `rectform` is unable to compute the required decomposition. Then it still tries to return some partial information by extracting as much as possible from the real and imaginary part of `z`. The extracted parts are stored in the first two operands, and the third operand contains the remainder, where no further extraction is possible. In extreme cases, the first two operands may even be zero. “Example 6” on page 1-1752 illustrates some possible cases.

Arithmetical operations with elements of the domain type `rectform` are possible. The result of an arithmetical operation is again an element of this domain (see “Example 4” on page 1-1749).

Most MuPAD functions handling arithmetical expressions (e.g., `expand`, `normal`, `simplify` etc.) can be applied to elements of type `rectform`. They act on each of the three operands individually.

Use `expr` to convert the result of `rectform` into an element of a basic domain. See “Example 4” on page 1-1749.

Environment Interactions

The function is sensitive to properties of identifiers set via `assume`. See “Example 3” on page 1-1749.

Examples

Example 1

The rectangular form of *sin*(*z*) for complex values *z* is:

```
delete z: r := rectform(sin(z))
```

$$\cosh(\Im(z))\,\sin(\Re(z)) + (\cos(\Re(z))\,\sinh(\Im(z)))\,i$$

The real and the imaginary part can be extracted as follows:

```
Re(r), Im(r)
```

$$\cosh(\Im(z))\,\sin(\Re(z)),\ \cos(\Re(z))\,\sinh(\Im(z))$$

The complex conjugate of `r` can be obtained directly:

```
conjugate(r)
```

$$\cosh(\Im(z))\,\sin(\Re(z)) - (\cos(\Re(z))\,\sinh(\Im(z)))\,i$$

Example 2

The real and the imaginary part of a constant arithmetical expression can be determined by the functions `Re` and `Im`, as in the following example:

```
Re(ln(-4)) + I*Im(ln(-4))
```

$$\ln(4) + \pi\,i$$

In fact, they work much faster than `rectform`. However, they fail to compute the real and the imaginary part of arbitrary symbolic expressions, such as for the term $e^{i\,sin(z)}$:

```
delete z: f := exp(I*sin(z)):
Re(f), Im(f)
```

$$\Re\left(e^{\sin(z)\,i}\right),\ \Im\left(e^{\sin(z)\,i}\right)$$

The function `rectform` is more powerful. It is able to split the expression above into its real and imaginary part:

```
r := rectform(f)
```

$$\cos(\cosh(\Im(z))\,\sin(\Re(z)))\,e^{-\cos(\Re(z))\,\sinh(\Im(z))} + \left(\sin(\cosh(\Im(z))\,\sin(\Re(z)))\,e^{-\cos(\Re(z))\,\sinh(\Im(z))}\right)i$$

Now we can extract the real and the imaginary part of `f`:

```
Re(r)
```

$$\cos(\cosh(\Im(z))\,\sin(\Re(z)))\,e^{-\cos(\Re(z))\,\sinh(\Im(z))}$$

```
Im(r)
```

$$\sin(\cosh(\Im(z))\,\sin(\Re(z)))\,e^{-\cos(\Re(z))\,\sinh(\Im(z))}$$

Example 3

Identifiers without properties are considered to be complex variables:

```
delete z: rectform(ln(z))
```

$$\frac{\ln(\Im(z)^2+\Re(z)^2)}{2}+\arg(\Im(z)\,i+\Re(z))\,i$$

However, you can affect the behavior of `rectform` by attaching properties to the identifiers. For example, if z assumes only real negative values, the real and the imaginary part simplify considerably:

```
assume(z < 0): rectform(ln(z))
```

$$\ln(-z)+\pi\,i$$

Example 4

We compute the rectangular form of the complex variable x:

```
delete x: a := rectform(x)
```

$$\Re(x)+\Im(x)\,i$$

Then we do the same for the real variable y:

```
delete y: assume(y, Type::Real): b := rectform(y)
```

$$y$$

```
domtype(a), domtype(b)
```

$$\text{rectform, rectform}$$

We have stored the results, i.e., the elements of domain type `rectform`, in the two identifiers `a` and `b`. We compute the sum of `a` and `b`, which is again of domain type `rectform`, i.e., it is already splitted into its real and imaginary part:

```
c := a + b
```

$$y + \Re(x) + \Im(x)\,\mathrm{i}$$

```
domtype(c)
```

$$\text{rectform}$$

The result of an arithmetical operation between an element of domain type `rectform` and an arbitrary arithmetical expression is of domain type `rectform` as well:

```
delete z: d := a + 2*b + exp(z)
```

$$2\,y + \Re(x) + \cos(\Im(z))\,\mathrm{e}^{\Re(z)} + \left(\Im(x) + \sin(\Im(z))\,\mathrm{e}^{\Re(z)}\right)\mathrm{i}$$

```
domtype(d)
```

$$\text{rectform}$$

Use the function `expr` to convert an element of domain type `rectform` into an element of a basic domain:

```
expr(d)
```

$$2\,y + \Im(x)\,\mathrm{i} + \Re(x) + \cos(\Im(z))\,\mathrm{e}^{\Re(z)} + \sin(\Im(z))\,\mathrm{e}^{\Re(z)}\,\mathrm{i}$$

```
domtype(%)
```

DOM_EXPR

Example 5

`rectform` also works for polynomials and series expansions, namely individually on each coefficient:

```
delete x, y: p := poly(ln(-4) + y*x, [x]):
rectform(p)
```

$$\mathrm{poly}((\Re(y) + \Im(y)\,\mathrm{i})\,x + \ln(4) + \pi\,\mathrm{i},\ [x])$$

Similarly, `rectform` works for lists, sets, or arrays, where it is applied to each individual entry:

```
a := array(1..2, [x, y]):
rectform(a)
```

$$(\ \Re(x) + \Im(x)\,\mathrm{i}\ \ \Re(y) + \Im(y)\,\mathrm{i}\ )$$

hfarrays are returned unchanged:

```
a := hfarray(1..2, [1.0, 2.0]):
rectform(a)
```

$$(\ 1.0\ \ 2.0\ )$$

Note that `rectform` does not work directly for other basic data types. For example, if the input expression is a table of arithmetical expressions, then `rectform` responds with an error message:

```
a := table("1st" = x, "2nd" = y):
rectform(a)
```

```
Error: An arithmetical expression is expected. [rectform::new]
```

Use `map` to apply `rectform` to the operands of such an object:

```
map(a, rectform)
```

"1st"	$\Im(x)\,\mathrm{i} + \Re(x)$
"2nd"	$\Im(y)\,\mathrm{i} + \Re(y)$

Example 6

This example illustrates the meaning of the three operands of an object returned by `rectform`.

We start with the expression $x + sin(y)$, for which `rectform` is able to compute a complete decomposition into real and imaginary part:

```
delete x, y: r := rectform(x + sin(y))
```

$$\Re(x) + \cosh(\Im(y))\,\sin(\Re(y)) + (\Im(x) + \cos(\Re(y))\,\sinh(\Im(y)))\,\mathrm{i}$$

The first two operands of `r` are the real and imaginary part of the expression, and the third operand is 0:

```
op(r)
```

$$\Re(x) + \cosh(\Im(y))\,\sin(\Re(y)),\ \Im(x) + \cos(\Re(y))\,\sinh(\Im(y)),\ 0$$

Next we consider the expression $x + f(y)$, where $f(y)$ represents an unknown function in a complex variable. `rectform` can split x into its real and imaginary part, but fails to do this for the subexpression $f(y)$:

```
delete f: r := rectform(x + f(y))
```

$$\Re(x) + \Im(x)\,\mathrm{i} + f(y)$$

The first two operands of the returned object are the real and the imaginary part of x, and the third operand is the remainder $f(y)$, for which `rectform` was not able to extract any information about its real and imaginary part:

```
op(r)
```

$$\Re(x),\ \Im(x),\ f(y)$$

```
Re(r), Im(r)
```

$$\Re(f(y)) + \Re(x),\ \Im(f(y)) + \Im(x)$$

Sometimes `rectform` is not able to extract any information about the real and imaginary part of the input expression. Then the third operand contains the whole input expression, possibly in a rewritten form, due to the recursive mode of operation of `rectform`. The first two operands are 0. Here is an example:

```
r := rectform(sin(x + f(y)))
```

$$\sin(\Im(x)\,\mathrm{i} + \Re(x) + f(y))$$

```
op(r)
```

$$0,\ 0,\ \sin(\Re(x) + f(y) + \Im(x)\,\mathrm{i})$$

```
Re(r), Im(r)
```

$$\Re(\sin(\Re(x) + f(y) + \Im(x)\,\mathrm{i})),\ \Im(\sin(\Re(x) + f(y) + \Im(x)\,\mathrm{i}))$$

Example 7

Advanced users can extend `rectform` to their own special mathematical functions (see section “Backgrounds” below). To this end, embed your mathematical function into a function environment`f` and implement the behavior of `rectform` for this function as the `"rectform"` slot of the function environment.

If a subexpression of the form `f(u,..)` occurs in `z`, then `rectform` issues the call `f::rectform(u,..)` to the slot routine to determine the rectangular form of `f(u,..)`.

For illustration, we show how this works for the sine function. Of course, the function environment `sin` already has a `"rectform"` slot. We call our function environment `Sin` in order not to overwrite the existing system function `sin`:

```
Sin := funcenv(Sin):
```

```
Sin::rectform := proc(u) // compute rectform(Sin(u))
  local r, a, b;
begin
  // recursively compute rectform of u
  r := rectform(u);

   if op(r, 3) <> 0 then
    // we cannot split Sin(u)
    new(rectform, 0, 0, Sin(u))
  else
    a := op(r, 1); // real part of u
    b := op(r, 2); // imaginary part of u
    new(rectform, Sin(a)*cosh(b), cos(a)*sinh(b), 0)
  end_if
end:

delete z: rectform(Sin(z))
```

$$\mathrm{Sin}(\Re(z))\cosh(\Im(z)) + (\cos(\Re(z))\sinh(\Im(z)))\,i$$

If the `if` condition is true, then `rectform` is unable to split `u` completely into its real and imaginary part. In this case, `Sin::rectform` is unable to split `Sin(u)` into its real and imaginary part and indicates this by storing the whole expression `Sin(u)` in the third operand of the resulting `rectform` object:

```
delete f: rectform(Sin(f(z)))
```

$$\mathrm{Sin}(f(z))$$

```
op(%)
```

$$0,\ 0,\ \mathrm{Sin}(f(z))$$

Parameters

z

An arithmetical expression, a polynomial, a series expansion, an array, an hfarray, a list, or a set

Return Values

Element of the domain `rectform` if `z` is an arithmetical expression, and an object of the same type as `z` otherwise.

Function Calls

Calling an element of `rectform` as a function yields the object itself, regardless of the arguments. The arguments are *not* evaluated.

Operations

You can apply (almost) any function to elements of `rectform` which transforms a complex-valued expression into a complex-valued expression.

For example, you may add or multiply those elements, or apply functions such as `expand` and `diff` to them. The result of such an operation, which is not explicitely overloaded by a method of `rectform` (see below), is an element of `rectform`.

This "automatic overloading" works as follows: Each argument of the operation, which is an element of `rectform`, is converted to an expression using the method `"expr"` (see below). Then, the operation is applied and the result is re-converted to an element of `rectform`.

Use the function `expr` to convert an element of `rectform` to an arithmetical expression (as an element of a kernel domain).

The functions `Re` and `Im` return the real and imaginary part of elements of `rectform`.

Operands

An element *z* of `rectform` consists of three operands:

1. the real part of *z*,
2. the imaginary part of *z*,
3. the part of *z*, for that the real and imaginary part cannot be computed (possibly the integer 0, if there are not such subexpressions).

Algorithms

If a subexpression of the form `f(u,..)` occurs in `z` and `f` is a function environment, then `rectform` attempts to call the slot `"rectform"` of `f` to determine the rectangular form of `f(u,..)`. In this way, you can extend the functionality of `rectform` to your own special mathematical functions.

The slot `"rectform"` is called with the arguments `u,..` of `f`. If the slot routine `f::rectform` is not able to determine the rectangular form of `f(u,..)`, then it should return `new(rectform(0,0,f(u,..)))`. See “Example 7” on page 1-1753. If `f` does not have a slot `"rectform"`, then `rectform` returns the object `new(rectform(0,0,f(u,..)))` for the corresponding subexpression.

Similarly, if an element `d` of a library domain`T` occurs as a subexpression of `z`, then `rectform` attempts to call the slot `"rectform"` of that domain with `d` as argument to compute the rectangular form of `d`.

If the slot routine `T::rectform` is not able to determine the rectangular form of `d`, then it should return `new(rectform(0,0,d))`.

If the domain `T` does not have a slot `"rectform"`, then `rectform` returns the object `new(rectform(0,0,d))` for the corresponding subexpression.

See Also

MuPAD Functions

`abs` | `assume` | `collect` | `combine` | `conjugate` | `expand` | `Im` | `normal` | `radsimp` | `Re` | `rewrite` | `sign` | `simplify`

More About

- “Manipulate Expressions”
- “Choose Simplification Functions”

rectangularPulse

Rectangular pulse function

Syntax

```
rectangularPulse(a, b, x)

rectangularPulse(x)
```

Description

`rectangularPulse(a, b, x)` represents the rectangular function.

`rectangularPulse(x)` is a shortcut for `rectangularPulse(-1/2, 1/2, x)`.

The rectangular function is also called the rectangle function, box function, Pi function, or gate function.

If `a` and `b` are variables or expressions with variables, `rectangularPulse` assumes that `a < b`. If `a` and `b` are numerical values, such that `a > b`, `rectangularPulse` throws an error.

If `a < x < b`, the rectangular pulse function equals 1. If `x = a` or `x = b`, the rectangular pulse function equals 1/2. Otherwise, it equals 0. See “Example 1” on page 1-1758 and “Example 2” on page 1-1758.

If `a = b`, `rectangularPulse` returns 0. See “Example 3” on page 1-1758.

`rectangularPulse(x)` is equivalent to `rectangularPulse(-1/2, 1/2, x)`. See “Example 4” on page 1-1758.

`rectangularPulse` also accepts infinities as its arguments. See “Example 7” on page 1-1760.

`rectangularPulse` and `rectpulse` are equivalent.

Examples

Example 1

Compute the rectangular pulse function for these input arguments:

```
[rectangularPulse(-1, 1, -2), rectangularPulse(-1, 1, -1),
rectangularPulse(-1, 1, 0), rectangularPulse(-1, 1, 1),
rectangularPulse(-1, 1, 2)]
```

$$\left[0, \frac{1}{2}, 1, \frac{1}{2}, 0\right]$$

Example 2

If `a < b`, the rectangular pulse function for `x = a` and `x = b` equals 1/2:

```
assume(a < b);
[rectangularPulse(a, b, a), rectangularPulse(a, b, b)]
```

$$\left[\frac{1}{2}, \frac{1}{2}\right]$$

Example 3

For `a = b`, the rectangular pulse function returns 0:

```
rectangularPulse(a, a, x)
```

$$0$$

Example 4

Use `rectangularPulse` with one input argument as a shortcut for computing `rectangularPulse(-1/2, 1/2, x)`:

```
rectangularPulse(x)
```

$$\text{rectangularPulse}\left(-\frac{1}{2}, \frac{1}{2}, x\right)$$

```
[rectangularPulse(-1), rectangularPulse(-1/2), rectangularPulse(0),
rectangularPulse(1/2), rectangularPulse(1)]
```

$$\left[0, \frac{1}{2}, 1, \frac{1}{2}, 0\right]$$

Example 5

Rewrite the rectangular pulse function in terms of the Heaviside step function:

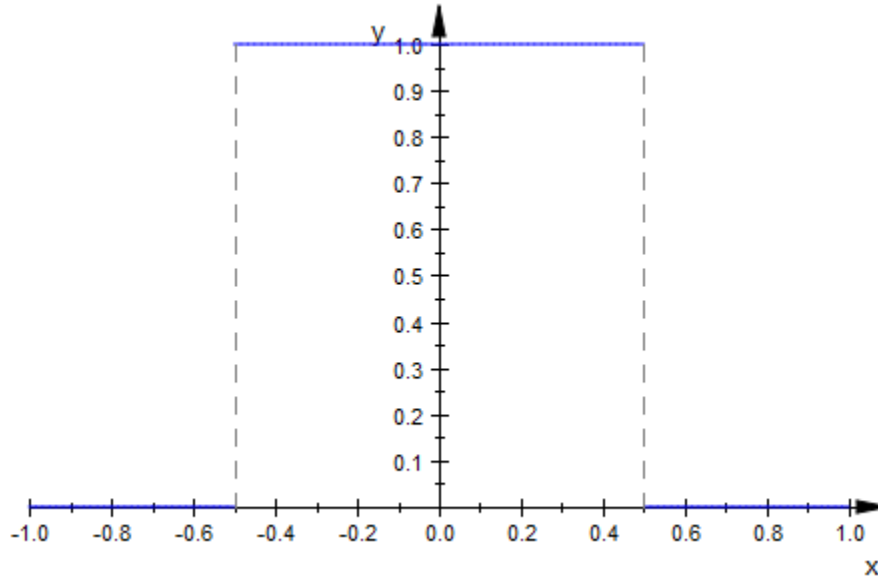
```
rewrite(rectangularPulse(a, b, x), heaviside)
```

$$\text{heaviside}(x-a) - \text{heaviside}(x-b)$$

Example 6

Plot the rectangular pulse function:

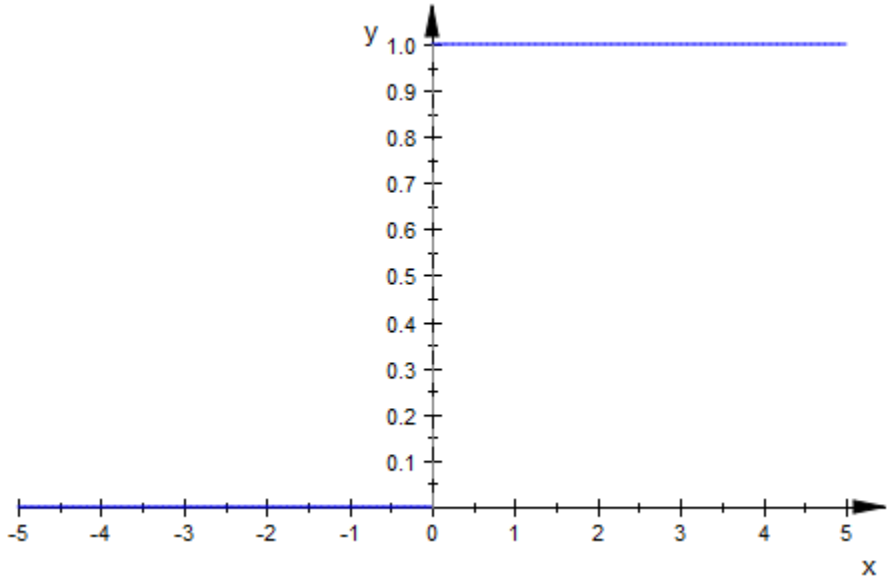
```
plot(rectangularPulse(x), x = -1..1)
```



Example 7

Plot the rectangular pulse function for which the argument b is a positive infinity:

```
plot(rectangularPulse(0, infinity, x))
```



Parameters

a, b, x

Arithmetical expressions.

Return Values

Arithmetical expression.

Overloaded By

x

See Also

MuPAD Functions
`heaviside` | `piecewise` | `triangularPulse`

rectpulse

Rectangular pulse function

Syntax

```
rectpulse(a, b, x)
```

```
rectpulse(x)
```

Description

`rectpulse(a, b, x)` represents the rectangular function.

`rectpulse(x)` is a shortcut for `rectpulse(-1/2, 1/2, x)`.

`rectpulse` and `rectangularPulse` are equivalent. These functions represent the triangular pulse function. For details and examples, see `rectangularPulse`.

Parameters

`a, b, x`

Arithmetical expressions.

Return Values

Arithmetical expression.

Overloaded By

x

See Also

MuPAD Functions

`heaviside` | `piecewise` | `rectangularPulse` | `triangularPulse`

rem

Remainder after division

Syntax

```
rem(a,b)
```

Description

`rem(a,b)` finds the remainder after division. If `b <> 0`, then `rem(a,b) = a - trunc(a/b)*b`. See “Example 1” on page 1-1764 and “Example 2” on page 1-1764.

If `b = 0` or `b = infinity` or `b = -infinity`, then `rem` returns `undefined`.

The `rem` function does not support complex numbers: all values must be real numbers.

Examples

Example 1

Find the remainder after division in case both the dividend and divisor are integers.

Find the modulus after division for these numbers.

```
rem(27, 4), rem(27, -4), rem(-27, 4), rem(-27, -4)
```

```
ans = 3, 3, -3, -3
```

Example 2

Find the remainder after division in case the dividend is a rational number, and the divisor is an integer.

Find the remainder after division for these numbers.

```
rem(22/3, 5), rem(1/2, -7), rem(27/6, -11)
```

$$\frac{7}{3}, \frac{1}{2}, \frac{9}{2}$$

Example 3

Find the remainder after division in case the dividend and divisor are floating-point numbers.

```
rem(2.3, 0.2), rem(-4.5, 1.3), rem(2.7, 1.0)
```

$$0.1, -0.6, 0.7$$

Parameters

a

A real number

b

A real number

Return Values

A number or an arithmetical expression.

Overloaded By

a, b

See Also

MuPAD Domains
`Dom::IntegerMod`

MuPAD Functions

`/` | `div` | `divide` | `frac` | `gcd` | `gcdex` | `igcd` | `igcdex` | `IntMod` | `mod` | `modp` | `mods` | `powermod`

reset

Re-initialize a session

Syntax

```
reset()
```

Description

`reset()` re-initializes a MuPAD session, so that the current session will behave like a freshly started MuPAD session.

`reset` deletes the values of all identifiers and resets the environment variables to their default values. Finally, the initialization files `sysinit.mu` and `userinit.mu` are read again.

`reset` is permitted only at interactive level. Within a procedure, an error occurs.

Examples

Example 1

`reset` deletes the values of all identifiers and resets environment variables to their default values:

```
a := 1: DIGITS := 5: reset(): a, DIGITS
```

$$a, 10$$

Return Values

Void object `null()` of type `DOM_NULL`.

See Also

MuPAD Functions
`delete`

return

Exit a procedure

Syntax

```
return(x)
```

Description

`return(x)` terminates the execution of a procedure and returns `x`.

Usually, MuPAD ends a procedure when all statements of the procedure body were processed. In this case, the return value of the procedure is the result of the last statement that was executed.

Alternatively, the call `return(x)` inside a procedure leads to immediate exit from the procedure: `x` is evaluated and becomes the return value of the procedure. Execution proceeds after the point where the procedure was invoked.

`x` may be an expression sequence, i.e., calls such as `return(x1, x2, ...)` are allowed.

`return()` returns the void object of type `DOM_NULL`.

Note that `return` is a function, not a keyword. A statement such as `return x;` works in the programming language C, but causes a syntax error in MuPAD.

If called outside a procedure, `return(x)` just returns `x`.

Examples

Example 1

This example shows the implementation of a maximum function (which, in contrast to the system function `max`, accepts only two arguments). If `x` is larger than `y`, the value of `x` is returned and the execution of the procedure `mymax` stops. Otherwise, `return(x)` is not called. Consequently, `y` is the last evaluated object defining the return value:

```
mymax := proc(x : Type::Real, y : Type::Real)
begin
   if x > y then
     return(x)
   end_if;
   y
end_proc:

mymax(3, 2), mymax(4, 5)
```

$$3, 5$$

```
delete mymax:
```

Example 2

`return()` returns the void object:

```
f := x -> return(): type(f(anything))
```

$$\text{DOM\_NULL}$$

```
delete f:
```

Example 3

If `return` is called on the interactive level, the evaluated arguments are returned:

```
x := 1: return(x, y)
```

$$1, y$$

```
delete x:
```

Parameters

x

Any MuPAD object

Return Values

X.

See Also

MuPAD Domains
DOM_PROC

MuPAD Functions
-> | proc

revert

Revert polynomials, lists, character strings and tables, invert series expansions

Syntax

```
revert(object)
```

Description

`revert` reverses the ordering of the elements in a list and the ordering of characters in a string, as well as the ordering of the coefficients in a polynomial. For tables, it swaps indices and entries. For a series expansion, it returns the functional inverse.

`revert` is a general function to compute inverses with respect to functional composition, or to reverse the order of operands. This type of functionality may be extended to further types of objects via overloading.

Currently, the MuPAD library provides functionality for strings, polynomials, lists, and tables, where `revert` reverses the order of the elements, coefficients, or characters, respectively. In tables, entries are turned into indices and vice versa. E.g., `revert(table(x = y, 2 = 4))` yields the table `table(y = x, 4 = 2)`. For series expansions, the functional inverse is returned.

For all other types of MuPAD objects that do not overload `revert`, the symbolic expression `revert(object)` is returned.

Examples

Example 1

`revert` operates on lists and character strings:

```
revert([1, 2, 3, 4, 5])
```

$$[5, 4, 3, 2, 1]$$

```
revert("nuf si DAPuM ni gnimmargorP")
```

"Programming in MuPAD is fun"

`revert` operates on series:

```
revert(series(sin(x), x)) = series(arcsin(x), x)
```

$$x + \frac{x^3}{6} + \frac{3\,x^5}{40} + O(x^7) = x + \frac{x^3}{6} + \frac{3\,x^5}{40} + O(x^7)$$

`revert` operates on tables:

```
t := table():  t[x] := 1: t[y] := 2: t[z] := 3:
T := revert(t): T[1], T[2], T[3]
```

x, y, z

Beware: if an entry is stored under several distinct indices, `revert` reduces the number of table operands:

```
revert(table(x = 1, y = 1, z = 3))
```

1	y
3	z

The functional inverse of the expansion of `exp` around `x = 0` is the expansion of the inverse function `ln` around `x = exp(0) = 1`:

```
revert(series(exp(x), x, 3)) = series(ln(x), x = 1, 2)
```

$$x - 1 - \frac{(x-1)^2}{2} + O((x-1)^3) = x - 1 - \frac{(x-1)^2}{2} + O((x-1)^3)$$

```
delete t, T:
```

Example 2

`revert` computes the reverse of a polynomial:

```
revert(poly(x^3 + 2*x + 5))
```

$$\text{poly}(5\,x^3 + 2\,x^2 + 1, [x])$$

The same works for multivariate polynomials, too:

```
revert(poly(x^3 + 2*x*y + 5*x + 6*y + 7))
```

$$\text{poly}(7\,x^3\,y + 6\,x^3 + 5\,x^2\,y + 2\,x^2 + y, [x, y])$$

We could have achieved the same by substituting all indeterminates by their inverses; however, `revert` works faster.

```
numer(evalp(poly(x^3 + 2*x*y + 5*x + 6*y + 7), x = 1/x, y = 1/y))
```

$$y + 5\,x^2\,y + 7\,x^3\,y + 2\,x^2 + 6\,x^3$$

Example 3

For all other types of objects, a symbolic function call is returned:

```
revert(x + y)
```

$$\text{revert}(x + y)$$

The following series expansion is not of type `Series::Puiseux`. Instead, a generalized expansion of type `Series::gseries` is produced. Consequently, `revert` does not compute an inverse:

```
revert(series(exp(-x)/(1 + x), x = infinity, 3))
```

$$\text{revert}\left(\frac{e^{-x}}{x} - \frac{e^{-x}}{x^2} + \frac{e^{-x}}{x^3} + O\left(\frac{e^{-x}}{x^4}\right)\right)$$

Parameters

`object`

A polynomial, a list, a character string, a table, or a series expansion of type `Series::Puiseux`

Return Values

Object of the same type as the input object, or a symbolic call of type `"revert"`.

Overloaded By

object

See Also

MuPAD Functions
`series` | `sort` | `substring`

rewrite

Rewrite an expression

Syntax

```
rewrite(f, target)
```

Description

`rewrite(f, target)` transforms an expression `f` to a mathematically equivalent form, trying to express `f` in terms of the specified target function.

The target indicates the function that is to be used in the desired representation. Symbolic function calls in `f` are replaced by the target function if this is mathematically valid.

With the target `arg`, the function `ln(sign(x))` is rewritten as i $arg(x)$.

With the target `exp`, all trigonometric and hyperbolic functions are rewritten in terms of `exp`. Further, the inverse functions as well as `arg` are rewritten in terms of `ln`.

With the target `sincos`, the functions `tan`, `cot`, `exp`, `sinh`, `cosh`, `tanh`, and `coth` are rewritten in terms of `sin` and `cos`.

With the target `sin`, the same is done as in the case of `sincos`. Additionally, $cos(x)^2$ is rewritten as 1 - $sin(x)^2$. This holds for the target `cos` analogously.

With the target `sinhcosh`, the functions `exp`, `tanh`, `coth`, `sin`, `cos`, `tan`, and `cot` are rewritten in terms of `sinh` and `cosh`. With the targets `sinh` and `cosh`, the same is done, and `cosh(x)^2` is rewritten in terms of `sinh` (or `sinh(x)^2` in terms of `cosh`, respectively.)

With the targets `arcsin`, `arccos`, `arctan`, and `arccot`, the logarithm, all inverse trigonometric functions, and all inverse hyperbolic functions are rewritten in terms of the target function.

With the targets `arcsinh`, `arccosh`, `arctanh`, and `arccoth`, the logarithm, all inverse hyperbolic functions and all inverse trigonometric functions are rewritten in terms of the target function.

With the target `lambertW`, the function `wrightOmega` is rewritten in terms of `lambertW`.

With the target `erf`, the functions `erfc`, `erfi`, and `dawson` are rewritten in terms of `erf`.

With the target `erfc`, the functions `erf`, `erfi`, and `dawson` are rewritten in terms of `erfc`.

With the target `erfi`, the functions `erf`, `erfc`, and `dawson` are rewritten in terms of `erfi`.

With the target `bernoulli`, the function `euler` is rewritten in terms of `bernoulli`.

With the target `diff`, symbolic calls of the differential operator `D` are rewritten in terms of symbolic calls of the function `diff`. E.g., `D(f)(x)` is converted to `diff(f(x), x)`. A univariate expression `D(f)(x)` is rewritten if `x` is an identifier or an indexed identifier. A multivariate expression `D([n1, n2, ...], f)(x1, x2, ...)` is rewritten if `x1`, `x2` are *distinct* identifiers or indexed identifiers. Trying to rewrite a multivariate call `D(f)(x1, x2, ...)` of the univariate dervative `D(f)` raises an error.

With the target `D`, symbolic `diff` calls are rewritten in terms of the differential operator `D`. Derivatives of univariate function calls such as `diff(f(x), x)` are rewritten as `D(f)(x)`. Derivatives of multivariate function calls are expressed via `D([n1, n2, ...], f)`. E.g., `diff(f(x, y), x)` is rewritten as `D([1], f)(x, y)`.

With the target `andor`, the logical operators `xor`, `==>`, and `<=>` are rewritten in terms of `and`, `or`, and `not`.

With the targets `min` and `max`, expressions in `max` and `min` and, for real arguments, `abs` are rewritten in terms of the target function.

The targets `harmonic` and `psi` serve for rewriting symbolic calls of `psi` in terms of `harmonic` and vice versa.

With the target `inverf`, the function `inverfc(x)` is rewritten as `inverf(1 - x)`.

With the target `inverfc`, the function `inverf(x)` is rewritten as `inverfc(1 - x)`.

Examples

Example 1

This example demonstrates the use of `rewrite`:

```
rewrite(D(D(f))(x), diff)
```

$$\frac{\partial^2}{\partial x^2} f(x)$$

```
diff(f(x, x), x) = rewrite(diff(f(x, x), x), D)
```

$$\frac{\partial}{\partial x} f(x, x) = \mathrm{D}_1(f)(x, x) + \mathrm{D}_2(f)(x, x)$$

```
assume(n, Type::PosInt):
rewrite(fact(n), gamma), rewrite(gamma(n), fact);
delete n:
```

$$\Gamma(n+1),\ (n-1)!$$

```
rewrite(sign(x), heaviside), rewrite(heaviside(x), sign);
```

$$2\ \mathrm{heaviside}(x) - 1,\ \frac{\mathrm{sign}(x)}{2} + \frac{1}{2}$$

```
rewrite(heaviside(x), piecewise)
```

$$\begin{cases} \frac{1}{2} & \text{if } x = 0 \\ 1 & \text{if } 0 < x \\ 0 & \text{if } x < 0 \end{cases}$$

Example 2

Trigonometric functions can be rewritten in terms of `exp`, `sin`, `cos` etc.:

```
rewrite(tan(x), exp), rewrite(cot(x), sincos),
rewrite(sin(x), tan)
```

$$-\frac{e^{2xi}i-i}{e^{2xi}+1},\ \frac{\cos(x)}{\sin(x)},\ \frac{2\tan\left(\frac{x}{2}\right)}{\tan\left(\frac{x}{2}\right)^2+1}$$

```
rewrite(arcsinh(x), ln)
```

$$\ln\left(x+\sqrt{x^2+1}\right)$$

Example 3

Inverse trigonometric functions can be rewritten in terms of each other:

```
rewrite(arcsin(x), arctan)
```

$$2\arctan\left(\frac{x}{\sqrt{1-x^2}+1}\right)$$

The following result uses the function `signIm` ("sign of the imaginary part") to make the formula valid throughout the complex plane (apart from the singularities at $x=\pm\sqrt{-1}$):

```
rewrite(arctan(x), arcsin)
```

$$\mathrm{signIm}(x\,i)\left(\frac{\pi}{2}-\arcsin\left(\frac{1}{\sqrt{x^2+1}}\right)\right)$$

Parameters

`f`

An arithmetical or boolean expression

target

The target function to be used in the representation: one of andor, arccos, arccosh, arccot, arccoth, arcsin, arcsinh, arctan, arctanh, arg, bernoulli, cos, cosh, cot, coth, diff, D, erf, erfc, erfi, exp, fact, gamma, harmonic, heaviside, inverf, inverfc, lambertW, ln, max, min, piecewise, psi, sign, sin, sincos, sinh, sinhcosh, tan, or tanh

Return Values

arithmetical expression.

Overloaded By

f

See Also

MuPAD Functions

collect | combine | expand | factor | normal | partfrac | rationalize | rectform | simplify

RootOf

Set of roots of a polynomial

Syntax

```
RootOf(f, x)
```

```
RootOf(f)
```

Description

`RootOf(f, x)` represents the symbolic set of roots of the polynomial $f(x)$ with respect to the indeterminate x.

`RootOf` serves as a symbolic representation of the zero set of a polynomial. Since it is generally impossible to represent the roots of a polynomial in terms of radicals, `RootOf` is often the only possible way to represent the roots symbolically. `RootOf` mainly occurs in the output of `solve` or related functions; see "Example 3" on page 1-1783.

The parameter `f` must be either a polynomial, or an arithmetical expression representing a polynomial in `x`, or an equation `p=q`, where `p` and `q` are arithmetical expressions representing polynomials in `x`. In the latter case, `RootOf` represents the roots of `p-q` with respect to `x`.

The polynomial `f` need not be irreducible or even square-free. If `f` has multiple roots, `RootOf` represents each of the roots with its multiplicity.

If `x` is omitted, then `f` must be an arithmetical expression or polynomial equation containing exactly one indeterminate, and `RootOf` represents the roots with respect to this indeterminate.

`x` need not be an identifier or indexed identifier: it may be any expression that is neither rational nor constant.

If `f` contains only one indeterminate, then you can apply `float` to the `RootOf` object to obtain a set of floating-point approximations for all roots; see "Example 3" on page 1-1783.

Examples

Example 1

Each of the following calls represents the roots of the polynomial x^3 - x^2 with respect to x, i.e., the set {0, 1}:

```
RootOf(x^3 - x^2, x), RootOf(x^3 = x^2, x)
```

$$\text{RootOf}(x^3 - x^2, x), \text{RootOf}(x^3 - x^2, x)$$

```
RootOf(x^3 - x^2), RootOf(x^3 = x^2)
```

$$\text{RootOf}(x^3 - x^2, x), \text{RootOf}(x^3 - x^2, x)$$

```
RootOf(poly(x^3 - x^2, [x]), x)
```

$$\text{RootOf}(x^3 - x^2, x)$$

In general, however, `RootOf` is only used when no explicit symbolic representation of the roots is possible.

Example 2

The first argument of `RootOf` may contain parameters:

```
RootOf(y*x^2 - x + y^2, x)
```

$$\text{RootOf}(x^2\, y - x + y^2, x)$$

The set of roots of a polynomial is treated like an expression. For example, it may be differentiated with respect to a free parameter. The result is the set of derivatives of the roots; it is expressed in terms of `RootOf`, by giving a minimal polynomial:

```
diff(%, y)
```

$$\text{RootOf}(4\, x^2\, y^5 - x^2\, y^2 + 4\, x\, y^3 - x + y^4 + 2\, y, x)$$

For reducible polynomials, the result may be a multiple of the correct minimal polynomial.

Example 3

`solve` returns `RootOf` objects when the roots of a polynomial cannot be expressed in terms of radicals:

```
solve(x^5 + x + 7, x)
```

$$\text{RootOf}(z^5 + z + 7, z)$$

You can apply the function `float` to obtain floating-point approximations of all roots:

```
float(%)
```

$$\{-1.410813851,\ -0.508469409 + 1.368616488\,i,\ -0.508469409 - 1.368616488\,i,\ 1.213876335 + 0.9241881109\,i,\ 1.213876335 - 0.9241881109\,i\}$$

Example 4

The function `sum` is able to compute sums over all roots of a given polynomial:

```
sum(i^2, i = RootOf(x^3 + a*x^2 + b*x + c, x))
```

$$a^2 - 2\,b$$

```
sum(1/(z + i), i = RootOf(x^4 - y*x + 1, x))
```

$$\frac{4\,z^3 + y}{z^4 + y\,z + 1}$$

Example 5

A `RootOf` object represents the set of all roots. One can address the individual roots via indexed calls:

```
RootOf(z^3 - 1, z)[i] $ i = 1..3
```

$$\mathrm{RootOf}(z^3-1,\,z)_1,\ \mathrm{RootOf}(z^3-1,\,z)_2,\ \mathrm{RootOf}(z^3-1,\,z)_3$$

```
float(RootOf(z^3 - 1, z)[i]) $ i = 1..3
```

$$1.0,\ -0.5+0.8660254038\,i,\ -0.5-0.8660254038\,i$$

Parameters

f

A polynomial, an arithmetical expression representing a polynomial in `x`, or a polynomial equation in `x`

x

The indeterminate: typically, an identifier or indexed identifier

Return Values

Symbolic `RootOf` call, i.e., an expression of type `"RootOf"`.

See Also

MuPAD Functions

`isolate` | `numeric::polyroots` | `poly` | `solve`

Rule

Defining equivalence rules for mathematical expressions

Syntax

```
Rule(pattern, replacement, <conditions>)
```

```
Rule(procedure, <condProc>)
```

Description

`Rule` is a data type. Each object of `Rule` – a rule – describes the equivalence between mathematical expressions. The arguments of a rule are two pattern expressions, that are equivalent, and optional some conditions for the validity of the equivalence.

Rule can be applied to any expression, and returns an expression equivalent to the input, or `FAIL`.

Additionally, a rule can consist of a procedure that returns an equivalent expression to a given expression or `FAIL` *without* using the pattern matcher.

Rules created with `Rule` are mainly used to build a rule base for the new `Simplify`. See the documentation of `Simplify` and "Example 8" on page 1-1792 for a real application of `Rule`. "Example 3" on page 1-1788 shows, how to implement rewriting rules via `Rule`.

All other examples are only given to explain the behavior of rules. In practice, single rules and their manual application is unusual.

There are two kinds of rules: Use the library pattern matcher to determine whether the rule is suitable, or use a user defined procedure to analyze a given expression and return an equivalent expression.

`Rule(pattern, replacement, conditions)` defines a rule that describes the equivalence of the expressions `pattern` and `replacement`.

When this rule is applied to a given expression `ex`, the pattern matcher is called with the arguments

```
match(ex, pattern, Cond = conditions)
```

and returns a set of replacements `S:={var = ex_var, ...}` for each variable `var` of `pattern`, and `ex_var` is the corresponding subexpression of `ex` (see `match` for detailed description).

In this case the result of the substitution `subs(replacement, S)` is returned as equivalent expression to `ex`.

The call to `match` can also return `FAIL`, when `ex` doesn't have the same structure as `pattern`. Then the return value of the rule application is `FAIL`, too.

See “Example 1” on page 1-1787 and “Example 2” on page 1-1787.

See `match` for the description of valid conditions.

Alternatively, a rule can consist of a procedure that is called with a given expression, and must return an equivalent expression or `FAIL`. The “pattern matcher” is not called.

`Rule(procedure, condProc)` defines such a rule that returns an equivalent expression to any given input as return value of `procedure` or `FAIL`.

The optional condition `condProc` must be a procedure, too. This procedure is called *before* the procedure that produces equivalent expressions, with a given expression `ex`. When the call `condProc(ex)` returns `TRUE`, then the return value of the call `procedure(ex)` is returned as the result of the application of the rule, otherwise `FAIL`.

With a rule that consists of a procedure, several relations `pattern <=> result` can be expressed. This is mostly more efficient, than using `match` for each equivalence.

See “Example 6” on page 1-1790 and “Example 7” on page 1-1791.

Note: Rules with expressions as arguments must use identifiers that are protected from any assignment. Those identifiers must be of the form `#X`, where `X` can be any valid variable name. All variables that start with `#` in their names are protected by the kernel from any assignment.

Examples

Example 1

The first rule represents the simplification `sin(X)^2 + cos(X)^2 = 1`. The first argument of the rule is the expression `sin(X)^2 + cos(X)^2`. Each expression, which has the same structure, is found by `match`, and the second argument of the rule `1` is returned as result. There are no conditions for the validity of this equivalence. The identifiers used for defining the rule are write protected, because they have names beginning with #:

```
r := Rule(sin(`#X`)^2 + cos(`#X`)^2, 1):
Rule::apply(r, sin(2*x - 1)^2 + cos(2*x - 1)^2)
```

$$1$$

The next expression doesn't have the right form, the application of the rule fails:

```
Rule::apply(r, sin(2*x - 1)^2 + cos(2*x + 1)^2)
```

$$\text{FAIL}$$

Example 2

The next rule represents the addition theorem `sin(X + Y) = sin(X)*cos(Y) + sin(Y)*cos(X)`. The first argument of the rule is the expression `sin(X + Y)`. Each expression that is a call to `sin` with a sum as argument, is identified by `match`, and the sum `sin(X)*cos(Y) + sin(Y)*cos(X)` is returned, where `X` and `Y` are replaced by the corresponding parts of the given expression. There are no conditions for the validity of this equivalence. The second part of the rule is prevented from evaluation with `hold`. The identifiers used for defining the rule are write protected, because they have names beginning with #:

```
r := Rule(sin(`#X` + `#Y`),
          hold(sin(`#X`)*cos(`#Y`) + sin(`#Y`)*cos(`#X`))):
Rule::apply(r, sin(tan(x) + tan(y)))
```

$$\cos(\tan(x))\,\sin(\tan(y)) + \cos(\tan(y))\,\sin(\tan(x))$$

The matcher identifies the difference of two expressions `a` and `b` as the sum `a + -b`, therefore also the following example works:

```
Rule::apply(r, sin(tan(x) - tan(y)))
```

$$\cos(\tan(y))\,\sin(\tan(x)) - \cos(\tan(x))\,\sin(\tan(y))$$

Example 3

We define two rules based on the trigonometric identies $sin(x)^2 = 1 - cos(x)^2$ and $\tan(x)^2 = \frac{1}{\cos(x)^2} - 1$:

```
myrules := [Rule(sin(`#X`)^`#n`, (1 - cos(`#X`)^2)^(`#n`/2),
                          {`#n` -> is(`#n`, Type::Even)}),
            Rule(tan(`#X`)^`#n`, (1/cos(`#X`)^2 - 1)^(`#n`/2),
                                 {`#n` -> is(`#n`, Type::Even)})
            ]:
```

We wish to apply these rules as rewriting rules to various expressions. We forward `Rule::apply` to all subexpressions of an expression via `misc::maprec`. For convenience, an interface function `myrewrite` is implemented that calls `misc::maprec`:

```
myrewrite:= proc(f, rules)
local _rewrite;
begin
   _rewrite:= proc(x)
   local r, tmp;
   begin
      for r in rules do
         tmp:= Rule::apply(r, x);
         if tmp <> FAIL then
            x:= tmp;
         end;
      end;
      return(x)
    end;
    misc::maprec(f, TRUE = _rewrite);
end:
```

Now we can call `myrewrite(f, myrules)` to apply the rewriting rules to an expression `f`:

```
f:= tan(x) + sin(2*x) - tan(y)^2*sin(x + 3)^6 + sin(x)^2 * tan(23)^4:
myrewrite(f, myrules);
```

$$\sin(2x) + \tan(x) + \left(\frac{1}{\cos(y)^2} - 1\right)(\cos(x+3)^2 - 1)^3 - \left(\frac{1}{\cos(23)^2} - 1\right)^2 (\cos(x)^2 - 1)$$

```
delete myrules, myrewrite, f:
```

Example 4

Another rule represents the simplification `sin(X) = 0`, which is only true, when `X` is an integer multiple of `PI`:

```
r := Rule(sin(`#X`), 0, {`#X` -> is(`#X`/PI, Type::Integer)}):
Rule::apply(r, sin(2*x*PI))
```

FAIL

In the last call, the argument of `sin` doesn't have the necessary property, so the application of the rule fails.

After an assumption to `x`, the expression has the right form:

```
assume(x, Type::Integer):
Rule::apply(r, sin(2*x*PI))
```

0

The next application of the rule checks a constant expression:

```
Rule::apply(r, sin(2*PI))
```

FAIL

Why `FAIL`? The problem is, `sin` simplifies the constant input `2*PI` to `0` itself, so the rule gets `0` as input. However, `0` doesn't have the necessary form, so `FAIL` is returned.

Example 5

Another rule represents the simplification `ln(neg^even*r) = even*ln(-neg) + ln(r)`, which is only true, when `neg` is negative and `even` is an even number:

```
r := Rule(ln(`#Neg`^`#Even`*`#X`),
          `#Even`*ln(-`#Neg`) + ln(`#X`),
          {`#Neg` -> is(`#Neg`, Type::Negative) = TRUE,
           `#Even` -> is(`#Even`, Type::Even) = TRUE}):
delete e, n, x:
Rule::apply(r, ln(n^e*x))
```

$$\ln(n^e\,x)$$

The rule application fails, because the variables doesn't have the necessary properties.

With an assumption `n` should be a negative variable and `e` should be even:

```
assume(n < 0): assume(e, Type::Even):
Rule::apply(r, ln(n^e*x))
```

$$\ln(x) + e\,\ln(-n)$$

Example 6

This rule represents the application of `rewrite` to an expression with the target `exp`, when the expression has subexpressions of type `"sin"` or `"cos"`. The first argument of the rule is a procedure that calls `rewrite` with any expression and target `exp` and returns an expression equivalent to the input (because `rewrite` does it). The second argument is a procedure that checks, whether `sin` or `cos` is contained in the input expression:

```
r := Rule(X -> rewrite(X, exp), X -> has(X, sin) or has(X, cos)):
Rule::apply(r, sin(2*x - 1)^2 + cos(2*x - 1)^2)
```

$$\left(\frac{e^{-2xi+i}}{2} + \frac{e^{2xi-i}}{2}\right)^2 + \left(\frac{e^{-2xi+i}\,i}{2} - \frac{e^{2xi-i}\,i}{2}\right)^2$$

The next expression doesn't have `sin` or `cos`, so the application of the rule fails:

```
Rule::apply(r, tan(2*I*x))
```

FAIL

Example 7

This rule represents the application of `rewrite` to an expression with several targets. The first argument of the rule is a procedure that applies `rewrite` to the given expression, with a target depending on the input. This rule doesn't have a condition procedure:

```
rewProc :=
  proc(ex)
  begin
    rewrite(ex, (if has(ex, exp) then
                   tan
                 elif has(ex, sin) or has(ex, cos) then
                   cot
                 elif has(ex, tan) or has(ex, cot) then
                   sincos
                 else
                   exp
                 end_if))
  end_proc:
r := Rule(rewProc):
Rule::apply(r, exp(2*x))
```

$$-\frac{\tan(x\,i)+i}{\tan(x\,i)-i}$$

The rule is applied again to the last result:

```
Rule::apply(r, %)
```

$$-\frac{\frac{\sin(x\,i)}{\cos(x\,i)}+i}{\frac{\sin(x\,i)}{\cos(x\,i)}-i}$$

The last result should be simplified back to the first expression:

```
Simplify(%)
```

$$e^{2x}$$

Example 8

The new `Simplify` uses a rule base for applying a lot of rewriting rules for finding the simplest form of any given expression.

We want to rewrite only some powers and assume that all used variables are real (without using properties).

The list `PowerRules` consists of several rules. The procedure `powerRules` returns all this rules in a list.

Because of better readability, the names of the used identifiers are short and not protected names:

```
PowerRules :=
  [Rule(A^m*A^n, hold(A^(m + n))),
   Rule(A^m/A^n, hold(A^(m - n))),
   Rule(A^n*B^n, hold((A*B)^n)),
   Rule(A^n/B^n, hold((A/B)^n)),
   Rule(A^n/B^n, hold((B/A)^-n)),
   Rule((A^m)^n, hold(A^(m*n)))]:
powerRules := proc()
              begin
                PowerRules
              end_proc:
```

`Simplify` is called with the option `SelectRules`, and expects a procedure that returns a list of rules, applicable to a given expression. In this case, all of the rules are returned in every case.

`Simplify` applies all rules to a given expression and also to rewritten results, and tries to find the easiest form of the expression with respect to the default valuation procedure `Simplify::complexity`.

Because of the argument `SelectRules = powerRules`, only the given rules are used by `Simplify`:

```
Simplify(T^(1/2)*(R*T)^(-1/2), SelectRules = powerRules)
```

$$\frac{1}{\sqrt{R}}$$

Other expressions cannot be simplified with the same rule base:

```
Simplify(sin(x)^2 + cos(x)^2, SelectRules = powerRules)
```

$$\cos(x)^2 + \sin(x)^2$$

```
delete r, x, powerRules, PowerRules:
```

Parameters

`pattern`

A MuPAD expression; all identifiers are used as pattern variables for the pattern matcher

`replacement`

A MuPAD expression with the same identifiers, as `pattern`; the replacement expression should be protected from evaluation with the function `hold`

`conditions`

A set (of type `DOM_SET`) of procedures and expressions in the pattern variables, or the empty set (see `match` and option `Cond`)

`procedure`

A MuPAD procedure that is called with an expression and must return an equivalent expression or `FAIL`

`condProc`

A procedure that is called with an expression before `procedure`, and must return `TRUE`, when `procedure` should be called with the expression, otherwise `FAIL` is returned immediately

See Also

MuPAD Functions
`match` | `Simplify`

save, _save

Save the state of an identifier

Syntax

```
save x1, x2, …

_save(x1, x2, …)
```

Description

In a procedure, the statement "`save x;`" saves the state of the global identifier `x`.

The `save` statement saves the states of identifiers—i.e., their values and properties — during the execution of procedures. The original state of the identifiers is restored when procedure execution is finished. This holds even when an error occurs.

The `save` statement is to be used only inside the body of a procedure. It cannot be called on the interactive level.

The arguments of the `save` statement are evaluated as usual. In the statement '`save x;`', the symbol `x` must evaluate to an identifier `y`, say. It is the state of the identifier `y` that is saved.

The `save` statement is very similar to the `save` declaration for procedures. The main difference to the declaration is that, in order to make the declaration, one has to know the names of the identifiers to be saved in advance. The `save` statement allows to save identifiers which are known only at runtime.

The `save` statement is usually used in order to temporarily change the properties of an identifier, for example by calling the function `assume`. Eventually, the original properties of the identifiers are restored even if an error occurs.

The statement '`save x1, x2, ...;`' is equivalent to the function call `_save(x1, x2, ...)`.

Examples

Example 1

First, we define a property for the identifier y:

```
assume(y < 0)
```

The properties of the identifier stored in x are changed temporarily during the execution of the following procedure p:

```
p := proc(x : DOM_IDENT)
begin
    save x;
    assume(x > 0);
    is(x > 0)
end_proc:
```

From the procedure's result, we see that the properties of y were changed during the execution of p:

```
p(y)
```

```
    TRUE
```

However, the original properties were restored after exiting p. The identifier y has its original properties:

```
is(y > 0), is(y < 0)
```

```
    FALSE, TRUE
```

The restoration of the original properties is guaranteed even if some error occurs inside the procedure. The following procedure q raises an error after changing the identifier given by x:

```
q := proc(x : DOM_IDENT)
begin
    save x;
    assume(x > 0);
    error("some error")
```

```
end_proc:
q(y)

Error: some error [q]
```

Nevertheless, the original assumptions about `y` are restored:

```
is(y > 0), is(y < 0)

    FALSE, TRUE

unassume(y): delete p, q:
```

Parameters

x_1, x_2, …

Symbols evaluating to identifiers

Return Values

Void object of type `DOM_NULL`.

See Also

MuPAD Functions
`proc`

select

Select operands

Syntax

```
select(object, f, <p1, p2, …>)
```

Description

`select(object, f)` returns a copy of the object with all operands removed that do not satisfy a criterion defined by the procedure `f`.

`select` is a fast and handy function for picking out elements of lists, sets, tables etc. that satisfy a criterion set by the procedure `f`.

The function `f` must return a value that can be evaluated to one of the Boolean values `TRUE`, `FALSE`, or `UNKNOWN`. It may either return one of these values directly, or it may return an equation or an inequality that can be simplified to one of these values by the function `bool`.

Internally, the function `f` is applied to all operands `x` of the input object via the call `f(x, p1, p2, ...)`. If the result is not `TRUE`, this operand is removed. The original object is not modified in this process.

The output object is of the same type as the input object, i.e., a list yields a list, a set yields a set etc.

An input object that is an expression sequence is not flattened. Cf. “Example 2” on page 1-1800.

Also “atomic” objects such as numbers or identifiers can be passed to `select` as first argument. Such objects are handled like sequences with a single operand.

Examples

Example 1

`select` handles lists and sets. In the first example, we select all true statements from a list of logical statements. The result is again a list:

```
select([1 = 1, 1 = 2, 2 = 1, 2 = 2], bool)
```

$$[1 = 1, 2 = 2]$$

In the following example, we extract the subset of all elements that are recognized as zero by `iszero`:

```
select({0, 1, x, 0.0, 4*x}, iszero)
```

$$\{0.0, 0\}$$

`select` also works on tables:

```
T:= table(1 = "y", 2 = "n", 3 = "n", 4 = "y", 5 = "y"):
select(T, has, "y")
```

1	"y"
4	"y"
5	"y"

The following expression is a sum, i.e., an expression of type `"_plus"`. We extract the sum of all terms that do not contain `x`:

```
select(x^5 + 2*x + y - 4, _not@has, x)
```

$$y - 4$$

We extract all factors containing `x` from the following product. The result is a product with exactly one factor, and therefore, is not of the syntactical type `"_mult"`:

```
select(11*x^2*y*(1 - y), has, x)
```

$$x^2$$

```
delete T:
```

Example 2

`select` works for expression sequences:

```
select((1, -4, 3, 0, -5, -2), testtype, Type::Negative)
```

$$-4, -5, -2$$

The $ command generates such expression sequences:

```
select(i $ i = 1..20, isprime)
```

$$2, 3, 5, 7, 11, 13, 17, 19$$

Atomic objects are treated as expression sequences of length one:

```
select(5, isprime)
```

$$5$$

The following result is the void object `null()` of type `DOM_NULL`:

```
domtype(select(6, isprime))
```

$$\text{DOM\_NULL}$$

Example 3

It is possible to pass an “anonymous procedure” to `select`. This allows to perform more complex actions with one call. In the following example, the command `anames(All)` returns a set of all identifiers that have a value in the current MuPAD session. The `select` statement extracts all identifiers beginning with the letter `"h"`:

```
select(anames(All), x -> expr2text(x)[1] = "h")
```

$$\{\text{has, harmonic, hastype, heaviside, help, hessian, }{}_{h}\text{farray, history, hold, htranspose, hull,}$$
$$\text{hypergeom}\}$$

Parameters

`object`

A list, a set, a table, an expression sequence, or an expression of type `DOM_EXPR`

`f`

A procedure returning a Boolean value

`p1, p2, …`

Any MuPAD objects accepted by `f` as additional parameters

Return Values

Object of the same type as the input object.

Overloaded By

object

See Also

MuPAD Functions
`map` | `op` | `split` | `zip`

series

Compute a generalized series expansion

Syntax

```
series(f, x, <order>, <Left | Right | Real | Undirected>, <NoWarning>, <UseGseries>)
```

```
series(f, x = x0, <order>, <Left | Right | Real | Undirected>, <options>)
```

Description

`series(f, x = x0)` computes the first terms of a series expansion of `f` with respect to the variable `x` around the point x_0.

`series` tries to compute either the Taylor series, the Laurent series, the Puiseux series, or a generalized series expansion of `f` around `x = x0`. See `Series::gseries` for details on generalized series expansions.

The mathematical type of the series returned by `series` can be queried using the type expression `Type::Series`.

If `series` cannot compute a series expansion of `f`, a symbolic function call is returned. This is an expression of type `"series"`. Cf. “Example 11” on page 1-1812.

Mathematically, the expansion computed by `series` is valid in some neighborhood of the expansion point in the complex plane. Usually, this is an open disc centered at x_0. However, if the expansion point is a branch point, then the returned expansion may not approximate the function `f` for values of `x` close to the branch cut. Cf. “Example 12” on page 1-1813.

Using the options `Left` or `Right`, one can compute directed expansions that are valid along the real axis. With the option `Real`, a two-sided expansion along the real axis is computed. See “Example 5” on page 1-1807 and “Example 6” on page 1-1807.

If x_0 is `infinity` or `-infinity`, then a directed series expansion along the real axis from the left to the positive real infinity or from the right to the negative real infinity, respectively, is computed. If x_0 is `complexInfinity` and `dir` is not specified or

`Undirected`, then an undirected series expansion around the complex infinity, i.e., the north pole of the Riemann sphere, is computed. Specifying `x0= infinity` is equivalent to `x0= complexInfinity` and `dir = Left`. Similarly, `x0= -infinity` is equivalent to `x0= complexInfinity` and `dir = Right`. Cf. "Example 7" on page 1-1808.

Such a series expansion is computed as follows: The series variable `x` in `f` is replaced by $x = \frac{1}{u}$ (or $x = -\frac{1}{u}$ for `x0= -infinity`). Then, a series expansion of `f` around $u = 0$ is computed. Finally, $u = \frac{1}{x}$ (or $u = -\frac{1}{x}$, respectively) is substituted in the result.

Mathematically, the result of such a series expansion is a series in $\frac{1}{x}$. However, it may happen that the coefficients of the returned series depend on the series variable. See the corresponding paragraph below.

The number of requested terms for the expansion is the argument `order` if specified. Otherwise, the value of the environment variable `ORDER` is used. One can change the default value 6 by assigning a new value to `ORDER`.

The number of terms is counted from the lowest degree term on for finite expansion points, and from the highest degree term on for expansions around infinity, i.e., "`order`" has to be regarded as a "relative truncation order".

`series` implements a limited amount of precision management to circumvent cancellation. If the number of terms of the computed expansion is less than `order`, a second series computation with a higher value of `order` is tried automatically, and the result of the latter is returned.

Note: Nevertheless, the actual number of terms in the resulting series expansion may differ from the requested number of terms. See "Example 13" on page 1-1814 and "Example 15" on page 1-1816.

Taylor/Laurent/Puiseux expansions (all of domain type `Series::Puiseux`) can be restricted easily to an absolute order term by adding an appropriate `O` term. Cf. "Example 14" on page 1-1815.

Expansions of symbolic integrals can be computed. Cf. "Example 16" on page 1-1817.

If `f` is an expression of type `RootOf`, then `series` returns the set of all non-zero series solutions of the corresponding algebraic equation. Cf. "Example 9" on page 1-1810.

If `order` has the value `infinity`, then the system tries to convert the first argument into a formal infinite series, i.e., it computes a general formula for the n-th coefficient in the Taylor expansion of `f`. The result is an inactive symbolic sum or a polynomial expression. Cf. “Example 10” on page 1-1812.

If `series` returns a series expansion of domain type `Series::Puiseux`, it may happen that the “coefficients” of the returned series depend on the series variable. In this case, the expansion is not a proper Puiseux series in the mathematical sense. See “Example 7” on page 1-1808 and “Example 8” on page 1-1809. However, if the series variable is x and the expansion point is x_0, then the following is valid for each coefficient function $c(x)$ and every positive ε: $c(x)\,(x - x_0)^{\varepsilon}$ converges to zero and $\frac{c(x)}{(x-x_0)^{\varepsilon}}$ is unbounded when x approaches x_0. Similarly, if the expansion point is *infinity*, then, for every positive ε, $\frac{c(x)}{x^{\varepsilon}}$ converges to zero and $c(x)\,x^{\varepsilon}$ is unbounded when x approaches *infinity*.

The function returns a domain object that can be manipulated by the standard arithmetical operations. Moreover, the following methods are available: `ldegree` returns the exponent of the leading term; `Series::Puiseux::order` returns the exponent of the error term; `expr` converts to an arithmetical expression, removing the error term; `coeff(s, n)` returns the coefficient of the term of `s` with exponent `n`; `lcoeff` returns the leading coefficient; `revert` computes the inverse with respect to composition; `diff` and `int` differentiate and integrate a series expansion, respectively; `map` applies a function to all coefficients. See the help pages for `Series::Puiseux` and `Series::gseries` for further details.

Note: `series` works on a symbolic level and should not be called with arguments containing floating point arguments.

Environment Interactions

The function is sensitive to the environment variable `ORDER`, which determines the default number of terms in series computations.

Examples

Example 1

We compute a series expansion of $sin(x)$ around $x = 0$. The result is a Taylor series:

```
s := series(sin(x), x)
```

$$x - \frac{x^3}{6} + \frac{x^5}{120} + O(x^7)$$

Syntactically, the result is an object of domain type `Series::Puiseux`:

```
domtype(s)
```

Series::Puiseux

The mathematical type of the series expansion can be queried using the type expression `Type::Series`:

```
testtype(s, Type::Series(Taylor))
```

TRUE

Various system functions are overloaded to operate on series objects. E.g., the function `coeff` can be used to extract the coefficients of a series expansion:

```
coeff(s, 5)
```

$$\frac{1}{120}$$

The standard arithmetical operators can be used to add or multiply series expansions:

```
s + 2*s, s*s
```

$$3\,x - \frac{x^3}{2} + \frac{x^5}{40} + O(x^7),\ x^2 - \frac{x^4}{3} + \frac{2\,x^6}{45} + O(x^8)$$

```
delete s:
```

Example 2

This example computes the composition of `s` by itself, i.e. the series expansion of $sin(sin(x))$.

```
s := series(sin(x), x): s @ s = series(sin(sin(x)), x)
```

$$x - \frac{x^3}{3} + \frac{x^5}{10} + O(x^7) = x - \frac{x^3}{3} + \frac{x^5}{10} + O(x^7)$$

```
delete s:
```

Example 3

We compute the series expansion of the tangent function around the origin in two ways:

```
series(sin(x), x) / series(cos(x), x) = series(tan(x), x)
```

$$x + \frac{x^3}{3} + \frac{2\,x^5}{15} + O(x^7) = x + \frac{x^3}{3} + \frac{2\,x^5}{15} + O(x^7)$$

```
bool(%)
```

TRUE

Example 4

We compute a Laurent expansion around the point 1:

```
s := series(1/(x^2 - 1), x = 1)
```

$$\frac{1}{2\,(x-1)} - \frac{1}{4} + \frac{x-1}{8} - \frac{(x-1)^2}{16} + \frac{(x-1)^3}{32} - \frac{(x-1)^4}{64} + O((x-1)^5)$$

```
testtype(s, Type::Series(Taylor)),
testtype(s, Type::Series(Laurent))
```

FALSE, TRUE

Example 5

Without an optional argument or with the option `Undirected`, the `sign` function is not expanded:

```
series(x*sign(x^2 + x), x) =
series(x*sign(x^2 + x), x, Undirected)
```

$$x\,\mathrm{sign}(x^2+x)+O(x^7) = x\,\mathrm{sign}(x^2+x)+O(x^7)$$

Some simplification occurs if one requests an expansion that is valid along the real axis only:

```
series(x*sign(x^2 + x), x, Real)
```

$$x\,\mathrm{sign}(x)+O(x^7)$$

The `sign` vanishes from the result if one requests a one-sided expansion along the real axis:

```
series(x*sign(x^2 + x), x, Right),
series(x*sign(x^2 + x), x, Left)
```

$$x+O(x^7),\ -x+O(x^7)$$

Example 6

In MuPAD, the `heaviside` function is defined only on the real axis. Thus an undirected expansion in the complex plane does not make sense:

```
series(x*heaviside(x + 1), x)
```

```
Warning: Cannot find an undirected series expansion. Try the 'Left', 'Right', or 'Real
```

$$\mathrm{series}(x\,\mathrm{heaviside}(x+1),\ x)$$

After specifying corresponding options, the system computes an expansion along the real axis:

```
series(x*heaviside(x + 1), x, Real),
series(x*heaviside(x + 1), x, Right)
```

$$x + O(x^7), x + O(x^7)$$

At the point `I` in the complex plane, the function `heaviside` is not defined, and neither is a series expansion:

```
series(heaviside(x), x = I, Real)
```

$$\text{series}(\text{heaviside}(x), x = \text{i}, \text{Real})$$

Example 7

We compute series expansions around infinity:

```
s1 := series((x + 1)/(x - 1), x = complexInfinity)
```

$$1 + \frac{2}{x} + \frac{2}{x^2} + \frac{2}{x^3} + \frac{2}{x^4} + \frac{2}{x^5} + O\left(\frac{1}{x^6}\right)$$

```
s2 := series(psi(x), x = infinity)
```

$$\ln(x) - \frac{1}{2\,x} - \frac{1}{12\,x^2} + \frac{1}{120\,x^4} + O\left(\frac{1}{x^6}\right)$$

```
domtype(s1), domtype(s2)
```

$$\text{Series::Puiseux}, \text{Series::Puiseux}$$

Although both expansions are of domain type `Series::Puiseux`, `s2` is not a Puiseux series in the mathematical sense, since the first term contains a logarithm, which has an essential singularity at infinity:

```
testtype(s1, Type::Series(Puiseux)),
testtype(s2, Type::Series(Puiseux))
```

TRUE, FALSE

```
coeff(s2)
```

$$\ln(x),\ -\frac{1}{2},\ -\frac{1}{12},\ 0,\ \frac{1}{120}$$

The following expansion is of domain type `Series::gseries`:

```
s3 := series(exp(x)/(1 - x), x = infinity, 4)
```

$$-\frac{e^x}{x}-\frac{e^x}{x^2}-\frac{e^x}{x^3}-\frac{e^x}{x^4}+O\left(\frac{e^x}{x^5}\right)$$

```
domtype(s3)
```

Series::gseries

```
delete s1, s2, s3:
```

Example 8

Oscillating but bounded functions may appear in the "coefficients" of a series expansion as well:

```
s := series(sin(x + 1/x), x = infinity)
```

$$\sin(x)+\frac{\cos(x)}{x}-\frac{\sin(x)}{2\,x^2}-\frac{\cos(x)}{6\,x^3}+\frac{\sin(x)}{24\,x^4}+\frac{\cos(x)}{120\,x^5}+O\left(\frac{1}{x^6}\right)$$

```
domtype(s), testtype(s, Type::Series(Puiseux))
```

Series::Puiseux, FALSE

```
coeff(s, -1)
```

$\cos(x)$

Example 9

The algebraic equation $y^5 - y - x = 0$ cannot be resolved in terms of radicals:

```
solve(y^5 - y - x, y)
```

$$\text{RootOf}(z^5 - z - x,\ z)$$

However, `series` can compute all series solutions of this equation around $x = 0$:

```
series(%, x = 0)
```

$$\left\{-x - x^5 + O(x^7),\ -1 + \frac{x}{4} + \sigma_5 + \sigma_1 + \sigma_4 + \frac{x^5}{4} + O(x^6),\ 1 + \frac{x}{4} - \sigma_5 + \sigma_1 - \sigma_4 + \frac{x^5}{4} + O(x^6),\right.$$
$$\left.-i + \frac{x}{4} - \sigma_3 - \sigma_1 + \sigma_2 + \frac{x^5}{4} + O(x^6),\ i + \frac{x}{4} + \sigma_3 - \sigma_1 - \sigma_2 + \frac{x^5}{4} + O(x^6)\right\}$$

where

$$\sigma_1 = \frac{5x^3}{32}$$

$$\sigma_2 = \frac{385x^4 i}{2048}$$

$$\sigma_3 = \frac{5x^2 i}{32}$$

$$\sigma_4 = \frac{385x^4}{2048}$$

$$\sigma_5 = \frac{5x^2}{32}$$

It may happen that the series solutions themselves are expressed in terms of `RootOf`s:

```
series(RootOf(y^5 -(x + 2*x^2)*y^3 - x^3*y^2
              + (x^3 + x^4)*y + x^4 + x^5, y), x)
```

$$\left\{-\sqrt{x}-\sigma_6+\sigma_5-\sigma_4+\sigma_2-\sigma_1+\sigma_3,\ \sqrt{x}+\sigma_6-\sigma_5+\sigma_4-\sigma_2+\sigma_1+\sigma_3\right\}$$
$$\cup\left\{x\,z+O(x^7)\ \middle|\ z\in \mathrm{RootOf}(z1^3-z1-1,\ z1)\right\}$$

where

$$\sigma_1=\frac{7x^{11/2}}{256}$$

$$\sigma_2=\frac{5x^{9/2}}{128}$$

$$\sigma_3=O\left(x^{13/2}\right)$$

$$\sigma_4=\frac{x^{7/2}}{16}$$

$$\sigma_5=\frac{x^{5/2}}{8}$$

$$\sigma_6=\frac{x^{3/2}}{2}$$

The coefficients of the algebraic equation are allowed to be transcendental. They are internally converted into Puiseux series by `series`:

```
series(RootOf(y^3 - y - exp(x - 1) + 1, y), x = 1, 4)
```

$$\left\{-(x-1)-\frac{(x-1)^2}{2}-\frac{7(x-1)^3}{6}+\sigma_1,\ 1+\frac{x-1}{2}-\frac{(x-1)^2}{8}+\frac{5(x-1)^3}{24}+\sigma_1,\right.$$
$$\left.-1+\frac{x-1}{2}+\frac{5(x-1)^2}{8}+\frac{23(x-1)^3}{24}+\sigma_1\right\}$$

where

$$\sigma_1 = O((x-1)^4)$$

An error occurs if some coefficient cannot be expanded into a Puiseux series:

```
series(RootOf(y^3 - y - exp(x), y), x = infinity)
```

```
Error: Cannot expand the coefficients of 'RootOf(y^3 - y - exp(1/x), y)' into a series
```

Example 10

In this example, we compute a formula for the n-th coefficient a_n in the Taylor expansion of the function $e^{-x} = \sum_{n \geq 0} a_n x^n$ around zero, by specifying `infinity` as `order`. The result is a symbolic sum:

```
series(exp(-x), x, infinity)
```

$$\left(\sum_{k=1}^{\infty} \frac{(-1)^k x^k}{k\,\Gamma(k)}\right) + 1$$

If the input is a polynomial expression, then so is the output:

```
series(x^5 - 1, x = 1, infinity)
```

$$5(x-1) + 10(x-1)^2 + 10(x-1)^3 + 5(x-1)^4 + (x-1)^5$$

Example 11

No asymptotic expansion exists for the Bessel J function of unspecified index, and `series` returns a symbolic function call:

```
series(besselJ(k, x), x=infinity)
```

$$\text{series}\left(J_k(x), x = \infty\right)$$

```
domtype(%), type(%)
```

$$\text{DOM\_EXPR}, \text{"series"}$$

Example 12

The branch cut of the logarithm and the square root is the negative real axis. For a series expansion on the branch cut, `series` uses the function `signIm` to return an expansion that is valid in an open disc around the expansion point:

```
series(ln(x), x = -1, 3)
```

$$\pi\ \text{signIm}(x)\ i - (x+1) - \frac{(x+1)^2}{2} - \frac{(x+1)^3}{3} + O((x+1)^4)$$

```
series(sqrt(x), x = -1, 3)
```

$$(-1)^{\frac{\text{signIm}(x)}{2}} - \frac{(-1)^{\frac{\text{signIm}(x)}{2}}(x+1)}{2} - \frac{(-1)^{\frac{\text{signIm}(x)}{2}}(x+1)^2}{8} + O((x+1)^3)$$

The situation is more intricate when the expansion point is a branch point. The following expansion of the function `arcsin(x + 1)` is valid in an open disc around the branch point 0:

```
series(arcsin(x + 1), x, 4)
```

$$\frac{\pi}{2} - \sqrt{2}\sqrt{-x} - \frac{\sqrt{2}(-x)^{3/2}}{12} - \frac{3\sqrt{2}(-x)^{5/2}}{160} - \frac{5\sqrt{2}(-x)^{7/2}}{896} + O\left(x^{9/2}\right)$$

However, the expansion of `f = ln(x + I*x^3)` around the branch point 0 that is returned by `series` does not approximate `f` for values of `x` that are close to the negative real axis:

```
f := ln(x + I*x^3);
```

```
g := series(f, x, 4);
```

$$\ln(x^3\,\mathrm{i}+x)$$

$$\ln(x)+x^2\,\mathrm{i}+O(x^4)$$

```
DIGITS := 20:
float(subs([f, expr(g)], x = -0.01 + 0.0000001*I));
delete DIGITS:
```

$$[-4.6051701789380914\,16-3.1415026535903362385\,\mathrm{i},\ -4.6051701839380913\,68+3.1416826535897835718\,\mathrm{i}]$$

The situation is similar for algebraic branch points:

```
f := sqrt(x + I*x^3);
g := series(f, x, 4);
```

$$\sqrt{x^3\,\mathrm{i}+x}$$

$$\sqrt{x}+\frac{x^{5/2}\,\mathrm{i}}{2}+O\left(x^{9/2}\right)$$

```
DIGITS := 20:
float(subs([f, expr(g)], x = -0.01 + 0.0000001*I));
delete DIGITS:
```

$$[0.00000449999998719375007\,25-0.10000000025124999\,91\,\mathrm{i},\ -0.00000449999999906875+0.10000000012625\,\mathrm{i}]$$

```
delete f, g:
```

Example 13

The first six terms, including zeroes, of the following two series expansions agree:

```
series(sin(tan(x)), x, 12);
series(tan(sin(x)), x, 12);
```

$$x + \frac{x^3}{6} - \frac{x^5}{40} - \frac{55\,x^7}{1008} - \frac{143\,x^9}{3456} - \frac{968167\,x^{11}}{39916800} + O(x^{13})$$

$$x + \frac{x^3}{6} - \frac{x^5}{40} - \frac{107\,x^7}{5040} - \frac{73\,x^9}{24192} + \frac{41897\,x^{11}}{39916800} + O(x^{13})$$

If we want to compute the series expansion of the difference `sin(tan(x)) - tan(sin(x))`, cancellation happens and produces too few terms in the result. `series` detects this automatically and performs a second series computation with increased precision:

```
series(sin(tan(x)) - tan(sin(x)), x, 6)
```

$$-\frac{x^7}{30} - \frac{29\,x^9}{756} - \frac{1913\,x^{11}}{75600} + O(x^{13})$$

It may nevertheless happen that the result has too few terms; cf. “Example 15” on page 1-1816.

If rational exponents occur in the series expansion, then it may even happen that the result has more than the number of terms requested by the third argument:

```
series(x^2*exp(x) + x*sqrt(sin(x)), x, 3)
```

$$x^{3/2} + x^2 + x^3 - \frac{x^{7/2}}{12} + \frac{x^4}{2} + O(x^5)$$

Example 14

`series`'s control of the order term is based on the concept of `relative order', counting the number of terms beginning with the lowest order that is present in the expansion. An `absolute order' control can be achieved by simply adding an appropriate order term to restrict a result returned by `series`:

```
series(exp(x) + x*sqrt(sin(x)), x, 7)
```

$$1 + x + x^{3/2} + \frac{x^2}{2} + \frac{x^3}{6} - \frac{x^{7/2}}{12} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^{11/2}}{1440} + \frac{x^6}{720} + O(x^7)$$

```
series(exp(x) + x*sqrt(sin(x)), x, 7) + O(x^4)
```

$$1 + x + x^{3/2} + \frac{x^2}{2} + \frac{x^3}{6} - \frac{x^{7/2}}{12} + O(x^4)$$

Note, however, that the series must have enough terms for the added order term to have any effect:

```
series(exp(x^2), x, 4)
```

$$1 + x^2 + \frac{x^4}{2} + O(x^6)$$

```
series(exp(x^2), x, 4) + O(x^8)
```

$$1 + x^2 + \frac{x^4}{2} + O(x^6)$$

Example 15

If the specified order for the expansion is too small to compute the reciprocal (due to cancellation), `series` returns a symbolic call:

```
series(exp(x), x, 4)
```

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + O(x^4)$$

```
series(1/(exp(x) - 1 - x - x^2/2 - x^3/6), x, 2)
```

$$\text{series}\left(-\frac{1}{x - e^x + \frac{x^2}{2} + \frac{x^3}{6} + 1}, x, 2\right)$$

After increasing the order, an expansion is computed, but possibly with fewer terms:

```
series(1/(exp(x) - 1 - x - x^2/2 - x^3/6), x, 3);
series(1/(exp(x) - 1 - x - x^2/2 - x^3/6), x, 4)
```

$$\frac{24}{x^4} - \frac{24}{5\,x^3} + O\left(\frac{1}{x^2}\right)$$

$$\frac{24}{x^4} - \frac{24}{5\,x^3} + \frac{4}{25\,x^2} + \frac{12}{875\,x} + O(1)$$

Example 16

`series` and `int` support each other. On the one hand, series expansions can be integrated:

```
int(series(1/(2 - x), x), x = 0..1)
```

$$\int_0^1 O(x^6)\,\mathrm{d}x + \frac{1327}{1920}$$

On the other hand, `series` knows how to handle symbolic integrals:

```
int(x^x, x)
```

$$\int x^x\,\mathrm{d}x$$

```
series(%, x = 0, 3)
```

$$x + x^2\left(\frac{\ln(x)}{2} - \frac{1}{4}\right) + x^3\left(\frac{\ln(x)^2}{6} - \frac{\ln(x)}{9} + \frac{1}{27}\right) + O(x^4)$$

```
int(exp(-x*sin(t)), t = 0.. x)
```

$$\int_0^x \mathrm{e}^{-x\sin(t)}\,\mathrm{d}t$$

```
series(%, x = 0)
```

$$x - \frac{x^3}{2} + \frac{5\,x^5}{24} + O(x^7)$$

```
int(cos((x*t^2 + x^2*t))^(1/3), t = 0..2)
```

$$\int_0^2 \cos(t^2 x + t x^2)^{1/3}\, \mathrm{d}t$$

```
series(%, x)
```

$$2 - \frac{16 x^2}{15} - \frac{4 x^3}{3} - \frac{100 x^4}{81} - \frac{16 x^5}{9} + O(x^6)$$

Example 17

Users can extend the power of `series` by implementing `series` attributes (slots) for their own special mathematical functions.

We illustrate how to write such a series attribute, using the case of the exponential function. (Of course, this function already has a `series` attribute in MuPAD, which you can inspect via `expose(exp::series)`.) In order not to overwrite the already existing attribute, we work on a copy of the exponential function called `Exp`.

The `series` attribute must be a procedure with four arguments. This procedure is called whenever a series expansion of `Exp` with an arbitrary argument is to be computed. The first argument is the argument of `Exp` in the `series` call. The second argument is the series variable; the expansion point is always the origin 0; other expansion points are internally moved to the origin by a change of variables. The third and the fourth argument are identical with the `order` and the `dir` argument of `series`, respectively.

For example, the command `series(Exp(x^2 + 2), x, 5)` is internally converted into the call `Exp::series(x^2 + x, x, 5, Undirected)`. Here is an example of a `series` attribute for `Exp`.

```
// The series attribute for Exp. It handles the call
// series(Exp(f), x = 0, order, dir)
ExpSeries := proc(f, x, order, dir)
  local t, x0, s, r, i;
begin
  // Expand the argument into a series.
  t := series(f, x, order, dir);
```

```
  // Determine the order k of the lowest term in t, so that
  // t = c*x^k + higher order terms, for some non-zero c.
  k := ldegree(t);

  if k = FAIL then
    // t consists only of an error term O(..)
    return(FAIL);

  elif k < 0 then
    // This corresponds to an expansion of exp around infinity,
    // which does not exist for the exponential
    // function, since it has an essential singularity. Thus we
    // return FAIL, which makes series return unevaluatedly. For
    // other special functions, you may add an asymptotic
    // expansion here.
    return(FAIL);

  else // k >= 0
    // This corresponds to an expansion of exp around a
    // finite point x0. We write t = x0 + y, where all
    // terms in y have positive order, use the
    // formula exp(x0 + y) = exp(x0)*exp(y) and compute
    // the series expansion of exp(y) as the functional
    // composition of the Taylor series of exp(x) around
    // x = 0 with t - x0. If your special function has
    // any finite singularities, then they should be
    // treated here.
    x0 := coeff(t, x, 0);
    s := Series::Puiseux::create(1, 0, order,
           [1/i! $ i = 0..(order - 1)], x, 0, dir);
    return(Series::Puiseux::scalmult(s @ (t - x0), Exp(x0), 0))
  end_if
end_proc:
```

This special function must be embedded in a function environment. The following command defines `Exp` as a function environment and lets the system function `exp` do the evaluation. The `subs` command applied on the result achieves that `Exp` with symbolic arguments is returned as `Exp` and not as `exp`.

```
Exp := funcenv(x -> subs(exp(x), hold(exp)=hold(Exp))):
Exp(1), Exp(-1.0), Exp(x^2 + x)
```

$$\mathrm{Exp}(1),\ 0.3678794412,\ \mathrm{Exp}(x^2 + x)$$

`series` can already handle this "new" function, but it can only compute a Taylor expansion with symbolic derivatives:

```
ORDER := 3: series(Exp(x), x = 0)
```

$$1 + x\,\mathrm{Exp}'(0) + \frac{x^2\,\mathrm{Exp}''(0)}{2} + O(x^3)$$

One can define the `series` attribute of `Exp` by assigning the procedure above to its `series` slot:

```
Exp::series := ExpSeries:
```

Now we can test the new attribute:

```
series(Exp(x^2 + x), x = 0) = series(exp(x^2 + x), x = 0)
```

$$1 + x + \frac{3\,x^2}{2} + O(x^3) = 1 + x + \frac{3\,x^2}{2} + O(x^3)$$

```
series(Exp(x^2 + x), x = 2) = series(exp(x^2 + x), x = 2)
```

$$\mathrm{Exp}(6) + 5\,\mathrm{Exp}(6)\,(x-2) + \frac{27\,\mathrm{Exp}(6)\,(x-2)^2}{2} + O((x-2)^3) =$$

$$e^6 + 5\,e^6\,(x-2) + \frac{27\,e^6\,(x-2)^2}{2} + O((x-2)^3)$$

```
series(Exp(x^2 + x), x = 0, 1)
```

$$1 + O(x)$$

```
series(Exp(x^2 + x), x = infinity)
```

$$\mathrm{series}(\mathrm{Exp}(x^2 + x), x = \infty)$$

Another possibility to obtain series expansions of user-defined functions is to define the `diff` attribute of the corresponding function environment. This is used by `series` to compute a Taylor expansion when no `series` attribute exists. However, this only works

when a Taylor expansion exists, whilst a `series` attribute can handle more general types of series expansions as well.

```
delete ExpSeries, Exp:
```

Parameters

`f`

An arithmetical expression representing a function in `x`

`x`

An identifier

x_0

The expansion point: an arithmetical expression. If not specified, the default expansion point 0 is used.

`order`

The number of terms to be computed: a nonnegative integer or `infinity`. The default order is given by the environment variable `ORDER` (default value 6).

Options

`Left`, `Real`, `Right`, `Undirected`

If no expansion exists that is valid in the complex plane, this argument can be used to request expansions that only need to be valid along the real line. The default is `Undirected`.

`NoWarning`

Supresses warning messages printed during the series computation. This can be useful if `series` is called within user-defined procedures.

`UseGseries`

Option, specified as `UseGseries = b`

Use `Series::gseries` to compute the series. `b` must be `TRUE` or `FALSE`. Default is `TRUE`. Even if this option is set to `TRUE`, computing a Puiseux expansion will be attempted first.

Return Values

If `order` is a nonnegative integer, then `series` returns either an object of the domain type `Series::Puiseux` or `Series::gseries`, an expression of type `"series"`, or, if `f` is a `RootOf` expression, a set of type `Type::Set`. If `order = infinity`, then `series` returns an arithmetical expression.

Overloaded By

f

See Also

MuPAD Functions

`asympt` | `limit` | `mtaylor` | `O` | `ORDER` | `RootOf` | `Series::gseries` | `Series::Puiseux` | `solve` | `taylor` | `Type::Series`

Si

Sine integral function

Syntax

```
Si(x)
```

Description

`Si(x)` represents the sine integral $\int_0^x \frac{\sin(t)}{t}\,dt$.

If `x` is a floating-point number, then `Si(x)` returns numerical values. The special values $Si(0) = 0$, $\mathrm{Si}(\infty) = \frac{\pi}{2}$, $\mathrm{Si}(-\infty) = -\frac{\pi}{2}$ are implemented. For all other arguments, `Si` returns symbolic function calls.

The reflection rule $Si(x) = -Si(-x)$ is used if the argument is a negative integer or a negative rational number. It is also used if the argument is a symbolic product involving such a factor. Cf. "Example 2" on page 1-1824.

The float attribute of `Si` is a kernel function, i.e., floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
Si(0), Si(1), Si(sqrt(2)), Si(x + 1), Si(infinity)
```

$$0,\ \mathrm{Si}(1),\ \mathrm{Si}(\sqrt{2}),\ \mathrm{Si}(x+1),\ \frac{\pi}{2}$$

```
Ssi(0), Ssi(1), Ssi(sqrt(2)), Ssi(x + 1), Ssi(infinity)
```

$$-\frac{\pi}{2},\ \mathrm{Ssi}(1),\ \mathrm{Ssi}(\sqrt{2}),\ \mathrm{Ssi}(x+1),\ 0$$

```
Shi(0), Shi(1), Shi(sqrt(2)), Shi(x + 1), Shi(infinity)
```

$$0,\ \mathrm{Shi}(1),\ \mathrm{Shi}(\sqrt{2}),\ \mathrm{Shi}(x+1),\ \infty$$

Floating point values are computed for floating-point arguments:

```
Si(-5.0), Si(1.0), Si(2.0 + 10.0*I)
```

$$-1.549931245,\ 0.9460830704,\ 1187.409493 - 242.5252717\,i$$

```
Ssi(-5.0), Ssi(1.0), Ssi(2.0 + 10.0*I)
```

$$-3.120727572,\ -0.6247132564,\ 1185.838696 - 242.5252717\,i$$

```
Shi(-5.0), Shi(1.0), Shi(2.0 + 10.0*I)
```

$$-20.09321183,\ 1.057250875,\ -0.225644485 + 1.861300836\,i$$

Example 2

The reflection rule *Si*(- *x*) = - *Si*(*x*), *Ssi*(- *x*) = - *Ssi*(*x*) - π, *Shi*(- *x*) = - *Shi*(*x*) is implemented for negative real numbers and products involving such numbers:

```
Si(-3), Si(-3/7), Si(-sqrt(2)), Si(-x/7), Si(-0.3*x)
```

$$-\mathrm{Si}(3),\ -\mathrm{Si}\left(\frac{3}{7}\right),\ -\mathrm{Si}(\sqrt{2}),\ -\mathrm{Si}\left(\frac{x}{7}\right),\ -\mathrm{Si}(0.3\,x)$$

```
Ssi(-3), Ssi(-3/7), Ssi(-sqrt(2)), Ssi(-x/7), Ssi(-0.3*x)
```

$$-\pi - \mathrm{Ssi}(3),\ -\pi - \mathrm{Ssi}\left(\frac{3}{7}\right),\ -\pi - \mathrm{Ssi}(\sqrt{2}),\ -\pi - \mathrm{Ssi}\left(\frac{x}{7}\right),\ -\mathrm{Ssi}(0.3\,x) - 3.141592654$$

```
Shi(-3), Shi(-3/7), Shi(-sqrt(2)), Shi(-x/7), Shi(-0.3*x)
```

$$-\mathrm{Shi}(3),\ -\mathrm{Shi}\left(\frac{3}{7}\right),\ -\mathrm{Shi}(\sqrt{2}),\ -\mathrm{Shi}\left(\frac{x}{7}\right),\ -\mathrm{Shi}(0.3\,x)$$

No such “normalization” occurs for complex numbers or arguments that are not products:

```
Si(- 3 - I), Si(3 + I), Si(x - 1), Si(1 - x)
```

$$\mathrm{Si}(-3-i),\ \mathrm{Si}(3+i),\ \mathrm{Si}(x-1),\ \mathrm{Si}(1-x)$$

```
Ssi(- 3 - I), Ssi(3 + I), Ssi(x - 1), Ssi(1 - x)
```

$$\mathrm{Ssi}(-3-i),\ \mathrm{Ssi}(3+i),\ \mathrm{Ssi}(x-1),\ \mathrm{Ssi}(1-x)$$

```
Shi(- 3 - I), Shi(3 + I), Shi(x - 1), Shi(1 - x)
```

$$\mathrm{Shi}(-3-i),\ \mathrm{Shi}(3+i),\ \mathrm{Shi}(x-1),\ \mathrm{Shi}(1-x)$$

Example 3

The functions `diff`, `float`, `limit`, and `series` handle expressions involving `Si` and `Shi`:

```
diff(Si(x), x, x, x), float(ln(3 + Si(sqrt(PI))))
```

$$\frac{2\sin(x)}{x^3} - \frac{\sin(x)}{x} - \frac{2\cos(x)}{x^2},\ 1.502020149$$

```
diff(Ssi(x), x, x, x), float(ln(3 + Ssi(sqrt(PI))))
```

$$\frac{2\sin(x)}{x^3} - \frac{\sin(x)}{x} - \frac{2\cos(x)}{x^2},\ 1.071568401$$

```
diff(Shi(x), x, x, x), float(ln(3 + Shi(sqrt(PI))))
```

$$\frac{\sinh(x)}{x} - \frac{2\cosh(x)}{x^2} + \frac{2\sinh(x)}{x^3},\ 1.631702794$$

```
limit(Si(2*x^2/(1+x)), x = infinity)
```

$$\frac{\pi}{2}$$

```
limit(Ssi(2*x^2/(1+x)), x = infinity)
```

$$0$$

```
limit(Shi(2*I*x^2/(1+x)), x = infinity)
```

$$\frac{\pi\, i}{2}$$

```
series(Si(x), x = 0)
```

$$x - \frac{x^3}{18} + \frac{x^5}{600} + O(x^7)$$

```
series(Ssi(x), x = 0)
```

$$-\frac{\pi}{2} + x - \frac{x^3}{18} + \frac{x^5}{600} + O(x^6)$$

```
series(Shi(x), x = 0)
```

$$x + \frac{x^3}{18} + \frac{x^5}{600} + O(x^7)$$

```
series(Si(x), x = infinity, 3)
```

$$\frac{\pi}{2} - \frac{\cos(x)}{x} - \frac{\sin(x)}{x^2} + O\left(\frac{1}{x^3}\right)$$

```
series(Ssi(x), x = infinity, 3)
```

$$-\frac{\cos(x)}{x}-\frac{\sin(x)}{x^2}+\frac{2\cos(x)}{x^3}+O\left(\frac{1}{x^4}\right)$$

```
series(Shi(I*x), x = infinity, 3)
```

$$\frac{\pi\,i}{2}-\frac{\cos(x)\,i}{x}-\frac{\sin(x)\,i}{x^2}+O\left(\frac{1}{x^3}\right)$$

Parameters

x

An arithmetical expression

Return Values

Arithmetical expression.

Overloaded By

x

Algorithms

Si, *Ssi*, and *Shi* are entire functions.

`Si` and `Ssi` are related by $Ssi(x) = Si(x) - \pi$ for all x in the complex plane.

`Si` and `Shi` are related by $i\,Si(x) = Shi(i\,x)$ for all x in the complex plane.

Reference: M. Abramowitz and I. Stegun, “Handbook of Mathematical Functions”, Dover Publications Inc., New York (1965).

See Also

MuPAD Functions
`Chi` | `Ci` | `Ei` | `int` | `Li` | `Shi` | `sin` | `Ssi`

Ssi

Shifted sine integral function

Syntax

```
Ssi(x)
```

Description

`Ssi(x)` represents the shifted sine integral $\mathrm{Si}(x) - \frac{\pi}{2}$.

The special values $\mathrm{Ssi}(0) = -\frac{\pi}{2}$, $Ssi(\infty) = 0$, $Ssi(-\infty) = -\pi$ are implemented.

The reflection rule $Ssi(x) = -Ssi(-x) - \pi$ is used if the argument is a negative integer or a negative rational number. It is also used if the argument is a symbolic product involving such a factor. Cf. “Example 2” on page 1-1830.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
Si(0), Si(1), Si(sqrt(2)), Si(x + 1), Si(infinity)
```

$$0,\ \mathrm{Si}(1),\ \mathrm{Si}(\sqrt{2}),\ \mathrm{Si}(x+1),\ \frac{\pi}{2}$$

```
Ssi(0), Ssi(1), Ssi(sqrt(2)), Ssi(x + 1), Ssi(infinity)
```

$$-\frac{\pi}{2},\ \mathrm{Ssi}(1),\ \mathrm{Ssi}(\sqrt{2}),\ \mathrm{Ssi}(x+1),\ 0$$

```
Shi(0), Shi(1), Shi(sqrt(2)), Shi(x + 1), Shi(infinity)
```

$$0,\ \mathrm{Shi}(1),\ \mathrm{Shi}(\sqrt{2}),\ \mathrm{Shi}(x+1),\ \infty$$

Floating point values are computed for floating-point arguments:

```
Si(-5.0), Si(1.0), Si(2.0 + 10.0*I)
```

$$-1.549931245,\ 0.9460830704,\ 1187.409493 - 242.5252717\,i$$

```
Ssi(-5.0), Ssi(1.0), Ssi(2.0 + 10.0*I)
```

$$-3.120727572,\ -0.6247132564,\ 1185.838696 - 242.5252717\,i$$

```
Shi(-5.0), Shi(1.0), Shi(2.0 + 10.0*I)
```

$$-20.09321183,\ 1.057250875,\ -0.225644485 + 1.861300836\,i$$

Example 2

The reflection rule *Si*(- *x*) = - *Si*(*x*), *Ssi*(- *x*) = - *Ssi*(*x*) - π, *Shi*(- *x*) = - *Shi*(*x*) is implemented for negative real numbers and products involving such numbers:

```
Si(-3), Si(-3/7), Si(-sqrt(2)), Si(-x/7), Si(-0.3*x)
```

$$-\mathrm{Si}(3),\ -\mathrm{Si}\left(\frac{3}{7}\right),\ -\mathrm{Si}(\sqrt{2}),\ -\mathrm{Si}\left(\frac{x}{7}\right),\ -\mathrm{Si}(0.3\,x)$$

```
Ssi(-3), Ssi(-3/7), Ssi(-sqrt(2)), Ssi(-x/7), Ssi(-0.3*x)
```

$$-\pi - \mathrm{Ssi}(3),\ -\pi - \mathrm{Ssi}\left(\frac{3}{7}\right),\ -\pi - \mathrm{Ssi}(\sqrt{2}),\ -\pi - \mathrm{Ssi}\left(\frac{x}{7}\right),\ -\mathrm{Ssi}(0.3\,x) - 3.141592654$$

```
Shi(-3), Shi(-3/7), Shi(-sqrt(2)), Shi(-x/7), Shi(-0.3*x)
```

$$-\mathrm{Shi}(3),\ -\mathrm{Shi}\left(\frac{3}{7}\right),\ -\mathrm{Shi}(\sqrt{2}),\ -\mathrm{Shi}\left(\frac{x}{7}\right),\ -\mathrm{Shi}(0.3\,x)$$

No such “normalization” occurs for complex numbers or arguments that are not products:

```
Si(- 3 - I), Si(3 + I), Si(x - 1), Si(1 - x)
```

$$\mathrm{Si}(-3-i),\ \mathrm{Si}(3+i),\ \mathrm{Si}(x-1),\ \mathrm{Si}(1-x)$$

```
Ssi(- 3 - I), Ssi(3 + I), Ssi(x - 1), Ssi(1 - x)
```

$$\mathrm{Ssi}(-3-i),\ \mathrm{Ssi}(3+i),\ \mathrm{Ssi}(x-1),\ \mathrm{Ssi}(1-x)$$

```
Shi(- 3 - I), Shi(3 + I), Shi(x - 1), Shi(1 - x)
```

$$\mathrm{Shi}(-3-i),\ \mathrm{Shi}(3+i),\ \mathrm{Shi}(x-1),\ \mathrm{Shi}(1-x)$$

Example 3

The functions `diff`, `float`, `limit`, and `series` handle expressions involving `Si` and `Shi`:

```
diff(Si(x), x, x, x), float(ln(3 + Si(sqrt(PI))))
```

$$\frac{2\sin(x)}{x^3}-\frac{\sin(x)}{x}-\frac{2\cos(x)}{x^2},\ 1.502020149$$

```
diff(Ssi(x), x, x, x), float(ln(3 + Ssi(sqrt(PI))))
```

$$\frac{2\sin(x)}{x^3}-\frac{\sin(x)}{x}-\frac{2\cos(x)}{x^2},\ 1.071568401$$

```
diff(Shi(x), x, x, x), float(ln(3 + Shi(sqrt(PI))))
```

$$\frac{\sinh(x)}{x}-\frac{2\cosh(x)}{x^2}+\frac{2\sinh(x)}{x^3},\ 1.631702794$$

```
limit(Si(2*x^2/(1+x)), x = infinity)
```

$$\frac{\pi}{2}$$

```
limit(Ssi(2*x^2/(1+x)), x = infinity)
```

$$0$$

```
limit(Shi(2*I*x^2/(1+x)), x = infinity)
```

$$\frac{\pi\,\mathrm{i}}{2}$$

```
series(Si(x), x = 0)
```

$$x - \frac{x^3}{18} + \frac{x^5}{600} + O(x^7)$$

```
series(Ssi(x), x = 0)
```

$$-\frac{\pi}{2} + x - \frac{x^3}{18} + \frac{x^5}{600} + O(x^6)$$

```
series(Shi(x), x = 0)
```

$$x + \frac{x^3}{18} + \frac{x^5}{600} + O(x^7)$$

```
series(Si(x), x = infinity, 3)
```

$$\frac{\pi}{2} - \frac{\cos(x)}{x} - \frac{\sin(x)}{x^2} + O\left(\frac{1}{x^3}\right)$$

```
series(Ssi(x), x = infinity, 3)
```

$$-\frac{\cos(x)}{x} - \frac{\sin(x)}{x^2} + \frac{2\cos(x)}{x^3} + O\left(\frac{1}{x^4}\right)$$

```
series(Shi(I*x), x = infinity, 3)
```

$$\frac{\pi\,\mathrm{i}}{2} - \frac{\cos(x)\,\mathrm{i}}{x} - \frac{\sin(x)\,\mathrm{i}}{x^2} + O\left(\frac{1}{x^3}\right)$$

Parameters

x

An arithmetical expression

Return Values

Arithmetical expression.

Overloaded By

x

Algorithms

Si, *Ssi*, and *Shi* are entire functions.

`Si` and `Ssi` are related by $Ssi(x) = Si(x) - \pi$ for all x in the complex plane.

`Si` and `Shi` are related by $\mathrm{i}\,Si(x) = Shi(\mathrm{i}\,x)$ for all x in the complex plane.

Reference: M. Abramowitz and I. Stegun, “Handbook of Mathematical Functions”, Dover Publications Inc., New York (1965).

See Also

MuPAD Functions

`Chi` | `Ci` | `Ei` | `int` | `Li` | `Shi` | `Si` | `sin`

Shi

Hyperbolic sine integral function

Syntax

```
Shi(x)
```

Description

`Shi(x)` represents the hyperbolic sine integral $\int_0^x \frac{\sinh(t)}{t}\,dt$.

If `x` is a floating-point number, then `Shi(x)` returns numerical values. The special values $Shi(0) = 0$, $Shi(\infty) = \infty$, $Shi(-\infty) = -\infty$ are implemented. For all other arguments, `Shi` returns symbolic function calls.

The reflection rule $Shi(x) = -Shi(-x)$ is used if the argument is a negative integer or a negative rational number. It is also used if the argument is a symbolic product involving such a factor. Cf. "Example 2" on page 1-1835.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
Si(0), Si(1), Si(sqrt(2)), Si(x + 1), Si(infinity)
```

$$0,\ \mathrm{Si}(1),\ \mathrm{Si}(\sqrt{2}),\ \mathrm{Si}(x+1),\ \frac{\pi}{2}$$

```
Ssi(0), Ssi(1), Ssi(sqrt(2)), Ssi(x + 1), Ssi(infinity)
```

$$-\frac{\pi}{2},\ \mathrm{Ssi}(1),\ \mathrm{Ssi}(\sqrt{2}),\ \mathrm{Ssi}(x+1),\ 0$$

```
Shi(0), Shi(1), Shi(sqrt(2)), Shi(x + 1), Shi(infinity)
```

$$0,\ \mathrm{Shi}(1),\ \mathrm{Shi}(\sqrt{2}),\ \mathrm{Shi}(x+1),\ \infty$$

Floating point values are computed for floating-point arguments:

```
Si(-5.0), Si(1.0), Si(2.0 + 10.0*I)
```

$$-1.549931245,\ 0.9460830704,\ 1187.409493 - 242.5252717\,i$$

```
Ssi(-5.0), Ssi(1.0), Ssi(2.0 + 10.0*I)
```

$$-3.120727572,\ -0.6247132564,\ 1185.838696 - 242.5252717\,i$$

```
Shi(-5.0), Shi(1.0), Shi(2.0 + 10.0*I)
```

$$-20.09321183,\ 1.057250875,\ -0.225644485 + 1.861300836\,i$$

Example 2

The reflection rule *Si*(- *x*) = - *Si*(*x*), *Ssi*(- *x*) = - *Ssi*(*x*) - π, *Shi*(- *x*) = - *Shi*(*x*) is implemented for negative real numbers and products involving such numbers:

```
Si(-3), Si(-3/7), Si(-sqrt(2)), Si(-x/7), Si(-0.3*x)
```

$$-\mathrm{Si}(3),\ -\mathrm{Si}\left(\frac{3}{7}\right),\ -\mathrm{Si}(\sqrt{2}),\ -\mathrm{Si}\left(\frac{x}{7}\right),\ -\mathrm{Si}(0.3\,x)$$

```
Ssi(-3), Ssi(-3/7), Ssi(-sqrt(2)), Ssi(-x/7), Ssi(-0.3*x)
```

$$-\pi - \mathrm{Ssi}(3),\ -\pi - \mathrm{Ssi}\left(\frac{3}{7}\right),\ -\pi - \mathrm{Ssi}(\sqrt{2}),\ -\pi - \mathrm{Ssi}\left(\frac{x}{7}\right),\ -\mathrm{Ssi}(0.3\,x) - 3.141592654$$

```
Shi(-3), Shi(-3/7), Shi(-sqrt(2)), Shi(-x/7), Shi(-0.3*x)
```

$$-\mathrm{Shi}(3),\ -\mathrm{Shi}\left(\frac{3}{7}\right),\ -\mathrm{Shi}(\sqrt{2}),\ -\mathrm{Shi}\left(\frac{x}{7}\right),\ -\mathrm{Shi}(0.3\,x)$$

No such “normalization” occurs for complex numbers or arguments that are not products:

```
Si(- 3 - I), Si(3 + I), Si(x - 1), Si(1 - x)
```

$$\mathrm{Si}(-3-i),\ \mathrm{Si}(3+i),\ \mathrm{Si}(x-1),\ \mathrm{Si}(1-x)$$

```
Ssi(- 3 - I), Ssi(3 + I), Ssi(x - 1), Ssi(1 - x)
```

$$\mathrm{Ssi}(-3-i),\ \mathrm{Ssi}(3+i),\ \mathrm{Ssi}(x-1),\ \mathrm{Ssi}(1-x)$$

```
Shi(- 3 - I), Shi(3 + I), Shi(x - 1), Shi(1 - x)
```

$$\mathrm{Shi}(-3-i),\ \mathrm{Shi}(3+i),\ \mathrm{Shi}(x-1),\ \mathrm{Shi}(1-x)$$

Example 3

The functions `diff`, `float`, `limit`, and `series` handle expressions involving `Si` and `Shi`:

```
diff(Si(x), x, x, x), float(ln(3 + Si(sqrt(PI))))
```

$$\frac{2\sin(x)}{x^3}-\frac{\sin(x)}{x}-\frac{2\cos(x)}{x^2},\ 1.502020149$$

```
diff(Ssi(x), x, x, x), float(ln(3 + Ssi(sqrt(PI))))
```

$$\frac{2\sin(x)}{x^3}-\frac{\sin(x)}{x}-\frac{2\cos(x)}{x^2},\ 1.071568401$$

```
diff(Shi(x), x, x, x), float(ln(3 + Shi(sqrt(PI))))
```

$$\frac{\sinh(x)}{x}-\frac{2\cosh(x)}{x^2}+\frac{2\sinh(x)}{x^3},\ 1.631702794$$

```
limit(Si(2*x^2/(1+x)), x = infinity)
```

$$\frac{\pi}{2}$$

```
limit(Ssi(2*x^2/(1+x)), x = infinity)
```

$$0$$

```
limit(Shi(2*I*x^2/(1+x)), x = infinity)
```

$$\frac{\pi\,\mathrm{i}}{2}$$

```
series(Si(x), x = 0)
```

$$x - \frac{x^3}{18} + \frac{x^5}{600} + O(x^7)$$

```
series(Ssi(x), x = 0)
```

$$-\frac{\pi}{2} + x - \frac{x^3}{18} + \frac{x^5}{600} + O(x^6)$$

```
series(Shi(x), x = 0)
```

$$x + \frac{x^3}{18} + \frac{x^5}{600} + O(x^7)$$

```
series(Si(x), x = infinity, 3)
```

$$\frac{\pi}{2} - \frac{\cos(x)}{x} - \frac{\sin(x)}{x^2} + O\left(\frac{1}{x^3}\right)$$

```
series(Ssi(x), x = infinity, 3)
```

$$-\frac{\cos(x)}{x} - \frac{\sin(x)}{x^2} + \frac{2\cos(x)}{x^3} + O\left(\frac{1}{x^4}\right)$$

```
series(Shi(I*x), x = infinity, 3)
```

$$\frac{\pi\,\mathrm{i}}{2} - \frac{\cos(x)\,\mathrm{i}}{x} - \frac{\sin(x)\,\mathrm{i}}{x^2} + O\left(\frac{1}{x^3}\right)$$

Parameters

x

An arithmetical expression

Return Values

Arithmetical expression.

Overloaded By

x

Algorithms

Si, *Ssi*, and *Shi* are entire functions.

`Si` and `Ssi` are related by $Ssi(x) = Si(x) - \pi$ for all x in the complex plane.

`Si` and `Shi` are related by $\mathrm{i}\,Si(x) = Shi(\mathrm{i}\,x)$ for all x in the complex plane.

Reference: M. Abramowitz and I. Stegun, “Handbook of Mathematical Functions”, Dover Publications Inc., New York (1965).

See Also

MuPAD Functions

`Chi` | `Ci` | `Ei` | `int` | `Li` | `Si` | `sin` | `Ssi`

sign

Sign of a real or complex number

Syntax

```
sign(z)
```

Description

`sign(z)` returns the sign of the number `z`.

Mathematically, the sign of a complex number $z \neq 0$ is defined as $\frac{z}{|z|}$. For real numbers, this reduces to 1 or - 1.

`sign()` and `sign(0.0)` return `0`. The user may redefine this value by a direct assignment, e.g.:

```
unprotect(sign): sign(0) := 1: protect(sign):
```

If the type of `z` is `DOM_INT`, `DOM_RAT`, or `DOM_FLOAT`, a fast kernel function is used to determine the sign. The return value is either - 1, 0, or 1.

If the sign of the expression cannot be determined, a symbolic function call is returned. Certain simplifications are implemented. In particular, numerical factors of symbolic products are simplified. Cf. “Example 2” on page 1-1840.

The `expand` function rewrites the sign of a product to a product of signs. E.g., `expand(sign(x*y))` yields `sign(x)*sign(y)`. Cf. “Example 2” on page 1-1840.

For constant expressions such as `PI - sqrt(2)`, `exp(I*3) - I*sin(3)` etc., internal floating-point evaluation is used to determine, whether the expression represents a non-zero real number. If so, the sign - 1 or 1 is returned. Internally, the floating-point approximation is checked for reliability. Cf. “Example 4” on page 1-1841.

Environment Interactions

`sign` respects properties of identifiers. For real expressions, the result may depend on the value of the environment variable `DIGITS`.

Examples

Example 1

We compute the sign of various real numbers and expressions:

```
sign(-8/3), sign(3.2), sign(exp(3) - sqrt(2)*PI), sign(0)
```

$$-1,\ 1,\ 1,\ 0$$

The sign of a complex number `z` is the complex number `z/abs(z)`:

```
sign(0.5 + 1.1*I), sign(2 + 3*I), sign(exp(sin(2 + 3*I)))
```

$$0.4138029443 + 0.9103664775\,i,\ \sqrt{13}\left(\frac{2}{13}+\frac{3\,i}{13}\right),\ e^{\cos(2)\sinh(3)\,i}$$

Example 2

`sign` yields a symbolic, yet simplified, function call if identifiers are involved:

```
sign(x), sign(2*x*y), sign(2*x + y),  sign(PI*exp(2 + y))
```

$$\mathrm{sign}(x),\ \mathrm{sign}(x\,y),\ \mathrm{sign}(2\,x+y),\ \mathrm{sign}\left(e^{y+2}\right)$$

In special cases, the `expand` function may provide further simplifications:

```
expand(sign(2*x*y)), expand(sign(PI*exp(2 + y)))
```

$$\mathrm{sign}(x)\ \mathrm{sign}(y),\ \mathrm{sign}(e^{y})$$

Example 3

`sign` respects properties of identifiers:

```
sign(x + PI)
```

$$\operatorname{sign}(\pi + x)$$

```
assume(x > -3): sign(x + PI)
```

$$1$$

```
unassume(x):
```

Example 4

The following rational number approximates π to about 30 digits:

```
p:= 392699081698724154807830422 91/125000000000000000000000000000:
```

With the standard precision `DIGITS =10`, the float test inside `sign` does not give a decisive answer, whether `p` is larger or smaller than π:

```
float(PI - p)
```

$$0.0$$

This result is subject to numerical roundoff and does not allow a conclusion on the sign of the number `PI - p`. The float test inside `sign` checks the reliablity of floating-point approximations. In this case, no simplified result is returned:

```
sign(PI - p)
```

$$\operatorname{sign}\left(\pi - \frac{392699081698724154807830422 91}{125000000000000000000000000000}\right)$$

With increased `DIGITS`, a reliable decision can be taken:

```
DIGITS := 30: sign(PI - p)
```

```
     - 1
```

```
delete p, DIGITS:
```

Parameters

z

An arithmetical expression

Return Values

Arithmetical expression.

Overloaded By

z

See Also

MuPAD Functions

`abs` | `conjugate` | `Im` | `Re`

signIm

Sign of the imaginary part of a complex number

Syntax

```
signIm(z)
```

Description

`signIm(z)` represents the sign of `Im(z)`.

`signIm(z)` indicates whether the complex number `z` lies in the upper or in the lower half plane: `signIm(z)` yields 1 if `Im(z) > 0`, or if `z` is real and `z < 0`. At the origin: `signIm(0)=0`. For all other numerical arguments, - 1 is returned. Thus, `signIm(z)=sign(Im(z))` if `z` is not on the real axis.

If the position of the argument in the complex plane cannot be determined, then a symbolic call is returned. If appropriate, the reflection rule `signIm(-x) = -signIm(x)` is used.

The functions `diff` and `series` treat `signIm` as a constant function. Cf. “Example 2” on page 1-1845.

The following relation holds for arbitrary complex z and p:

$$(-z)^p = z^p \frac{1}{(-1)^{p\,\mathrm{signIm}(z)}}.$$

Further, for arbitrary complex z:

$$\sqrt{z^2} = z\,\mathrm{signIm}(\mathrm{i}\,z)$$

and

$$\ln(-z) = \ln(z) - \mathrm{signIm}(z)\,\pi\,\mathrm{i}.$$

Environment Interactions

Properties of identifiers set via `assume` are taken into account.

Examples

Example 1

For numerical values, the position in the complex plane can always be determined:

```
signIm(2 + I), signIm(- 4 - I*PI), signIm(0.3), signIm(-2/7),
signIm(-sqrt(2) + 3*I*PI)
```

$$1, -1, -1, 1, 1$$

Symbolic arguments without properties lead to symbolic calls:

```
signIm(x), signIm(x - I*sqrt(2))
```

$$\mathrm{signIm}(x),\ \mathrm{signIm}(x - \sqrt{2}\ \mathrm{i})$$

Properties set via `assume` are taken into account:

```
assume(x, Type::Real): signIm(x - I*sqrt(2))
```

$$-1$$

```
assume(x > 0): signIm(x)
```

$$-1$$

```
assume(x < 0): signIm(x)
```

$$1$$

```
assume(x = 0): signIm(x)
```

```
    0
```

```
unassume(x):
```

Example 2

`signIm` is a constant function, apart from the jump discontinuities along the real axis. These discontinuities are ignored by `diff`:

```
diff(signIm(z), z)
```

```
    0
```

Also `series` treats `signIm` as a constant function:

```
series(signIm(z/(1 - z)), z = 0)
```

$$-\mathrm{signIm}\left(\frac{z}{z-1}\right)+O(z^6)$$

Parameters

z

An arithmetical expression representing a complex number

Return Values

Either ± 1, 0, or a symbolic call of type `"signIm"`.

Overloaded By

z

simplify

Simplify an expression

Syntax

```
simplify(f, <target>, options)
```

```
simplify(L, <target>, options)
```

Description

`simplify(f)` tries to simplify the expression `f` by applying term rewriting rules.

`simplify(f, target)` restricts the simplification to term rewriting rules applicable to one or more target functions.

Note: The `cos`, `sin`, `exp`, `ln`, and `sqrt` targets has been removed. Use `simplify` function calls without these targets. Alternatively, use `radsimp` instead of `simplify` with the `sqrt` target.

The `simplify` function performs sequential simplifications. It applies a certain set of term-rewriting rules to the original expression, rewrites the expression according to these rules, takes the result, and applies the next set of term-rewriting rules. The `simplify` function assumes that the output of every rule is "simpler" than the input without further checks. Generally, this method is faster, but less reliable and controllable than the algorithm used by `Simplify`.

If you do not specify a target, `simplify` tries to simplify the whole expression. This first step includes rewriting of products of trigonometric and exponential terms. After that, the function tries to simplify the operands of the expression. If an expression contains special functions, MuPAD calls, the simplification methods available for these functions.

The call `simplify(L, target )` applies simplifications to the operands of the object `L`.

If you specify the `logic` target, the `simplify` function simplifies Boolean expressions. With the `logic` target, `simplify` does not use properties and assumptions specified for the terms of Boolean expressions.

If you specify the `condition` target, the `simplify` function simplifies Boolean expressions. With the `condition` target, `simplify` uses properties and assumptions specified for the terms of Boolean expressions.

Environment Interactions

`simplify` reacts to properties of identifiers.

Examples

Example 1

Use the `simplify` function to simplify the following algebraic expressions:

```
simplify(exp(x)-exp(x/2)^2)
```

$$0$$

```
f := ln(x) + ln(3) - ln(3*x) + (exp(x) - 1)/(exp(x/2) + 1):
simplify(f)
```

$$e^{\frac{x}{2}} - 1$$

Example 2

To simplify Boolean expressions, use the `logic` target:

```
simplify((a and b) or (a and (not b)), logic)
```

$$a$$

Example 3

Alternatively, to simplify Boolean expressions, use the `condition` target. With the `condition` target, `simplify` uses the properties and assumptions specified for the

terms of Boolean expressions. With the `logic` target, `simplify` ignores those properties and assumptions:

```
simplify(x > x, condition), simplify(x > x, logic)
```

$$\text{FALSE}, x < x$$

Example 4

The option `IgnoreAnalyticConstraints` allows you to get simpler results using a set of purely algebraic simplifications:

```
simplify(ln(x^2 + 2*x + 1) - ln(x + 1))
```

$$\ln((x+1)^2) - \ln(x+1)$$

```
simplify(ln(x^2 + 2*x + 1) - ln(x + 1), IgnoreAnalyticConstraints)
```

$$\ln(x+1)$$

If you use this option, the simplifier does not guarantee the equality of the initial expression and the result for all symbolic parameters.

Example 5

To change the number of simplification steps, use the `Steps` option:

```
f := ((exp(-x*i)*i)/2 - (exp(x*i)*i)/2)/(exp(-x*i)/2 + exp(x*i)/2):
simplify(f);
simplify(f, Steps = 10);
simplify(f, Steps = 30)
```

$$-\frac{i\left(e^{2ix}-1\right)}{e^{2ix}+1}$$

$$\frac{2i}{e^{2ix}+1} - i$$

$$-i\tanh(i\,x)$$

Example 6

Your custom functions can have simplification attributes. For example, suppose you know that *f* is an additive function, but you do not know more about *f*. Therefore, you cannot compute the function value of at any point except zero, but you can use the additivity:

```
f := funcenv( x -> if iszero(x) then 0 else procname(x) end):
f::simplify := proc(F)
                 local argument;
                 begin
                   argument := op(F,1);
                   if type(argument) = "_plus" then
                     map(argument, f)
                   else
                     F
                   end
                 end:

f(x + 3*y) - f(3*y) = simplify(f(x + 3*y) - f(3*y))
```

$$f(x+3\,y)-f(3\,y)=f(x)$$

You can refine the simplification attribute of `f` further. For example, you can specify that it must turn `f(3*y)` into `3*f(y)`. The reverse rule (rewriting $f(x) + f(y)$ as $f(x + y)$) is not context-free. Therefore, you cannot implement the reverse rule in a simplification attribute.

Parameters

`f`

An arithmetical expression

`L`

A container object: an array, an hfarray, a list, a matrix, a polynomial, a set, or a table.

target

One of the identifiers `unit`, `logic`, or `condition`

Options

IgnoreAnalyticConstraints

With this option the simplifier applies the following rules to expressions:

- $ln(a) + ln(b) = ln(a\,b)$ for all values of a and b. In particular:

  $(a\,b)^c = e^{c\,\ln(a\,b)} = e^{c\,(\ln(a)+\ln(b))} = a^c\,b^c$ for all values of a, b, and c
- $ln(a^b) = b\,ln(a)$ for all values of a and b. In particular:

  $(a^b)^c = e^{b\,c\,\ln(a)} = e^{\ln(a)^{b\,c}} = a^{b\,c}$ for all values of a, b, and c
- If f and g are standard mathematical functions and $f(g(x)) = x$ for all small positive numbers, $f(g(x)) = x$ is assumed to be valid for all complex x. In Particular:
- - $\ln(e^x) = x$
  - $arcsin(sin(x)) = x$, $arccos(cos(x)) = x$, $arctan(tan(x)) = x$
  - $arcsinh(sinh(x)) = x$, $arccosh(cosh(x)) = x$, $arctanh(tanh(x)) = x$
  - $W_k(x\,e^x) = x$ for all values of k

Using the `IgnoreAnalyticConstraints` option can give you simple results for expressions for which the direct use of the simplifier returns complicated results. With this option the simplifier does not guarantee the equality of the initial expression and the result for all symbolic parameters. See “Example 4” on page 1-1848.

Seconds

Limit the time allowed for the internal simplification process. The value denotes the maximal time in seconds. By default, the simplification time is unlimited.

Steps

Terminate algebraic simplification after the specified number of simplification steps. The value must be a positive integer. the default number of simplification steps is 1.

Return Values

Object of the same type as the input object `f` or `L`, respectively.

Overloaded By

f, L

See Also

MuPAD Functions

`collect` | `combine` | `expand` | `factor` | `match` | `normal` | `radsimp` | `rectform` | `rewrite` | `Simplify`

More About

- "If You Want to Simplify Results Further"

Simplify

Simplify an expression

Syntax

```
Simplify(f)
```

```
Simplify(f, Steps = numberOfSteps)
```

```
Simplify(f, options)
```

Description

`Simplify(f)` applies term-rewriting rules to `f` and returns the simplest expression it can find.

The methods of searching for the simplest representation of an expression are different for `Simplify` and `simplify`. The `simplify` function performs a linear search trying to improve the result returned by the previous simplification step. The `Simplify` function uses the results returned by all previous simplification steps (the best-first search). The `simplify` function is faster. The `Simplify` function is slower, but more effective and more configurable.

The term “simplest” is defined as follows. One object is simpler than another if it has smaller *valuation*. If two objects have the same valuation, `Simplify` chooses the simpler object using internal heuristics. A valuation is a function that assigns a nonnegative real number to each MuPAD object. To override the default valuation used by `Simplify`, use the `Valuation` option. `Simplify` uses the valuation for the best-first search as for determining the best result at the final step. However, you can define a separate method for the final simplification step by using the `Criterion` option.

The simplification process consists of *steps*. In each step, `Simplify` performs one of the following kinds of tasks for `a := f` or some (previously obtained) object `a` equivalent to `f`. In each step, `Simplify` can produce new objects equivalent to `f` (results) or new tasks to do or both:

- Initial step: Find all *rules* for `a`. The `Simplify` function performs the search for all rules for every new object `a`. This search produces no new result.

- Rewriting step: Apply one rule to `a`. This step can either fail or produce an equivalent object as result.
- Subexpression step: Perform one step in simplifying an operand of `a`. Replace the operand with the returned result (if there are any results). This step can produce a new equivalent object.

Each open task has a priority that determines what to do next. Simplification terminates in any of the following cases:

- There are no more open tasks.
- `Simplify` reached the time limit specified by `Seconds`.
- `Simplify` performed the maximal number of simplification steps specified by `Steps`.
- `Simplify` returned the object specified by `Goal`.

`Simplify` always returns the "simplest" equivalent object found in all simplification steps unless you specify another `OutputType`.

Rules form a particular domain `Rule`. They consist of a pattern (left side), an expression (right side), and options.

MuPAD organizes rules for `Simplify` in *rule bases*. You can change the default rule base by using the `RuleBase` option. You also can define your own rule selection mechanism by using the `SelectRules` option.

Typically, `Simplify` applies the selected rules to the given object `a` as a whole. The following case is an exception from this rule. If the pattern of the rule and the object `a` are both a sum or a product, then `Simplify` applies the rule to each subsum or subproduct of `a` that has the same number of operands as the pattern.

By using the `ApplyRule` option, you can specify your own function that applies a particular rule to a particular object. Otherwise, `Simplify` uses a default method.

The application of a rule to an object `a` fails if the pattern does not match (see `match`) the object `a`. The performance of `Simplify` strongly depends on the number of successful matches. Therefore, if you specify your own rule base, it must dispose of non-matching rules before rule selection.

A simplification step for an operand works like a simplification step on simplifying `f`. The exceptions are as follows. Performing a simplification step for an operand, MuPAD does not apply certain rules (see the details on `SelectRules`).

MuPAD determines priorities of open tasks as follows. The priority of doing the initial step for an expression depends on the valuation of the expression. The priority of doing a simplification step on an operand depends on the ratio between the overall valuation of the expression and the valuation of the operand and the priority of the highest-rank task in the to-do list of the operand. Finally, the priority of applying a rule to an expression equals to the priority of the rule multiplied by the valuation of the expression.

The strategy determines the priority of a rule. See the `Strategy` option for details.

`Simplify` never uses the symmetry of mathematical equivalence of expressions. Therefore, you can use `Simplify` as a general rewriting system.

`Simplify` maps to lists, sets, and tables.

For domain elements `d`, `Simplify` can be overloaded in two ways. First, `Simplify` uses the slot `d::dom::Simplify`. If that slot does not exist, `Simplify` uses `d::dom::simplify`. If the slot that `Simplify` uses is not a list, `Simplify` calls the slot and accepts the result as simple (even if the valuation does not agree). In this case, `Simplify` does not apply any other rules to `d`. However, `Simplify` uses the valuation to decide whether it must replace a domain element that occurs as an operand in another object with its "simplified" version. If the slot is a list, its entries must be rules, and `Simplify` applies them according to their priority.

Examples

Example 1

The easiest way to use `Simplify` is to accept all defaults, and then plug in the expression you want to simplify:

```
Simplify(sin(x)^2 + cos(x)^2)
```

```
1
```

Example 2

By default, `Simplify` returns only one expression that the function considers as simplest. To return a list of all equivalent expressions, use the `All` option:

```
Simplify(sin(x)^2 + cos(x)^2, All)
```

$$[1, \cos(x)^2 + \sin(x)^2]$$

Example 3

The output of the previous example is short because as soon as the simplifier finds 1, it stops immediately. After that, the simplifier does not look for other equivalent expressions. In addition, the simplifier discards the equivalent expressions that are significantly more complicated than the best expression found earlier. You can switch off both mechanisms:

```
Simplify(sin(x)^2 + cos(x)^2, All, Discard = FALSE,
                                   IsSimple = FALSE)
```

$$\left[1,\ \cos(x)^2+\sin(x)^2,\ -\cos(x)^2-\sin(x)^2+2,\ \sin(x)^2+\sigma_5+\frac{1}{2},\ \cos(x)^2-\sigma_5+\frac{1}{2},\right.$$
$$-\cos(x)^2+\sigma_5+\frac{3}{2},\ -\sin(x)^2-\sigma_5+\frac{3}{2},\ \sigma_7+\sigma_8,\ 2-\sigma_8-\sigma_7,\ \sigma_8-\sigma_2+\frac{1}{2},\ \sigma_2-\sigma_8+\frac{3}{2},$$
$$\sigma_2+\sigma_7+\frac{1}{2},\ \frac{3}{2}-\sigma_7-\sigma_2,\ \sigma_3+\sigma_1,\ 2-\sigma_1-\sigma_3,\ \frac{1}{2}-\sigma_6+\sigma_3-\sigma_4,\ \sigma_4+\sigma_6-\sigma_3+\frac{3}{2},$$
$$\left.\sigma_4+\sigma_6+\sigma_1+\frac{1}{2},\ \frac{3}{2}-\sigma_6-\sigma_1-\sigma_4,\ \frac{1}{\sigma_9}+\frac{2\,\sigma_{10}}{\sigma_9}+\frac{\sigma_{11}}{\sigma_9}\right]$$

where

$$\sigma_1=\left(\frac{e^{-x\,i_i}}{2}-\frac{e^{x\,i_i}}{2}\right)^2$$

$$\sigma_2=\frac{\tan(x)^2-1}{2\left(\tan(x)^2+1\right)}$$

$$\sigma_3=\left(\frac{e^{-x\,i}}{2}+\frac{e^{x\,i}}{2}\right)^2$$

$$\sigma_4=\frac{e^{-2\,x\,i}}{4}$$

$$\sigma_5=\frac{\cos(2\,x)}{2}$$

$$\sigma_6=\frac{e^{2\,x\,i}}{4}$$

$$\sigma_7=\frac{\left(\sigma_{10}-1\right)^2}{\left(\sigma_{10}+1\right)^2}$$

$$\sigma_8=\frac{4\,\sigma_{10}}{\left(\sigma_{10}+1\right)^2}$$

$$\sigma_9=\sigma_{11}+2\,\sigma_{10}+1$$

Example 4

By default, `Simplify` uses a valuation that favors expressions with fewer different irrational subexpressions. For example, `Simplify` assumes that an expression containing only `sin(x)` or `cos(x)` is simpler than an expression containing both:

```
Simplify(cos(x)*sin(x))
```

$$\frac{\sin(2\,x)}{2}$$

If you take the `length` as a complexity measure for expressions, `Simplify` returns another result:

```
Simplify(cos(x)*sin(x), Valuation = length)
```

$$\cos(x)\,\sin(x)$$

Example 5

The default number of steps is 100. To change the maximal number of possible simplification steps, use the `Steps` option. For example, decrease (resulting in a speedup) and increase (resulting in a probably better simplification) the number of simplification steps:

```
f := ln(x) + ln(3) - ln(3*x) + (exp(x) - 1)/(exp(x/2) + 1):
Simplify(f, Steps = 8), Simplify(f, Steps = 120)
```

$$\frac{e^{x}}{e^{\frac{x}{2}}+1}-\frac{1}{e^{\frac{x}{2}}+1},\ e^{\frac{x}{2}}-1$$

```
delete f:
```

Example 6

For many expressions, the default number of simplification steps does not allow the simplifier to find a good simplification:

```
Simplify(e^(a* x *(a + 1) + b2* y *(y + b2* x* y)))
```

$$e^{b2\,y\,(y+b2\,x\,y)+a\,x\,(a+1)}$$

Increasing this limit often helps:

```
Simplify(e^(a* x *(a + 1) + b2* y *(y + b2* x* y)), Steps=125)
```

$$e^{b2\,(b2\,x+1)\,y^2+a\,x\,(a+1)}$$

Example 7

By default, simplification functions do not combine logarithms:

```
Simplify(ln(x^3 - 1) - ln(x - 1))
```

$$\ln(x^3-1)-\ln(x-1)$$

Using the `IgnoreAnalyticConstraints` option, you often can get shorter results:

```
Simplify(ln(x^3 - 1) - ln(x - 1), IgnoreAnalyticConstraints)
```

$$\ln(x^2+x+1)$$

Example 8

You can write the same expression in different coordinate systems. For example, use Cartesian and polar coordinates:

```
assume(x/r = cos(Symbol::theta)):
assumeAlso(y/r = sin(Symbol::theta)):
assumeAlso(r = sqrt(x^2+y^2)):
x/sqrt(x^2+y^2) + I*y/sqrt(x^2+y^2) = exp(I*Symbol::theta);
Simplify(%)
```

$$\frac{x}{\sqrt{x^2+y^2}}+\frac{y\,\mathrm{i}}{\sqrt{x^2+y^2}}=\mathrm{e}^{\theta\,\mathrm{i}}$$

TRUE

Example 9

The following expression is equivalent to `exp(x)`:

```
a := -1/(sin(1/2*I*x)^2 + 4*sin(1/4*I*x)^4 -
     4*sin(1/4*I*x)^2 + 1)*(sin(1/2*I*x)^2 -
     4*I*sin(1/2*I*x)*sin(1/4*I*x)^2 + 2*I*sin(1/2*I*x) -
     4*sin(1/4*I*x)^4 + 4*sin(1/4*I*x)^2 - 1)
```

$$-\frac{{\sigma_1}^2-4\,\sigma_1\,\sin\left(\frac{x\,i}{4}\right)^2\,i+2\,\sigma_1\,i-4\,\sin\left(\frac{x\,i}{4}\right)^4+4\,\sin\left(\frac{x\,i}{4}\right)^2-1}{{\sigma_1}^2+4\,\sin\left(\frac{x\,i}{4}\right)^4-4\,\sin\left(\frac{x\,i}{4}\right)^2+1}$$

where

$$\sigma_1=\sin\left(\frac{x\,i}{2}\right)$$

`Simplify` recognizes the equivalence of `a` and `exp(x)` within 100 steps. To show how the function proves the equivalence at each step, use the `OutputType` option. Note that the proof returned by `Simplify` is not a proof in a strict mathematical sense. `Simplify` uses the rules from the default rule base:

```
Simplify(a, OutputType = "Proof")
```

```
Input was
  -(sin((x*I)/2)^2 - 4*sin((x*I)/2)*sin((x*I)/4)^2*I + 2*sin((x*I)/2)*I - \
4*sin((x*I)/4)^4 + 4*sin((x*I)/4)^2 - 1)/(sin((x*I)/2)^2 + 4*sin((x*I)/4)^\
4 - 4*sin((x*I)/4)^2 + 1)
Applying the rule
  Simplify::combineSinCos
gives
  cos(x*I) - sin(x*I)*I
Applying the rule
  Simplify::expand
gives
  cosh(x) + sinh(x)
```

```
Applying the rule
  X -> rewrite(X, exp)
gives
  exp(x)
END OF PROOF
```

Example 10

You also can use `Simplify` for experiments with formal grammars given by only a few rules. In this case, the better approach is not to use rule bases, but to use a `SelectRules` method that returns a list of all rules. The following example presents a general associative operator `?`. The example computes the number of all possible placements of parentheses. First, define the `operator`, and then attach it to a function that controls its output (see `funcenv`). Specify that the only applicable rule is the associative law. In the call to `Simplify`, set the number of steps to a very large value to perform a complete search. Note that most grammars produce infinitely many words and spend infinite time to finish a complete search:

```
_f := funcenv(() -> procname(args())):
operator("?", _f, Binary, 1000):
R := Rule((X ? Y) ? Z, X ? (Y ? Z)):
selectProc := () -> [R]:
S := Simplify(u ? v ? x ? y ? z, Steps = 10^10,
              SelectRules = selectProc, All):

PRETTYPRINT := FALSE:
print(Plain, S):
PRETTYPRINT := TRUE:


[u ? (v ? (x ? (y ? z))), u ? (v ? ((x ? y) ? z)), u ? ((v ? (x ? y)) ? z)\
, u ? (((v ? x) ? y) ? z), u ? ((v ? x) ? (y ? z)), (u ? (v ? x)) ? (y ? z\
), ((u ? v) ? x) ? (y ? z), (u ? v) ? (x ? (y ? z)), (u ? v) ? ((x ? y) ? \
z), (u ? (v ? (x ? y))) ? z, (u ? ((v ? x) ? y)) ? z, ((u ? (v ? x)) ? y) \
? z, (((u ? v) ? x) ? y) ? z, ((u ? v) ? (x ? y)) ? z]
```

There are 14 possible ways of placing parentheses:

```
nops(S);
delete fout, _f, R, S, selectProc:
operator("?", Delete):

     14
```

Example 11

If you want to specify a larger set of rules, the best approach is to use your own rule base. A classic example is differentiation. Although a heuristic search must be slower than a simple recursive algorithm, this example is suitable for demonstrating some efficiency considerations. Start by defining a function environment `mydiff` that does not do anything:

```
mydiff := funcenv(mydiff):
mydiff::type := "mydiff"
```

```
"mydiff"
```

The goal of this definition is to show that MuPAD sorts rules in rule bases by the types of expressions to which MuPAD applies those rules. Therefore, `mydiff` gets its own type. Now, define a rule base `Myrules` with the usual differentiation rules. Do not use any additional rules:

```
Myrules := newDomain("Myrules"):
Myrules::mydiff :=
     [Rule(mydiff(f, x), 0, {(f, x) -> not has(f, x)}),
      Rule(mydiff(x, x), 1),
      Rule(mydiff(x^n, x), n*x^(n - 1)),
      Rule(mydiff(f*g, x), f*mydiff(g, x) + g*mydiff(f, x)),
      Rule(mydiff(f + g, x), mydiff(f, x) + mydiff(g, x))
     ]:
```

This rule base works for the expression x^2:

```
Simplify(mydiff(x^2, x), RuleBase = Myrules)
```

$$2\,x$$

However, the rule base does not work for the following expression:

```
Simplify(mydiff(x + 3, x), RuleBase = Myrules)
```

$$\mathrm{mydiff}(x+3,\,x)$$

Try to improve that rule base. As a first step, increase the number of simplification steps. Increasing the number of steps does not help in this case:

```
Simplify(mydiff(x + 3, x), RuleBase = Myrules, Steps = 200)
```

$$\text{mydiff}(x+3, x)$$

As a second step, take a closer look on the equivalent expressions returned by `Simplify`. Sometimes, `Simplify` finds the expected result, but does not return it because the valuation of the expected result is higher than the valuation of some other equivalent expression. For the expression `x + 3`, the `Simplify` function does not find the expected result:

```
l := Simplify(mydiff(x + 3, x), RuleBase = Myrules, All)
```

$$[\sigma_1, \text{mydiff}(x, x) + \text{mydiff}(3, x), (x+3)\,\text{mydiff}(1, x) + \sigma_1, 2\,(x+3)\,\text{mydiff}(1, x) + \sigma_1,$$
$$3\,(x+3)\,\text{mydiff}(1, x) + \sigma_1, 4\,(x+3)\,\text{mydiff}(1, x) + \sigma_1, 5\,(x+3)\,\text{mydiff}(1, x) + \sigma_1,$$
$$6\,(x+3)\,\text{mydiff}(1, x) + \sigma_1, 7\,(x+3)\,\text{mydiff}(1, x) + \sigma_1, 8\,(x+3)\,\text{mydiff}(1, x) + \sigma_1,$$
$$9\,(x+3)\,\text{mydiff}(1, x) + \sigma_1, 10\,(x+3)\,\text{mydiff}(1, x) + \sigma_1, 11\,(x+3)\,\text{mydiff}(1, x) + \sigma_1,$$
$$12\,(x+3)\,\text{mydiff}(1, x) + \sigma_1, 13\,(x+3)\,\text{mydiff}(1, x) + \sigma_1, 14\,(x+3)\,\text{mydiff}(1, x) + \sigma_1,$$
$$15\,(x+3)\,\text{mydiff}(1, x) + \sigma_1, 16\,(x+3)\,\text{mydiff}(1, x) + \sigma_1, 17\,(x+3)\,\text{mydiff}(1, x) + \sigma_1,$$
$$18\,(x+3)\,\text{mydiff}(1, x) + \sigma_1, 19\,(x+3)\,\text{mydiff}(1, x) + \sigma_1, 20\,(x+3)\,\text{mydiff}(1, x) + \sigma_1,$$
$$21\,(x+3)\,\text{mydiff}(1, x) + \sigma_1, 22\,(x+3)\,\text{mydiff}(1, x) + \sigma_1, 23\,(x+3)\,\text{mydiff}(1, x) + \sigma_1,$$
$$24\,(x+3)\,\text{mydiff}(1, x) + \sigma_1, 25\,(x+3)\,\text{mydiff}(1, x) + \sigma_1, 26\,(x+3)\,\text{mydiff}(1, x) + \sigma_1,$$
$$27\,(x+3)\,\text{mydiff}(1, x) + \sigma_1, 28\,(x+3)\,\text{mydiff}(1, x) + \sigma_1, 29\,(x+3)\,\text{mydiff}(1, x) + \sigma_1,$$
$$30\,(x+3)\,\text{mydiff}(1, x) + \sigma_1, 31\,(x+3)\,\text{mydiff}(1, x) + \sigma_1]$$

where

$$\sigma_1 = \text{mydiff}(x+3, x)$$

Note that the derivative of 1 appears in the result. Use the `OutputType` option, to check how `Simplify` manipulates the third term `l[3]` and how it proves the equivalence of input and output at each step:

```
Simplify(mydiff(x + 3, x), RuleBase = Myrules,
                           Goal = l[3],
                           OutputType = "Proof")
```

```
Input was
  mydiff(x + 3, x)
Applying the rule
  mydiff(f*g, x) -> f*mydiff(g, x) + g*mydiff(f, x)
gives
  (x + 3)*mydiff(1, x) + mydiff(x + 3, x)
END OF PROOF
```

Now you can see that for each expression f, you must specify the rule for diffentiating products because $f = 1 f$. Modify that rule:

```
(Myrules::mydiff)[4] := Rule(mydiff(f*g, x),
                             f*mydiff(g, x) + g*mydiff(f, x),
                             {(f, g) -> f <> 1 and g <> 1}):
```

The updated rule base works:

```
Simplify(mydiff(x + 3, x), RuleBase=Myrules, Remember=FALSE)
```

```
     1
```

Use a few options to optimize the call to `Simplify`. As a first step, measure how many steps a typical example takes before returning the expected output:

```
Simplify(mydiff(5*x^4 + x^3 + x^2 + x + 1, x),
         RuleBase = Myrules,
         Steps = 2000,
         Goal = 20*x^3 + 3*x^2 + 2*x + 1,
         OutputType = "NumberOfSteps")
```

```
     134
```

Avoid the application of the equality $f = f + 0$. Switch off the remember mechanism. When the remember mechanism works, `Simplify` ignores changes in the rule base:

```
Myrules::mydiff[5] := Rule(mydiff(f + g, x),
                           mydiff(f, x) + mydiff(g, x),
                           {(f, g) -> f <> 0 and g <> 0}):
Simplify(mydiff(5*x^4 + x^3 + x^2 + x + 1, x),
         RuleBase = Myrules,
         Steps = 2000,
         Goal = 20*x^3 + 3*x^2 + 2*x + 1,
         OutputType = "NumberOfSteps",
         Remember = FALSE)
```

```
     129
```

Next, try to change the valuation criteria. For example, use `length`:

```
Simplify(mydiff(5*x^4 + x^3 + x^2 + x + 1, x),
         RuleBase = Myrules,
         Steps = 2000,
         Goal = 20*x^3 + 3*x^2 + 2*x + 1,
         OutputType = "NumberOfSteps",
         Valuation = length)
```

```
     121
```

To optimize the call to `Simplify`, you also can specify your own simplification *strategy*. For example, the first rule seems to provide a very useful simplification whenever it applies. Therefore, assign a high priority to this rule by assuming that on average this rule simplifies its input to 0.03 of the original complexity:

```
Myrules::mydiff[1] := subsop(Myrules::mydiff[1],
                             4 = table("MyStrategy" = 0.03)):
Simplify(mydiff(5*x^4 + x^3 + x^2 + x + 1, x),
         RuleBase = Myrules,
         Steps = 3000,
         Goal = 20*x^3 + 3*x^2 + 2*x + 1,
         OutputType = "NumberOfSteps",
         Strategy = "MyStrategy")
```

```
     124
```

When using the valuation `length`, you get the following result:

```
Simplify(mydiff(5*x^4 + x^3 + x^2 + x + 1, x),
         RuleBase = Myrules,
         Steps = 3000,
         Goal = 20*x^3 + 3*x^2 + 2*x + 1,
         OutputType = "NumberOfSteps",
         Strategy = "MyStrategy",
         Valuation = length)
```

```
     125
```

When you use a matcher-based simplification, most of the rules do not match to most objects. Trying to match all rules to all objects produces many failing rewriting steps. The recommended approach is to discard these failing rules during the initial step. Discarding failling rules decreases the number of steps. It also increases the running time per step by a small amount. Defining a procedure instead of a list of rules can help you to discard the failing rules during an initial step. You can define the rules by using a pattern or a procedure as their first argument:

```
Myrules::mydiff :=
proc(df)
  begin
  [if not has(op(df, 1), op(df, 2)) then
      Rule(X -> 0)
   else
      case type(op(df, 1))
        of "_plus" do
           Rule(X -> map(op(X, 1), mydiff, op(X, 2)));
           break
        of "_mult" do
           Rule(mydiff(f*g, x), f*mydiff(g, x) + g*mydiff(f, x));
           break
        of "_power" do
           Rule(X -> op(X, [1,2])*op(X, [1,1])^(op(X, [1,2]) - 1));
           break
        of DOM_IDENT do
           assert(op(df, 1) = op(df, 2));
           Rule(X -> 1);
           break
        otherwise
           null()
       end_case
    end_if]
end_proc:
```

```
Simplify(mydiff(5*x^4 + x^3 + x^2 + x + 1, x),
         RuleBase = Myrules,
         Steps = 200,
         Goal = 20*x^3 + 3*x^2 + 2*x + 1,
         OutputType = "NumberOfSteps")

     35

delete Myrules, mydiff:
```

Parameters

`f`

Any object

Options

`All`

When you use the `All` option, the `Simplify` function returns a list of all equivalent objects that the function can find. This syntax is a shortcut for `OutputType = "All"`.

`ApplyRule`

Option, specified as `ApplyRule = applyFunction`

Specify the function `applyFunction` that `Simplify` calls every time when a rule `R` must be applied to an object `a`. Here, `applyFunction` must be a function of two arguments `R` (a rule) and `a` (an object). It must return the result of applying the rule `R` to an object `a`. If the rule is not applicable, the `applyFunction` function must return `FAIL`.

`Discard`

Option, specified as `Discard = discardFunction`

Specify the function `discardFunction(newvalue, bestvalue)` that `Simplify` calls every time it finds a new object equivalent to `f`. Here `newvalue` is the valuation

of the new object, and `bestvalue` is the minimal valuation among all equivalent objects that `Simplify` found earlier. If Boolean evaluation of the result produces `TRUE`, then `Simplify` discards the new object. By default, `Simplify` discards a result if its valuation exceeds `10×current best valuation + 1`. To prevent the loss of results, switch this mechanism off: `Discard = FALSE`.

`Criterion`

Option, specified as `Criterion = CriterionFunction`

Specify the function `CriterionFunction(a, vala)` that `Simplify` calls at the end of the computation to perform the final sorting of the results. For each result `a` and its valuation `vala`, `CriterionFunction(a, vala)` returns a number. `Simplify` uses that number to sort the results. By default, `Simplify` uses `vala` to sort the results.

`Goal`

Option, specified as `Goal = a`

If the `Simplify` function finds the equivalent object `a`, stop the computation and return `a` even if this object is not the simplest equivalent expression found.

`IgnoreAnalyticConstraints`

With this option the simplifier applies the following rules to expressions:

- $ln(a) + ln(b) = ln(a\,b)$ for all values of a and b. In particular:

  $(a\,b)^c = e^{c\,\ln(a\,b)} = e^{c\,(\ln(a)+\ln(b))} = a^c\,b^c$ for all values of a, b, and c
- $ln(a^b) = b\,ln(a)$ for all values of a and b. In particular:

  $(a^b)^c = e^{b\,c\,\ln(a)} = e^{\ln(a)^{b\,c}} = a^{b\,c}$ for all values of a, b, and c
- If f and g are standard mathematical functions and $f(g(x)) = x$ for all small positive numbers, $f(g(x)) = x$ is assumed to be valid for all complex x. In particular:
- - $\ln(e^x) = x$
  - $arcsin(sin(x)) = x$, $arccos(cos(x)) = x$, $arctan(tan(x)) = x$
  - $arcsinh(sinh(x)) = x$, $arccosh(cosh(x)) = x$, $arctanh(tanh(x)) = x$
  - $W_k(x\,e^x) = x$ for all values of k

With this option, the Simplify function can return simple results for expressions for which Simplify without this option returns more complicated results. With this option the simplifier does not guarantee the equality of the initial expression and the result for all symbolic parameters. See “Example 7” on page 1-1858.

IsSimple

Option, specified as IsSimple = B

Specify the function B(a) that the Simplify function calls for any expression a that is equivalent to any subexpression of the input. If the result of the call is TRUE, then the Simplify function does not simplify this subexpression any further. B must return TRUE or FALSE for every input.

KernelSteps

Option, specified as KernelSteps = n

Limit the effort invested in one simplification step. Here n must be a positive integer. The default value is 100.

OutputType

Option, specified as OutputType = output

Specify the type of return value. The value output must be one of the strings "All", "Best", "NumberOfSteps", or "Proof". This option makes Simplify return all results, the best result, the number of performed simplification steps, or a proof for the equivalence of the input and the best result. By default, Simplify returns the simplest result found.

Even if you specify the output type as "All", Simplify does not return any results discarded due to the Discard option. To get all results, set Discard to FALSE. See “Example 3” on page 1-1855.

If you set this option to "Proof", the Simplify function returns text displaying proof steps and lemmas. Proof steps state that f_{i-1} is equivalent to f_i for $1 \le i \le n$, where $f_0 = f$ is the input and f_n is the result of the simplification. Each proof step is either a rule application or a lemma application. A rule application step shows that applying a rule to f_{i-1} gives f_i. A lemma application steps shows that replacing some operand of f_{i-1} by an equivalent object gives f_i. A lemma consists of the statement that two objects are equivalent, a proof in the above sense, and the END OF LEMMA tag.

Technically, proofs are objects of the same type as the output of `expose`.

`Remember`

Option, specified as `Remember = bool`

The `Remember` option switches the remember mechanism on and off. If you call `Simplify` with the same argument several times, the remember mechanism saves running time. If the argument of one call reappears as a subexpression in the argument of another call, the remember mechanism does not help to save time. By default, `bool` is `TRUE`.

`RuleBase`

Option, specified as `RuleBase = base`

A rule base `base` is a domain that contains its rules for expressions of type `T` in its slot `slot(base, T)`. In addition, the following three slots can contain rules for a rule base: `base::All`, `base::Global`, and `base::Local`. The `base::All` slot contains generally applicable rules. The `base::Global` slot contains rules that the `Simplify` function applies only to expressions. `Simplify` does not apply the rules defined in `base::Global` within a subexpression step. The `base::Local` slot contains rules that the `Simplify` function applies only within a subexpression step. If no slots exist for the type of a given object, MuPAD does not generate any rules for that object. A slot of a rule base is a list of rules or a procedure that returns such a list for any given object of appropriate type. Any rule must be an object of type `Rule`. See the `Rule` help page for details. If you use your own `SelectRules`, you can ignore these conventions. See “Example 11” on page 1-1861.

`Seconds`

Option, specified as `Seconds = t`

When you use the `Seconds` option, the `Simplify` function limits the time allowed for the internal simplification process. The value `t` is the maximal time in seconds. By default, the simplification process never terminates due to a time limitation: `t = infinity`.

`SelectRules`

Option, specified as `SelectRules = selFunction`

When you use the `SelectRules` option, MuPAD lets you specify the function `selFunction(base, ex, global, strat)` that `Simplify` calls to obtain the rules

applicable to ex in the rule base base for strategy strat. The boolean flag global indicates whether ex is the whole expression accepted by Simplify (global = TRUE) or a subexpression of the original expression (global = FALSE). Using the arguments given to selFunction is optional. For example, for small rule bases the easiest method is to return a list of all rules independent of the given expression. See “Example 10” on page 1-1860. However, returning a list of all rules can result in unnecessary rule applications. Applying each unnecessary rule returns FAIL and only affects the performance.

You can define any rule base and use any kind the rules. The only restriction is that selFunction must return a list of rules.

Steps

Option, specified as Steps = numberOfSteps

When you use the Steps option, the Simplify function terminates a simplification after numberOfSteps simplification steps. The default number of steps is 100.

StopIf

Option, specified as StopIf = B

When you use the StopIf option, the Simplify function lets you specify the function B(a) that Simplify calls for any expression a that is equivalent to the original expression. If the result is TRUE, the simplification stops immediately, and the Simplify function returns the expression a as the simplest result regardless of its valuation. The specified function B must return TRUE or FALSE for any input.

Strategy

Option, specified as Strategy = strat

When you use the Strategy option, the Simplify function lets you set the rule selection strategy. The value of strat must be a string. The SelectRules option uses strat as an argument that determines the priority for applying each rule.

By default, Simplify uses the strategy "Default". The default rule base also uses the strategy "Default".

If a particular rule does not recognize the strategy strat, the Simplify function uses the strategy "Default" to determine the priority of that rule. Finally, if no entry for the

default strategy is available, the rule has the priority 1. In this case, expect an output to be as complicated as the input.

If you use the `IgnoreAnalyticConstraints`, `Simplify` uses the strategy that comes with that option instead of using the strategy `"Default"`.

If `Simplify` uses a strategy, that strategy does not affect the valuation of results of rule applications.

`Valuation`

Option, specified as `Valuation = valFunction`

When you use the `Strategy` option, the `Simplify` function lets you specify a function MuPAD uses for computing valuations of returned objects. `Simplify` computes the valuation for many intermediate results. Generally, to compute the valuation, `Simplify` evaluates each node of the expression tree. Therefore, the `Valuation` option can significantly affect the running time.

A good valuation is a compromise between context-free and maximum-type concepts. For a context-free valuation, both the operator of an expression and the valuations of the operands determine the valuation of the expression. For a maximum-type valuation, generally the valuation of an expression equals the maximum of valuations of its operands.

A typical context-free example is `length`. A typical maximum-type example is `X -> 2^nops(indets(X))` .

MuPAD offers a context-free valuation `Simplify::complexity`. This valuation favors usual operators like `exp` over unusual ones like `besselJ` and puts a penalty factor on arguments of unusual operators.

Return Values

`Simplify` returns an object mathematically equivalent to the input. With the option `OutputType = "All"`, the `Simplify` function returns a list of all equivalent objects found during the simplification. With the option `OutputType = "NumberOfSteps"`, the function returns a positive integer. With the option `OutputType = "Proof"`, the function returns a string containing a proof of the equivalence of the input and the result.

Overloaded By

f

See Also

MuPAD Functions

`collect` | `combine` | `expand` | `factor` | `match` | `normal` | `radsimp` | `rectform` | `rewrite` | `simplify`

More About

- "If You Want to Simplify Results Further"

sin

Sine function

Syntax

```
sin(x)
```

Description

`sin(x)` represents the sine function.

The arguments have to be specified in radians, not in degrees. E.g., use π to specify an angle of 180^o.

All trigonometric functions are defined for complex arguments.

Floating point values are returned for floating-point arguments. Floating point intervals are returned for floating-point interval arguments. Unevaluated function calls are returned for most exact arguments.

Translations by integer multiples of π are eliminated from the argument. Further, arguments that are rational multiples of π lead to simplified results; symmetry relations are used to rewrite the result using an argument from the standard interval $\left[0, \frac{\pi}{2}\right)$.

Explicit expressions are returned for the following arguments:

$$0, \frac{\pi}{2}, \frac{\pi}{3}, \frac{\pi}{4}, \frac{\pi}{5}, \frac{2\pi}{5}, \frac{\pi}{6}, \frac{\pi}{8}, \frac{3\pi}{8}, \frac{\pi}{10}, \frac{3\pi}{10}, \frac{\pi}{12}, \frac{5\pi}{12}.$$

Cf. “Example 2” on page 1-1875.

The result is rewritten in terms of hyperbolic functions, if the argument is a rational multiple of `I`. Cf. “Example 3” on page 1-1875.

The functions `expand` and `combine` implement the addition theorems for the trigonometric functions. Cf. “Example 4” on page 1-1876.

The trigonometric functions do not respond to properties set via `assume`. Use `simplify` to take such properties into account. Cf. “Example 4” on page 1-1876.

Use `rewrite` to rewrite expressions involving `tan` and `cot` in terms of `sin` and `cos`. Cf. “Example 5” on page 1-1877.

The inverse function is implemented by `arcsin`. Cf. “Example 6” on page 1-1877.

The float attributes are kernel functions, i.e., floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
sin(PI), cos(1), tan(5 + I), csc(PI/2), sec(PI/11), cot(PI/8)
```

$$0,\ \cos(1),\ \tan(5+i),\ 1,\ \frac{1}{\cos\left(\frac{\pi}{11}\right)},\ \sqrt{2}+1$$

```
sin(-x), cos(x + PI), tan(x^2 - 4)
```

$$-\sin(x),\ -\cos(x),\ \tan(x^2-4)$$

Floating point values are computed for floating-point arguments:

```
sin(123.4), cos(5.6 + 7.8*I), cot(1.0/10^20)
```

$$-0.7693905459,\ 946.4239673 + 770.3351731\,i,\ 1.0\ 10^{20}$$

Floating point intervals are computed for interval arguments:

```
sin(0 ... 1), cos(20 ... 30), tan(0 ... 5)
```

$$0.0 \ldots 0.8414709849,\ -1.0 \ldots 1.0,\ \mathrm{RD\_NINF} \ldots \mathrm{RD\_INF}$$

For the functions with discontinuities, the result may be a union of intervals:

```
csc(-1 ... 1), tan(1 ... 2)
```

$$\mathrm{RD\_NINF} \ldots -1.188395105 \cup 1.188395105 \ldots \mathrm{RD\_INF},\ \mathrm{RD\_NINF} \ldots -2.185039863 \cup 1.557407724 \ldots \mathrm{RD\_INF}$$

Example 2

Some special values are implemented:

```
sin(PI/10), cos(2*PI/5), tan(123/8*PI), cot(-PI/12)
```

$$\frac{\sqrt{5}}{4} - \frac{1}{4},\ \frac{\sqrt{5}}{4} - \frac{1}{4},\ \sqrt{2} + 1,\ -\sqrt{3} - 2$$

Translations by integer multiples of π are eliminated from the argument:

```
sin(x + 10*PI), cos(3 - PI), tan(x + PI), cot(2 - 10^100*PI)
```

$$\sin(x),\ -\cos(3),\ \tan(x),\ \cot(2)$$

All arguments that are rational multiples of π are transformed to arguments from the interval $\left[0, \frac{\pi}{2}\right)$:

```
sin(4/7*PI), cos(-20*PI/9), tan(123/11*PI), cot(-PI/13)
```

$$\sin\left(\frac{3\,\pi}{7}\right),\ \cos\left(\frac{2\,\pi}{9}\right),\ \tan\left(\frac{2\,\pi}{11}\right),\ -\cot\left(\frac{\pi}{13}\right)$$

Example 3

Arguments that are rational multiples of `I` are rewritten in terms of hyperbolicfunctions:

```
sin(5*I), cos(5/4*I), tan(-3*I)
```

$$\sinh(5)\,i,\ \cosh\left(\frac{5}{4}\right),\ -\tanh(3)\,i$$

For other complex arguments, use `expand` to rewrite the result:

```
sin(5*I + 2*PI/3), cos(PI/4 - 5/4*I), tan(-3*I + PI/2)
```

$$\sin\left(\frac{2\,\pi}{3}+5\,i\right),\ \cos\left(\frac{\pi}{4}-\frac{5\,i}{4}\right),\ \tan\left(\frac{\pi}{2}-3\,i\right)$$

```
expand(sin(5*I + 2*PI/3)), expand(cos(5/4*I - PI/4)),
expand(tan(-3*I + PI/2))
```

$$\frac{\sqrt{3}\,\cosh(5)}{2}-\frac{\sinh(5)\,i}{2},\ \frac{\sqrt{2}\,\cosh\left(\frac{5}{4}\right)}{2}+\frac{\sqrt{2}\,\sinh\left(\frac{5}{4}\right)i}{2},\ -\frac{i}{\tanh(3)}$$

Example 4

The `expand` function implements the addition theorems:

```
expand(sin(x + PI/2)), expand(cos(x + y))
```

$$\cos(x),\ \cos(x)\cos(y)-\sin(x)\sin(y)$$

The `combine` function uses these theorems in the other direction, trying to rewrite products of trigonometric functions:

```
combine(sin(x)*sin(y), sincos)
```

$$\frac{\cos(x-y)}{2}-\frac{\cos(x+y)}{2}$$

The trigonometric functions do not immediately respond to properties set via `assume`:

```
assume(n, Type::Integer): sin(n*PI), cos(n*PI)
```

$$\sin(\pi\, n),\ \cos(\pi\, n)$$

Use `simplify` to take such properties into account:

```
simplify(sin(n*PI)), simplify(cos(n*PI))
```

$$0,\ (-1)^n$$

```
assume(n, Type::Odd): sin(n*PI + x), simplify(sin(n*PI + x))
```

$$\sin(x+\pi\, n),\ -\sin(x)$$

```
y := cos(x + n*PI) + cos(x - n*PI):  y, simplify(y)
```

$$\cos(x+\pi\, n)+\cos(x-\pi\, n),\ -2\cos(x)$$

```
delete n, y:
```

Example 5

Various relations exist between the trigonometric functions:

```
csc(x), sec(x)
```

$$\frac{1}{\sin(x)},\ \frac{1}{\cos(x)}$$

Use `rewrite` to obtain a representation in terms of a specific target function:

```
rewrite(tan(x)*exp(2*I*x), sincos), rewrite(sin(x), cot)
```

$$\frac{\sin(x)\,(\cos(2\,x)+\sin(2\,x)\,\mathrm{i})}{\cos(x)},\ \frac{2\cot\left(\frac{x}{2}\right)}{\cot\left(\frac{x}{2}\right)^2+1}$$

Example 6

The inverse functions are implemented by `arcsin`, `arccos` etc.:

```
sin(arcsin(x)), sin(arccos(x)), cos(arctan(x))
```

$$x,\ \sqrt{1-x^2},\ \frac{1}{\sqrt{x^2+1}}$$

Note that `arcsin(sin(x))` does not necessarily yield `x`, because `arcsin` produces values with real parts in the interval $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$:

```
arcsin(sin(3)), arcsin(sin(1.6 + I))
```

$$\pi - 3,\ 1.541592654 - 1.0\,i$$

Example 7

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the trigonometric functions:

```
diff(sin(x^2), x), float(sin(3)*cot(5 + I))
```

$$2\,x\cos(x^2),\ -0.01668502608 - 0.1112351327\,i$$

```
limit(x*sin(x)/tan(x^2), x = 0)
```

$$1$$

```
series((tan(sin(x)) - sin(tan(x)))/sin(x^7), x = 0)
```

$$\frac{1}{30} + \frac{29\,x^2}{756} + \frac{1913\,x^4}{75600} + O(x^6)$$

Parameters

x

An arithmetical expression or a floating-point interval

Return Values

Arithmetical expression or a floating-point interval

Overloaded By

x

See Also

MuPAD Functions

`arccos` | `arccot` | `arccsc` | `arcsec` | `arcsin` | `arctan` | `cos` | `cot` | `csc` | `sec` | `tan`

cos

Cosine function

Syntax

```
cos(x)
```

Description

`cos(x)` represents the cosine function.

The arguments have to be specified in radians, not in degrees. E.g., use π to specify an angle of 180^o.

All trigonometric functions are defined for complex arguments.

Floating point values are returned for floating-point arguments. Floating point intervals are returned for floating-point interval arguments. Unevaluated function calls are returned for most exact arguments.

Translations by integer multiples of π are eliminated from the argument. Further, arguments that are rational multiples of π lead to simplified results; symmetry relations are used to rewrite the result using an argument from the standard interval $\left[0, \frac{\pi}{2}\right)$.

Explicit expressions are returned for the following arguments:

$$0, \frac{\pi}{2}, \frac{\pi}{3}, \frac{\pi}{4}, \frac{\pi}{5}, \frac{2\pi}{5}, \frac{\pi}{6}, \frac{\pi}{8}, \frac{3\pi}{8}, \frac{\pi}{10}, \frac{3\pi}{10}, \frac{\pi}{12}, \frac{5\pi}{12}.$$

Cf. “Example 2” on page 1-1882.

The result is rewritten in terms of hyperbolic functions, if the argument is a rational multiple of `I`. Cf. “Example 3” on page 1-1882.

The functions `expand` and `combine` implement the addition theorems for the trigonometric functions. Cf. “Example 4” on page 1-1883.

The trigonometric functions do not respond to properties set via `assume`. Use `simplify` to take such properties into account. Cf. “Example 4” on page 1-1883.

Use `rewrite` to rewrite expressions involving `tan` and `cot` in terms of `sin` and `cos`. Cf. “Example 5” on page 1-1884.

The inverse function is implemented by `arccos`. Cf. “Example 6” on page 1-1884.

The float attributes are kernel functions, i.e., floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
sin(PI), cos(1), tan(5 + I), csc(PI/2), sec(PI/11), cot(PI/8)
```

$$0,\ \cos(1),\ \tan(5+i),\ 1,\ \frac{1}{\cos\left(\frac{\pi}{11}\right)},\ \sqrt{2}+1$$

```
sin(-x), cos(x + PI), tan(x^2 - 4)
```

$$-\sin(x),\ -\cos(x),\ \tan(x^2-4)$$

Floating point values are computed for floating-point arguments:

```
sin(123.4), cos(5.6 + 7.8*I), cot(1.0/10^20)
```

$$-0.7693905459,\ 946.4239673 + 770.3351731\ i,\ 1.0\ 10^{20}$$

Floating point intervals are computed for interval arguments:

```
sin(0 ... 1), cos(20 ... 30), tan(0 ... 5)
```

$$0.0 \ldots 0.8414709849,\ -1.0 \ldots 1.0,\ \mathrm{RD\_NINF} \ldots \mathrm{RD\_INF}$$

For the functions with discontinuities, the result may be a union of intervals:

```
csc(-1 ... 1), tan(1 ... 2)
```

$$\mathrm{RD\_NINF} \ldots -1.188395105 \cup 1.188395105 \ldots \mathrm{RD\_INF},\ \mathrm{RD\_NINF} \ldots -2.185039863 \cup 1.557407724 \ldots \mathrm{RD\_INF}$$

Example 2

Some special values are implemented:

```
sin(PI/10), cos(2*PI/5), tan(123/8*PI), cot(-PI/12)
```

$$\frac{\sqrt{5}}{4} - \frac{1}{4},\ \frac{\sqrt{5}}{4} - \frac{1}{4},\ \sqrt{2} + 1,\ -\sqrt{3} - 2$$

Translations by integer multiples of π are eliminated from the argument:

```
sin(x + 10*PI), cos(3 - PI), tan(x + PI), cot(2 - 10^100*PI)
```

$$\sin(x),\ -\cos(3),\ \tan(x),\ \cot(2)$$

All arguments that are rational multiples of π are transformed to arguments from the interval $\left[0, \frac{\pi}{2}\right)$:

```
sin(4/7*PI), cos(-20*PI/9), tan(123/11*PI), cot(-PI/13)
```

$$\sin\left(\frac{3\,\pi}{7}\right),\ \cos\left(\frac{2\,\pi}{9}\right),\ \tan\left(\frac{2\,\pi}{11}\right),\ -\cot\left(\frac{\pi}{13}\right)$$

Example 3

Arguments that are rational multiples of `I` are rewritten in terms of hyperbolicfunctions:

```
sin(5*I), cos(5/4*I), tan(-3*I)
```

$$\sinh(5)\,i,\ \cosh\left(\frac{5}{4}\right),\ -\tanh(3)\,i$$

For other complex arguments, use `expand` to rewrite the result:

```
sin(5*I + 2*PI/3), cos(PI/4 - 5/4*I), tan(-3*I + PI/2)
```

$$\sin\left(\frac{2\,\pi}{3}+5\,i\right),\ \cos\left(\frac{\pi}{4}-\frac{5\,i}{4}\right),\ \tan\left(\frac{\pi}{2}-3\,i\right)$$

```
expand(sin(5*I + 2*PI/3)), expand(cos(5/4*I - PI/4)),
expand(tan(-3*I + PI/2))
```

$$\frac{\sqrt{3}\,\cosh(5)}{2}-\frac{\sinh(5)\,i}{2},\ \frac{\sqrt{2}\,\cosh\left(\frac{5}{4}\right)}{2}+\frac{\sqrt{2}\,\sinh\left(\frac{5}{4}\right)i}{2},\ -\frac{i}{\tanh(3)}$$

Example 4

The `expand` function implements the addition theorems:

```
expand(sin(x + PI/2)), expand(cos(x + y))
```

$$\cos(x),\ \cos(x)\cos(y)-\sin(x)\sin(y)$$

The `combine` function uses these theorems in the other direction, trying to rewrite products of trigonometric functions:

```
combine(sin(x)*sin(y), sincos)
```

$$\frac{\cos(x-y)}{2}-\frac{\cos(x+y)}{2}$$

The trigonometric functions do not immediately respond to properties set via `assume`:

```
assume(n, Type::Integer): sin(n*PI), cos(n*PI)
```

$$\sin(\pi\, n),\ \cos(\pi\, n)$$

Use `simplify` to take such properties into account:

```
simplify(sin(n*PI)), simplify(cos(n*PI))
```

$$0,\ (-1)^n$$

```
assume(n, Type::Odd): sin(n*PI + x), simplify(sin(n*PI + x))
```

$$\sin(x + \pi\, n),\ -\sin(x)$$

```
y := cos(x + n*PI) + cos(x - n*PI):  y, simplify(y)
```

$$\cos(x + \pi\, n) + \cos(x - \pi\, n),\ -2\cos(x)$$

```
delete n, y:
```

Example 5

Various relations exist between the trigonometric functions:

```
csc(x), sec(x)
```

$$\frac{1}{\sin(x)},\ \frac{1}{\cos(x)}$$

Use `rewrite` to obtain a representation in terms of a specific target function:

```
rewrite(tan(x)*exp(2*I*x), sincos), rewrite(sin(x), cot)
```

$$\frac{\sin(x)\,(\cos(2\,x) + \sin(2\,x)\,\mathrm{i})}{\cos(x)},\ \frac{2\cot\left(\frac{x}{2}\right)}{\cot\left(\frac{x}{2}\right)^2 + 1}$$

Example 6

The inverse functions are implemented by `arcsin`, `arccos` etc.:

```
sin(arcsin(x)), sin(arccos(x)), cos(arctan(x))
```

$$x,\ \sqrt{1-x^2},\ \frac{1}{\sqrt{x^2+1}}$$

Note that `arcsin(sin(x))` does not necessarily yield `x`, because `arcsin` produces values with real parts in the interval $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$:

```
arcsin(sin(3)), arcsin(sin(1.6 + I))
```

$$\pi - 3,\ 1.541592654 - 1.0\,i$$

Example 7

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the trigonometric functions:

```
diff(sin(x^2), x), float(sin(3)*cot(5 + I))
```

$$2\,x\cos(x^2),\ -0.01668502608 - 0.1112351327\,i$$

```
limit(x*sin(x)/tan(x^2), x = 0)
```

$$1$$

```
series((tan(sin(x)) - sin(tan(x)))/sin(x^7), x = 0)
```

$$\frac{1}{30} + \frac{29\,x^2}{756} + \frac{1913\,x^4}{75600} + O(x^6)$$

Parameters

x

An arithmetical expression or a floating-point interval

Return Values

Arithmetical expression or a floating-point interval

Overloaded By

x

See Also

MuPAD Functions

`arccos` | `arccot` | `arccsc` | `arcsec` | `arcsin` | `arctan` | `cot` | `csc` | `sec` | `sin` | `tan`

tan

Tangent function

Syntax

```
tan(x)
```

Description

`tan(x)` represents the tangent function `sin(x)/cos(x)`.

The arguments have to be specified in radians, not in degrees. E.g., use π to specify an angle of 180^o.

All trigonometric functions are defined for complex arguments.

Floating point values are returned for floating-point arguments. Floating point intervals are returned for floating-point interval arguments. Unevaluated function calls are returned for most exact arguments.

Translations by integer multiples of π are eliminated from the argument. Further, arguments that are rational multiples of π lead to simplified results; symmetry relations are used to rewrite the result using an argument from the standard interval $\left[0, \frac{\pi}{2}\right)$.

Explicit expressions are returned for the following arguments:

$$0, \frac{\pi}{2}, \frac{\pi}{3}, \frac{\pi}{4}, \frac{\pi}{5}, \frac{2\pi}{5}, \frac{\pi}{6}, \frac{\pi}{8}, \frac{3\pi}{8}, \frac{\pi}{10}, \frac{3\pi}{10}, \frac{\pi}{12}, \frac{5\pi}{12}.$$

Cf. “Example 2” on page 1-1889.

The result is rewritten in terms of hyperbolic functions, if the argument is a rational multiple of `I`. Cf. “Example 3” on page 1-1889.

The functions `expand` and `combine` implement the addition theorems for the trigonometric functions. Cf. “Example 4” on page 1-1890.

The trigonometric functions do not respond to properties set via `assume`. Use `simplify` to take such properties into account. Cf. “Example 4” on page 1-1890.

Use `rewrite` to rewrite expressions involving `tan` and `cot` in terms of `sin` and `cos`. Cf. “Example 5” on page 1-1891.

The inverse function is implemented by `arctan`. Cf. “Example 6” on page 1-1891.

The float attributes are kernel functions, i.e., floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
sin(PI), cos(1), tan(5 + I), csc(PI/2), sec(PI/11), cot(PI/8)
```

$$0,\ \cos(1),\ \tan(5+i),\ 1,\ \frac{1}{\cos\left(\frac{\pi}{11}\right)},\ \sqrt{2}+1$$

```
sin(-x), cos(x + PI), tan(x^2 - 4)
```

$$-\sin(x),\ -\cos(x),\ \tan(x^2-4)$$

Floating point values are computed for floating-point arguments:

```
sin(123.4), cos(5.6 + 7.8*I), cot(1.0/10^20)
```

$$-0.7693905459,\ 946.4239673 + 770.3351731\ i,\ 1.0\ 10^{20}$$

Floating point intervals are computed for interval arguments:

```
sin(0 ... 1), cos(20 ... 30), tan(0 ... 5)
```

$$0.0 \ldots 0.8414709849,\ -1.0 \ldots 1.0,\ \mathrm{RD\_NINF} \ldots \mathrm{RD\_INF}$$

For the functions with discontinuities, the result may be a union of intervals:

```
csc(-1 ... 1), tan(1 ... 2)
```

$$\mathrm{RD\_NINF} \ldots -1.188395105 \cup 1.188395105 \ldots \mathrm{RD\_INF},\ \mathrm{RD\_NINF} \ldots -2.185039863 \cup 1.557407724 \ldots \mathrm{RD\_INF}$$

Example 2

Some special values are implemented:

```
sin(PI/10), cos(2*PI/5), tan(123/8*PI), cot(-PI/12)
```

$$\frac{\sqrt{5}}{4} - \frac{1}{4},\ \frac{\sqrt{5}}{4} - \frac{1}{4},\ \sqrt{2} + 1,\ -\sqrt{3} - 2$$

Translations by integer multiples of π are eliminated from the argument:

```
sin(x + 10*PI), cos(3 - PI), tan(x + PI), cot(2 - 10^100*PI)
```

$$\sin(x),\ -\cos(3),\ \tan(x),\ \cot(2)$$

All arguments that are rational multiples of π are transformed to arguments from the interval $\left[0, \frac{\pi}{2}\right)$:

```
sin(4/7*PI), cos(-20*PI/9), tan(123/11*PI), cot(-PI/13)
```

$$\sin\left(\frac{3\,\pi}{7}\right),\ \cos\left(\frac{2\,\pi}{9}\right),\ \tan\left(\frac{2\,\pi}{11}\right),\ -\cot\left(\frac{\pi}{13}\right)$$

Example 3

Arguments that are rational multiples of `I` are rewritten in terms of hyperbolicfunctions:

```
sin(5*I), cos(5/4*I), tan(-3*I)
```

$$\sinh(5)\,i,\ \cosh\left(\frac{5}{4}\right),\ -\tanh(3)\,i$$

For other complex arguments, use `expand` to rewrite the result:

```
sin(5*I + 2*PI/3), cos(PI/4 - 5/4*I), tan(-3*I + PI/2)
```

$$\sin\left(\frac{2\,\pi}{3}+5\,i\right),\ \cos\left(\frac{\pi}{4}-\frac{5\,i}{4}\right),\ \tan\left(\frac{\pi}{2}-3\,i\right)$$

```
expand(sin(5*I + 2*PI/3)), expand(cos(5/4*I - PI/4)),
expand(tan(-3*I + PI/2))
```

$$\frac{\sqrt{3}\,\cosh(5)}{2}-\frac{\sinh(5)\,i}{2},\ \frac{\sqrt{2}\,\cosh\left(\frac{5}{4}\right)}{2}+\frac{\sqrt{2}\,\sinh\left(\frac{5}{4}\right)i}{2},\ -\frac{i}{\tanh(3)}$$

Example 4

The `expand` function implements the addition theorems:

```
expand(sin(x + PI/2)), expand(cos(x + y))
```

$$\cos(x),\ \cos(x)\cos(y)-\sin(x)\sin(y)$$

The `combine` function uses these theorems in the other direction, trying to rewrite products of trigonometric functions:

```
combine(sin(x)*sin(y), sincos)
```

$$\frac{\cos(x-y)}{2}-\frac{\cos(x+y)}{2}$$

The trigonometric functions do not immediately respond to properties set via `assume`:

```
assume(n, Type::Integer): sin(n*PI), cos(n*PI)
```

$$\sin(\pi\, n),\ \cos(\pi\, n)$$

Use `simplify` to take such properties into account:

```
simplify(sin(n*PI)), simplify(cos(n*PI))
```

$$0,\ (-1)^n$$

```
assume(n, Type::Odd): sin(n*PI + x), simplify(sin(n*PI + x))
```

$$\sin(x+\pi\, n),\ -\sin(x)$$

```
y := cos(x + n*PI) + cos(x - n*PI):  y, simplify(y)
```

$$\cos(x+\pi\, n)+\cos(x-\pi\, n),\ -2\cos(x)$$

```
delete n, y:
```

Example 5

Various relations exist between the trigonometric functions:

```
csc(x), sec(x)
```

$$\frac{1}{\sin(x)},\ \frac{1}{\cos(x)}$$

Use `rewrite` to obtain a representation in terms of a specific target function:

```
rewrite(tan(x)*exp(2*I*x), sincos), rewrite(sin(x), cot)
```

$$\frac{\sin(x)\,(\cos(2\,x)+\sin(2\,x)\,\mathrm{i})}{\cos(x)},\ \frac{2\cot\left(\frac{x}{2}\right)}{\cot\left(\frac{x}{2}\right)^2+1}$$

Example 6

The inverse functions are implemented by `arcsin`, `arccos` etc.:

```
sin(arcsin(x)), sin(arccos(x)), cos(arctan(x))
```

$$x,\ \sqrt{1-x^2},\ \frac{1}{\sqrt{x^2+1}}$$

Note that `arcsin(sin(x))` does not necessarily yield `x`, because `arcsin` produces values with real parts in the interval $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$:

```
arcsin(sin(3)), arcsin(sin(1.6 + I))
```

$$\pi - 3,\ 1.541592654 - 1.0\,i$$

Example 7

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the trigonometric functions:

```
diff(sin(x^2), x), float(sin(3)*cot(5 + I))
```

$$2\,x\cos(x^2),\ -0.01668502608 - 0.1112351327\,i$$

```
limit(x*sin(x)/tan(x^2), x = 0)
```

$$1$$

```
series((tan(sin(x)) - sin(tan(x)))/sin(x^7), x = 0)
```

$$\frac{1}{30} + \frac{29\,x^2}{756} + \frac{1913\,x^4}{75600} + O(x^6)$$

Parameters

x

An arithmetical expression or a floating-point interval

Return Values

Arithmetical expression or a floating-point interval

Overloaded By

x

See Also

MuPAD Functions

`arccos` | `arccot` | `arccsc` | `arcsec` | `arcsin` | `arctan` | `cos` | `cot` | `csc` | `sec` | `sin`

csc

Cosecant function

Syntax

```
csc(x)
```

Description

`csc(x)` represents the cosecant function `1/sin(x)`.

The arguments have to be specified in radians, not in degrees. E.g., use π to specify an angle of 180^o.

All trigonometric functions are defined for complex arguments.

Floating point values are returned for floating-point arguments. Floating point intervals are returned for floating-point interval arguments. Unevaluated function calls are returned for most exact arguments.

Translations by integer multiples of π are eliminated from the argument. Further, arguments that are rational multiples of π lead to simplified results; symmetry relations are used to rewrite the result using an argument from the standard interval $\left[0, \frac{\pi}{2}\right)$.

Explicit expressions are returned for the following arguments:

$$0, \frac{\pi}{2}, \frac{\pi}{3}, \frac{\pi}{4}, \frac{\pi}{5}, \frac{2\pi}{5}, \frac{\pi}{6}, \frac{\pi}{8}, \frac{3\pi}{8}, \frac{\pi}{10}, \frac{3\pi}{10}, \frac{\pi}{12}, \frac{5\pi}{12}.$$

Cf. “Example 2” on page 1-1896.

The result is rewritten in terms of hyperbolic functions, if the argument is a rational multiple of `I`. Cf. “Example 3” on page 1-1896.

The functions `expand` and `combine` implement the addition theorems for the trigonometric functions. Cf. “Example 4” on page 1-1897.

The trigonometric functions do not respond to properties set via `assume`. Use `simplify` to take such properties into account. Cf. “Example 4” on page 1-1897.

`csc(x)` is immediately rewritten as `1/sin(x)`. Cf. “Example 5” on page 1-1898.

The inverse function is implemented by `arccsc`. Cf. “Example 6” on page 1-1898.

The float attributes are kernel functions, i.e., floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
sin(PI), cos(1), tan(5 + I), csc(PI/2), sec(PI/11), cot(PI/8)
```

$$0,\ \cos(1),\ \tan(5+i),\ 1,\ \frac{1}{\cos\left(\frac{\pi}{11}\right)},\ \sqrt{2}+1$$

```
sin(-x), cos(x + PI), tan(x^2 - 4)
```

$$-\sin(x),\ -\cos(x),\ \tan(x^2-4)$$

Floating point values are computed for floating-point arguments:

```
sin(123.4), cos(5.6 + 7.8*I), cot(1.0/10^20)
```

$$-0.7693905459,\ 946.4239673 + 770.3351731\ i,\ 1.0\ 10^{20}$$

Floating point intervals are computed for interval arguments:

```
sin(0 ... 1), cos(20 ... 30), tan(0 ... 5)
```

$$0.0 \ldots 0.8414709849,\ -1.0 \ldots 1.0,\ \mathrm{RD\_NINF} \ldots \mathrm{RD\_INF}$$

For the functions with discontinuities, the result may be a union of intervals:

```
csc(-1 ... 1), tan(1 ... 2)
```

$$\mathrm{RD\_NINF} \ldots -1.188395105 \cup 1.188395105 \ldots \mathrm{RD\_INF},\ \mathrm{RD\_NINF} \ldots -2.185039863 \cup 1.557407724 \ldots \mathrm{RD\_INF}$$

Example 2

Some special values are implemented:

```
sin(PI/10), cos(2*PI/5), tan(123/8*PI), cot(-PI/12)
```

$$\frac{\sqrt{5}}{4} - \frac{1}{4},\ \frac{\sqrt{5}}{4} - \frac{1}{4},\ \sqrt{2} + 1,\ -\sqrt{3} - 2$$

Translations by integer multiples of π are eliminated from the argument:

```
sin(x + 10*PI), cos(3 - PI), tan(x + PI), cot(2 - 10^100*PI)
```

$$\sin(x),\ -\cos(3),\ \tan(x),\ \cot(2)$$

All arguments that are rational multiples of π are transformed to arguments from the interval $\left[0, \frac{\pi}{2}\right)$:

```
sin(4/7*PI), cos(-20*PI/9), tan(123/11*PI), cot(-PI/13)
```

$$\sin\left(\frac{3\,\pi}{7}\right),\ \cos\left(\frac{2\,\pi}{9}\right),\ \tan\left(\frac{2\,\pi}{11}\right),\ -\cot\left(\frac{\pi}{13}\right)$$

Example 3

Arguments that are rational multiples of `I` are rewritten in terms of hyperbolicfunctions:

```
sin(5*I), cos(5/4*I), tan(-3*I)
```

$$\sinh(5)\,i,\ \cosh\left(\frac{5}{4}\right),\ -\tanh(3)\,i$$

For other complex arguments, use `expand` to rewrite the result:

```
sin(5*I + 2*PI/3), cos(PI/4 - 5/4*I), tan(-3*I + PI/2)
```

$$\sin\left(\frac{2\,\pi}{3}+5\,i\right),\ \cos\left(\frac{\pi}{4}-\frac{5\,i}{4}\right),\ \tan\left(\frac{\pi}{2}-3\,i\right)$$

```
expand(sin(5*I + 2*PI/3)), expand(cos(5/4*I - PI/4)),
expand(tan(-3*I + PI/2))
```

$$\frac{\sqrt{3}\,\cosh(5)}{2}-\frac{\sinh(5)\,i}{2},\ \frac{\sqrt{2}\,\cosh\left(\frac{5}{4}\right)}{2}+\frac{\sqrt{2}\,\sinh\left(\frac{5}{4}\right)i}{2},\ -\frac{i}{\tanh(3)}$$

Example 4

The `expand` function implements the addition theorems:

```
expand(sin(x + PI/2)), expand(cos(x + y))
```

$$\cos(x),\ \cos(x)\cos(y)-\sin(x)\sin(y)$$

The `combine` function uses these theorems in the other direction, trying to rewrite products of trigonometric functions:

```
combine(sin(x)*sin(y), sincos)
```

$$\frac{\cos(x-y)}{2}-\frac{\cos(x+y)}{2}$$

The trigonometric functions do not immediately respond to properties set via `assume`:

```
assume(n, Type::Integer): sin(n*PI), cos(n*PI)
```

$$\sin(\pi\, n),\ \cos(\pi\, n)$$

Use `simplify` to take such properties into account:

```
simplify(sin(n*PI)), simplify(cos(n*PI))
```

$$0,\ (-1)^n$$

```
assume(n, Type::Odd): sin(n*PI + x), simplify(sin(n*PI + x))
```

$$\sin(x + \pi\, n),\ -\sin(x)$$

```
y := cos(x + n*PI) + cos(x - n*PI):  y, simplify(y)
```

$$\cos(x + \pi\, n) + \cos(x - \pi\, n),\ -2\cos(x)$$

```
delete n, y:
```

Example 5

Various relations exist between the trigonometric functions:

```
csc(x), sec(x)
```

$$\frac{1}{\sin(x)},\ \frac{1}{\cos(x)}$$

Use `rewrite` to obtain a representation in terms of a specific target function:

```
rewrite(tan(x)*exp(2*I*x), sincos), rewrite(sin(x), cot)
```

$$\frac{\sin(x)\,(\cos(2\,x) + \sin(2\,x)\,\mathrm{i})}{\cos(x)},\ \frac{2\cot\left(\frac{x}{2}\right)}{\cot\left(\frac{x}{2}\right)^2 + 1}$$

Example 6

The inverse functions are implemented by `arcsin`, `arccos` etc.:

```
sin(arcsin(x)), sin(arccos(x)), cos(arctan(x))
```

$$x,\ \sqrt{1-x^2},\ \frac{1}{\sqrt{x^2+1}}$$

Note that `arcsin(sin(x))` does not necessarily yield `x`, because `arcsin` produces values with real parts in the interval $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$:

```
arcsin(sin(3)), arcsin(sin(1.6 + I))
```

$$\pi - 3,\ 1.541592654 - 1.0\,i$$

Example 7

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the trigonometric functions:

```
diff(sin(x^2), x), float(sin(3)*cot(5 + I))
```

$$2\,x\cos(x^2),\ -0.01668502608 - 0.1112351327\,i$$

```
limit(x*sin(x)/tan(x^2), x = 0)
```

$$1$$

```
series((tan(sin(x)) - sin(tan(x)))/sin(x^7), x = 0)
```

$$\frac{1}{30} + \frac{29\,x^2}{756} + \frac{1913\,x^4}{75600} + O(x^6)$$

Parameters

x

An arithmetical expression or a floating-point interval

Return Values

Arithmetical expression or a floating-point interval

Overloaded By

x

See Also

MuPAD Functions

`arccos` | `arccot` | `arccsc` | `arcsec` | `arcsin` | `arctan` | `cos` | `cot` | `sec` | `sin` | `tan`

sec

Secant function

Syntax

```
sec(x)
```

Description

`sec(x)` represents the secant function `1/cos(x)`.

The arguments have to be specified in radians, not in degrees. E.g., use π to specify an angle of 180^o.

All trigonometric functions are defined for complex arguments.

Floating point values are returned for floating-point arguments. Floating point intervals are returned for floating-point interval arguments. Unevaluated function calls are returned for most exact arguments.

Translations by integer multiples of π are eliminated from the argument. Further, arguments that are rational multiples of π lead to simplified results; symmetry relations are used to rewrite the result using an argument from the standard interval $\left[0, \frac{\pi}{2}\right)$.

Explicit expressions are returned for the following arguments:

$$0, \frac{\pi}{2}, \frac{\pi}{3}, \frac{\pi}{4}, \frac{\pi}{5}, \frac{2\pi}{5}, \frac{\pi}{6}, \frac{\pi}{8}, \frac{3\pi}{8}, \frac{\pi}{10}, \frac{3\pi}{10}, \frac{\pi}{12}, \frac{5\pi}{12}.$$

Cf. “Example 2” on page 1-1903.

The result is rewritten in terms of hyperbolic functions, if the argument is a rational multiple of `I`. Cf. “Example 3” on page 1-1903.

The functions `expand` and `combine` implement the addition theorems for the trigonometric functions. Cf. “Example 4” on page 1-1904.

The trigonometric functions do not respond to properties set via `assume`. Use `simplify` to take such properties into account. Cf. “Example 4” on page 1-1904.

`sec(x)` is immediately rewritten as `1/cos(x)`. Cf. “Example 5” on page 1-1905.

The inverse function is implemented by `arcsec`. Cf. “Example 6” on page 1-1905.

The float attributes are kernel functions, i.e., floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
sin(PI), cos(1), tan(5 + I), csc(PI/2), sec(PI/11), cot(PI/8)
```

$$0,\ \cos(1),\ \tan(5+i),\ 1,\ \frac{1}{\cos\left(\frac{\pi}{11}\right)},\ \sqrt{2}+1$$

```
sin(-x), cos(x + PI), tan(x^2 - 4)
```

$$-\sin(x),\ -\cos(x),\ \tan(x^2-4)$$

Floating point values are computed for floating-point arguments:

```
sin(123.4), cos(5.6 + 7.8*I), cot(1.0/10^20)
```

$$-0.7693905459,\ 946.4239673 + 770.3351731\ i,\ 1.0\ 10^{20}$$

Floating point intervals are computed for interval arguments:

```
sin(0 ... 1), cos(20 ... 30), tan(0 ... 5)
```

$$0.0 \ldots 0.8414709849,\ -1.0 \ldots 1.0,\ \mathrm{RD\_NINF} \ldots \mathrm{RD\_INF}$$

For the functions with discontinuities, the result may be a union of intervals:

```
csc(-1 ... 1), tan(1 ... 2)
```

$$\mathrm{RD\_NINF} \ldots -1.188395105 \cup 1.188395105 \ldots \mathrm{RD\_INF},\ \mathrm{RD\_NINF} \ldots -2.185039863 \cup 1.557407724 \ldots \mathrm{RD\_INF}$$

Example 2

Some special values are implemented:

```
sin(PI/10), cos(2*PI/5), tan(123/8*PI), cot(-PI/12)
```

$$\frac{\sqrt{5}}{4} - \frac{1}{4},\ \frac{\sqrt{5}}{4} - \frac{1}{4},\ \sqrt{2} + 1,\ -\sqrt{3} - 2$$

Translations by integer multiples of π are eliminated from the argument:

```
sin(x + 10*PI), cos(3 - PI), tan(x + PI), cot(2 - 10^100*PI)
```

$$\sin(x),\ -\cos(3),\ \tan(x),\ \cot(2)$$

All arguments that are rational multiples of π are transformed to arguments from the interval $\left[0, \frac{\pi}{2}\right)$:

```
sin(4/7*PI), cos(-20*PI/9), tan(123/11*PI), cot(-PI/13)
```

$$\sin\left(\frac{3\,\pi}{7}\right),\ \cos\left(\frac{2\,\pi}{9}\right),\ \tan\left(\frac{2\,\pi}{11}\right),\ -\cot\left(\frac{\pi}{13}\right)$$

Example 3

Arguments that are rational multiples of `I` are rewritten in terms of hyperbolicfunctions:

```
sin(5*I), cos(5/4*I), tan(-3*I)
```

$$\sinh(5)\,i,\ \cosh\left(\frac{5}{4}\right),\ -\tanh(3)\,i$$

For other complex arguments, use `expand` to rewrite the result:

```
sin(5*I + 2*PI/3), cos(PI/4 - 5/4*I), tan(-3*I + PI/2)
```

$$\sin\left(\frac{2\,\pi}{3}+5\,i\right),\ \cos\left(\frac{\pi}{4}-\frac{5\,i}{4}\right),\ \tan\left(\frac{\pi}{2}-3\,i\right)$$

```
expand(sin(5*I + 2*PI/3)), expand(cos(5/4*I - PI/4)),
expand(tan(-3*I + PI/2))
```

$$\frac{\sqrt{3}\,\cosh(5)}{2}-\frac{\sinh(5)\,i}{2},\ \frac{\sqrt{2}\,\cosh\left(\frac{5}{4}\right)}{2}+\frac{\sqrt{2}\,\sinh\left(\frac{5}{4}\right)i}{2},\ -\frac{i}{\tanh(3)}$$

Example 4

The `expand` function implements the addition theorems:

```
expand(sin(x + PI/2)), expand(cos(x + y))
```

$$\cos(x),\ \cos(x)\cos(y)-\sin(x)\sin(y)$$

The `combine` function uses these theorems in the other direction, trying to rewrite products of trigonometric functions:

```
combine(sin(x)*sin(y), sincos)
```

$$\frac{\cos(x-y)}{2}-\frac{\cos(x+y)}{2}$$

The trigonometric functions do not immediately respond to properties set via `assume`:

```
assume(n, Type::Integer): sin(n*PI), cos(n*PI)
```

$$\sin(\pi\, n),\ \cos(\pi\, n)$$

Use `simplify` to take such properties into account:

```
simplify(sin(n*PI)), simplify(cos(n*PI))
```

$$0,\ (-1)^n$$

```
assume(n, Type::Odd): sin(n*PI + x), simplify(sin(n*PI + x))
```

$$\sin(x+\pi\, n),\ -\sin(x)$$

```
y := cos(x + n*PI) + cos(x - n*PI):  y, simplify(y)
```

$$\cos(x+\pi\, n)+\cos(x-\pi\, n),\ -2\,\cos(x)$$

```
delete n, y:
```

Example 5

Various relations exist between the trigonometric functions:

```
csc(x), sec(x)
```

$$\frac{1}{\sin(x)},\ \frac{1}{\cos(x)}$$

Use `rewrite` to obtain a representation in terms of a specific target function:

```
rewrite(tan(x)*exp(2*I*x), sincos), rewrite(sin(x), cot)
```

$$\frac{\sin(x)\,(\cos(2\,x)+\sin(2\,x)\,\mathrm{i})}{\cos(x)},\ \frac{2\,\cot\left(\frac{x}{2}\right)}{\cot\left(\frac{x}{2}\right)^2+1}$$

Example 6

The inverse functions are implemented by `arcsin`, `arccos` etc.:

```
sin(arcsin(x)), sin(arccos(x)), cos(arctan(x))
```

$$x,\ \sqrt{1-x^2},\ \frac{1}{\sqrt{x^2+1}}$$

Note that `arcsin(sin(x))` does not necessarily yield `x`, because `arcsin` produces values with real parts in the interval $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$:

```
arcsin(sin(3)), arcsin(sin(1.6 + I))
```

$$\pi - 3,\ 1.541592654 - 1.0\,i$$

Example 7

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the trigonometric functions:

```
diff(sin(x^2), x), float(sin(3)*cot(5 + I))
```

$$2\,x\cos(x^2),\ -0.01668502608 - 0.1112351327\,i$$

```
limit(x*sin(x)/tan(x^2), x = 0)
```

$$1$$

```
series((tan(sin(x)) - sin(tan(x)))/sin(x^7), x = 0)
```

$$\frac{1}{30} + \frac{29\,x^2}{756} + \frac{1913\,x^4}{75600} + O(x^6)$$

Parameters

x

An arithmetical expression or a floating-point interval

Return Values

Arithmetical expression or a floating-point interval

Overloaded By

x

See Also

MuPAD Functions

`arccos` | `arccot` | `arccsc` | `arcsec` | `arcsin` | `arctan` | `cos` | `cot` | `csc` | `sin` | `tan`

cot

Cotangent function

Syntax

```
cot(x)
```

Description

`cot(x)` represents the cotangent function `cos(x)/sin(x)`.

The arguments have to be specified in radians, not in degrees. E.g., use π to specify an angle of 180^o.

All trigonometric functions are defined for complex arguments.

Floating point values are returned for floating-point arguments. Floating point intervals are returned for floating-point interval arguments. Unevaluated function calls are returned for most exact arguments.

Translations by integer multiples of π are eliminated from the argument. Further, arguments that are rational multiples of π lead to simplified results; symmetry relations are used to rewrite the result using an argument from the standard interval $\left[0, \frac{\pi}{2}\right)$.

Explicit expressions are returned for the following arguments:

$$0, \frac{\pi}{2}, \frac{\pi}{3}, \frac{\pi}{4}, \frac{\pi}{5}, \frac{2\,\pi}{5}, \frac{\pi}{6}, \frac{\pi}{8}, \frac{3\,\pi}{8}, \frac{\pi}{10}, \frac{3\,\pi}{10}, \frac{\pi}{12}, \frac{5\,\pi}{12}.$$

Cf. “Example 2” on page 1-1910.

The result is rewritten in terms of hyperbolic functions, if the argument is a rational multiple of `I`. Cf. “Example 3” on page 1-1910.

The functions `expand` and `combine` implement the addition theorems for the trigonometric functions. Cf. “Example 4” on page 1-1911.

The trigonometric functions do not respond to properties set via `assume`. Use `simplify` to take such properties into account. Cf. “Example 4” on page 1-1911.

Use `rewrite` to rewrite expressions involving `tan` and `cot` in terms of `sin` and `cos`. Cf. “Example 5” on page 1-1912.

The inverse function is implemented by `arccot`. Cf. “Example 6” on page 1-1912.

The float attributes are kernel functions, i.e., floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
sin(PI), cos(1), tan(5 + I), csc(PI/2), sec(PI/11), cot(PI/8)
```

$$0,\ \cos(1),\ \tan(5+i),\ 1,\ \frac{1}{\cos\left(\frac{\pi}{11}\right)},\ \sqrt{2}+1$$

```
sin(-x), cos(x + PI), tan(x^2 - 4)
```

$$-\sin(x),\ -\cos(x),\ \tan(x^2-4)$$

Floating point values are computed for floating-point arguments:

```
sin(123.4), cos(5.6 + 7.8*I), cot(1.0/10^20)
```

$$-0.7693905459,\ 946.4239673 + 770.3351731\,i,\ 1.0\ 10^{20}$$

Floating point intervals are computed for interval arguments:

```
sin(0 ... 1), cos(20 ... 30), tan(0 ... 5)
```

$$0.0 \ldots 0.8414709849,\ -1.0 \ldots 1.0,\ \mathrm{RD\_NINF} \ldots \mathrm{RD\_INF}$$

For the functions with discontinuities, the result may be a union of intervals:

```
csc(-1 ... 1), tan(1 ... 2)
```

$$\mathrm{RD\_NINF} \ldots -1.188395105 \cup 1.188395105 \ldots \mathrm{RD\_INF},\ \mathrm{RD\_NINF} \ldots -2.185039863 \cup 1.557407724 \ldots \mathrm{RD\_INF}$$

Example 2

Some special values are implemented:

```
sin(PI/10), cos(2*PI/5), tan(123/8*PI), cot(-PI/12)
```

$$\frac{\sqrt{5}}{4} - \frac{1}{4},\ \frac{\sqrt{5}}{4} - \frac{1}{4},\ \sqrt{2} + 1,\ -\sqrt{3} - 2$$

Translations by integer multiples of π are eliminated from the argument:

```
sin(x + 10*PI), cos(3 - PI), tan(x + PI), cot(2 - 10^100*PI)
```

$$\sin(x),\ -\cos(3),\ \tan(x),\ \cot(2)$$

All arguments that are rational multiples of π are transformed to arguments from the interval $\left[0, \frac{\pi}{2}\right)$:

```
sin(4/7*PI), cos(-20*PI/9), tan(123/11*PI), cot(-PI/13)
```

$$\sin\left(\frac{3\,\pi}{7}\right),\ \cos\left(\frac{2\,\pi}{9}\right),\ \tan\left(\frac{2\,\pi}{11}\right),\ -\cot\left(\frac{\pi}{13}\right)$$

Example 3

Arguments that are rational multiples of `I` are rewritten in terms of hyperbolicfunctions:

```
sin(5*I), cos(5/4*I), tan(-3*I)
```

$$\sinh(5)\,i,\ \cosh\left(\frac{5}{4}\right),\ -\tanh(3)\,i$$

For other complex arguments, use `expand` to rewrite the result:

```
sin(5*I + 2*PI/3), cos(PI/4 - 5/4*I), tan(-3*I + PI/2)
```

$$\sin\left(\frac{2\,\pi}{3}+5\,i\right),\ \cos\left(\frac{\pi}{4}-\frac{5\,i}{4}\right),\ \tan\left(\frac{\pi}{2}-3\,i\right)$$

```
expand(sin(5*I + 2*PI/3)), expand(cos(5/4*I - PI/4)),
expand(tan(-3*I + PI/2))
```

$$\frac{\sqrt{3}\,\cosh(5)}{2}-\frac{\sinh(5)\,i}{2},\ \frac{\sqrt{2}\,\cosh\left(\frac{5}{4}\right)}{2}+\frac{\sqrt{2}\,\sinh\left(\frac{5}{4}\right)i}{2},\ -\frac{i}{\tanh(3)}$$

Example 4

The `expand` function implements the addition theorems:

```
expand(sin(x + PI/2)), expand(cos(x + y))
```

$$\cos(x),\ \cos(x)\cos(y)-\sin(x)\sin(y)$$

The `combine` function uses these theorems in the other direction, trying to rewrite products of trigonometric functions:

```
combine(sin(x)*sin(y), sincos)
```

$$\frac{\cos(x-y)}{2}-\frac{\cos(x+y)}{2}$$

The trigonometric functions do not immediately respond to properties set via `assume`:

```
assume(n, Type::Integer): sin(n*PI), cos(n*PI)
```

$$\sin(\pi\, n),\ \cos(\pi\, n)$$

Use `simplify` to take such properties into account:

```
simplify(sin(n*PI)), simplify(cos(n*PI))
```

$$0,\ (-1)^n$$

```
assume(n, Type::Odd): sin(n*PI + x), simplify(sin(n*PI + x))
```

$$\sin(x+\pi\, n),\ -\sin(x)$$

```
y := cos(x + n*PI) + cos(x - n*PI):  y, simplify(y)
```

$$\cos(x+\pi\, n)+\cos(x-\pi\, n),\ -2\,\cos(x)$$

```
delete n, y:
```

Example 5

Various relations exist between the trigonometric functions:

```
csc(x), sec(x)
```

$$\frac{1}{\sin(x)},\ \frac{1}{\cos(x)}$$

Use `rewrite` to obtain a representation in terms of a specific target function:

```
rewrite(tan(x)*exp(2*I*x), sincos), rewrite(sin(x), cot)
```

$$\frac{\sin(x)\,(\cos(2\,x)+\sin(2\,x)\,\mathrm{i})}{\cos(x)},\ \frac{2\,\cot\left(\frac{x}{2}\right)}{\cot\left(\frac{x}{2}\right)^2+1}$$

Example 6

The inverse functions are implemented by `arcsin`, `arccos` etc.:

```
sin(arcsin(x)), sin(arccos(x)), cos(arctan(x))
```

$$x,\ \sqrt{1-x^2},\ \frac{1}{\sqrt{x^2+1}}$$

Note that `arcsin(sin(x))` does not necessarily yield `x`, because `arcsin` produces values with real parts in the interval $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$:

```
arcsin(sin(3)), arcsin(sin(1.6 + I))
```

$$\pi - 3,\ 1.541592654 - 1.0\,i$$

Example 7

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the trigonometric functions:

```
diff(sin(x^2), x), float(sin(3)*cot(5 + I))
```

$$2\,x\,\cos(x^2),\ -0.01668502608 - 0.1112351327\,i$$

```
limit(x*sin(x)/tan(x^2), x = 0)
```

$$1$$

```
series((tan(sin(x)) - sin(tan(x)))/sin(x^7), x = 0)
```

$$\frac{1}{30} + \frac{29\,x^2}{756} + \frac{1913\,x^4}{75600} + O(x^6)$$

Parameters

x

An arithmetical expression or a floating-point interval

Return Values

Arithmetical expression or a floating-point interval

Overloaded By

x

See Also

MuPAD Functions

`arccos` | `arccot` | `arccsc` | `arcsec` | `arcsin` | `arctan` | `cos` | `csc` | `sec` | `sin` | `tan`

sinh

Hyperbolic sine function

Syntax

```
sinh(x)
```

Description

`sinh(x)` represents the hyperbolic sine function.

This function is defined for complex arguments.

Floating point values are returned for floating-point arguments. Floating point intervals are returned for floating-point interval arguments. Unevaluated function calls are returned for most exact arguments.

Arguments that are integer multiples of $\frac{i\pi}{2}$ lead to simplified results. If the argument involves a negative numerical factor of `Type::Real`, then symmetry relations are used to make this factor positive. Cf. “Example 2” on page 1-1916.

The special values $sinh(0) = 0$, $sinh(\infty) = \infty$, $sinh(-\infty) = -\infty$ are implemented.

The functions `expand` and `combine` implement the addition theorems for the hyperbolic functions. Cf. “Example 3” on page 1-1917.

`csch(x)` is rewritten as `1/sinh(x)`. Use `expand` or `rewrite` to rewrite expressions involving `tanh` and `coth` in terms of `sinh` and `cosh`. Cf. “Example 4” on page 1-1917.

The inverse function is implemented by `arcsinh`. Cf. “Example 5” on page 1-1918.

The float attributes are kernel functions, i.e., floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
sinh(I*PI), cosh(1), tanh(5 + I), csch(PI), sech(1/11), coth(8)
```

$$0,\ \cosh(1),\ \tanh(5+i),\ \frac{1}{\sinh(\pi)},\ \frac{1}{\cosh\left(\frac{1}{11}\right)},\ \coth(8)$$

```
sinh(x), cosh(x + I*PI), tanh(x^2 - 4)
```

$$\sinh(x),\ -\cosh(x),\ \tanh(x^2-4)$$

Floating point values are computed for floating-point arguments:

```
sinh(123.4), cosh(5.6 + 7.8*I), coth(1.0/10^20)
```

$$1.953930316\ 10^{53},\ 7.295585032 + 135.0143985\ i,\ 1.0\ 10^{20}$$

For floating-point intervals, intervals enclosing the image are calculated:

```
cosh(-1 ... 1), tanh(-1 ... 1)
```

$$1.0 \ldots 1.543080635,\ -0.761594156 \ldots 0.761594156$$

For functions with discontinuities, evaluation over an interval may result in a union of intervals:

```
coth(-1 ... 1)
```

$$\text{RD\_NINF} \ldots -1.313035285 \cup 1.313035285 \ldots \text{RD\_INF}$$

Example 2

Simplifications are implemented for arguments that are integer multiples of $\frac{i\,\pi}{2}$:

```
sinh(I*PI/2), cosh(40*I*PI), tanh(-10^100*I*PI),
coth(-17/2*I*PI)
```

$$i,\ 1,\ 0,\ 0$$

Negative real numerical factors in the argument are rewritten via symmetry relations:

```
sinh(-5), cosh(-3/2*x), tanh(-x*PI/12), coth(-12/17*x*y*PI)
```

$$-\sinh(5),\ \cosh\left(\frac{3x}{2}\right),\ -\tanh\left(\frac{\pi x}{12}\right),\ -\coth\left(\frac{12\pi xy}{17}\right)$$

Example 3

The `expand` function implements the addition theorems:

```
expand(sinh(x + PI*I)), expand(cosh(x + y))
```

$$-\sinh(x),\ \cosh(x)\cosh(y)+\sinh(x)\sinh(y)$$

The `combine` function uses these theorems in the other direction, trying to rewrite products of hyperbolic functions:

```
combine(sinh(x)*sinh(y), sinhcosh)
```

$$\frac{\cosh(x+y)}{2}-\frac{\cosh(x-y)}{2}$$

Example 4

Various relations exist between the hyperbolic functions:

```
csch(x), sech(x)
```

$$\frac{1}{\sinh(x)},\ \frac{1}{\cosh(x)}$$

Use `rewrite` to obtain a representation in terms of a specific target function:

```
rewrite(tanh(x)*exp(2*x), sinhcosh), rewrite(sinh(x), tanh)
```

$$\frac{\sinh(x)\,(\cosh(2\,x)+\sinh(2\,x))}{\cosh(x)},\ -\frac{2\tanh\left(\frac{x}{2}\right)}{\tanh\left(\frac{x}{2}\right)^2-1}$$

```
rewrite(sinh(x)*coth(y), exp), rewrite(exp(x), coth)
```

$$-\frac{\left(\frac{e^{-x}}{2}-\frac{e^{x}}{2}\right)\left(e^{2\,y}+1\right)}{e^{2\,y}-1},\ \frac{\coth\left(\frac{x}{2}\right)+1}{\coth\left(\frac{x}{2}\right)-1}$$

Example 5

The inverse functions are implemented by `arcsinh`, `arccosh` etc.:

```
sinh(arcsinh(x)), sinh(arccosh(x)), cosh(arctanh(x))
```

$$x,\ \sqrt{x^2-1},\ \frac{1}{\sqrt{1-x}\,\sqrt{x+1}}$$

Note that `arcsinh(sinh(x))` does not necessarily yield `x`, because `arcsinh` produces values with imaginary parts in the interval $\left[-\frac{\pi}{2},\frac{\pi}{2}\right]$:

```
arcsinh(sinh(3)), arcsinh(sinh(1.6 + 100*I))
```

$$3,\ 1.6-0.5309649149\,i$$

Example 6

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the hyperbolic functions:

```
diff(sinh(x^2), x), float(sinh(3)*coth(5 + I))
```

$$2\,x\,\cosh(x^2),\ 10.01749636 - 0.0008270853591\,i$$

```
limit(x*sinh(x)/tanh(x^2), x = 0)
```

$$1$$

```
series((tanh(sinh(x)) - sinh(tanh(x)))/sinh(x^7), x = 0)
```

$$-\frac{1}{30} + \frac{29\,x^2}{756} - \frac{1913\,x^4}{75600} + O(x^6)$$

```
series(tanh(x), x = infinity)
```

$$1 - 2\,e^{-2x} + 2\,e^{-4x} - 2\,e^{-6x} + 2\,e^{-8x} - 2\,e^{-10x} + O\left(e^{-12x}\right)$$

Parameters

x

An arithmetical expression or a floating-point interval

Return Values

Arithmetical expression or a floating-point interval

Overloaded By

x

See Also

MuPAD Functions

`arccosh` | `arccoth` | `arccsch` | `arcsech` | `arcsinh` | `arctanh` | `cosh` | `coth` | `csch` | `sech` | `tanh`

cosh

Hyperbolic cosine function

Syntax

```
cosh(x)
```

Description

`cosh(x)` represents the hyperbolic cosine function.

This function is defined for complex arguments.

Floating point values are returned for floating-point arguments. Floating point intervals are returned for floating-point interval arguments. Unevaluated function calls are returned for most exact arguments.

Arguments that are integer multiples of $\frac{i\pi}{2}$ lead to simplified results. If the argument involves a negative numerical factor of `Type::Real`, then symmetry relations are used to make this factor positive. Cf. “Example 2” on page 1-1921.

The special values $\cosh(0) = 1$, $\cosh(\infty) = \infty$, $\cosh(-\infty) = \infty$ are implemented.

The functions `expand` and `combine` implement the addition theorems for the hyperbolic functions. Cf. “Example 3” on page 1-1922.

`sech(x)` is rewritten as `1/cosh(x)`. Use `expand` or `rewrite` to rewrite expressions involving `tanh` and `coth` in terms of `sinh` and `cosh`. Cf. “Example 4” on page 1-1922.

The inverse function is implemented by `arccosh`. Cf. “Example 5” on page 1-1923.

The float attributes are kernel functions, i.e., floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
sinh(I*PI), cosh(1), tanh(5 + I), csch(PI), sech(1/11), coth(8)
```

$$0,\ \cosh(1),\ \tanh(5+i),\ \frac{1}{\sinh(\pi)},\ \frac{1}{\cosh\left(\frac{1}{11}\right)},\ \coth(8)$$

```
sinh(x), cosh(x + I*PI), tanh(x^2 - 4)
```

$$\sinh(x),\ -\cosh(x),\ \tanh(x^2-4)$$

Floating point values are computed for floating-point arguments:

```
sinh(123.4), cosh(5.6 + 7.8*I), coth(1.0/10^20)
```

$$1.953930316\ 10^{53},\ 7.295585032 + 135.0143985\ i,\ 1.0\ 10^{20}$$

For floating-point intervals, intervals enclosing the image are calculated:

```
cosh(-1 ... 1), tanh(-1 ... 1)
```

$$1.0 \ldots 1.543080635,\ -0.761594156 \ldots 0.761594156$$

For functions with discontinuities, evaluation over an interval may result in a union of intervals:

```
coth(-1 ... 1)
```

$$\text{RD\_NINF} \ldots -1.313035285 \cup 1.313035285 \ldots \text{RD\_INF}$$

Example 2

Simplifications are implemented for arguments that are integer multiples of $\frac{i\,\pi}{2}$:

```
sinh(I*PI/2), cosh(40*I*PI), tanh(-10^100*I*PI),
coth(-17/2*I*PI)
```

$$i,\ 1,\ 0,\ 0$$

Negative real numerical factors in the argument are rewritten via symmetry relations:

```
sinh(-5), cosh(-3/2*x), tanh(-x*PI/12), coth(-12/17*x*y*PI)
```

$$-\sinh(5),\ \cosh\left(\frac{3\,x}{2}\right),\ -\tanh\left(\frac{\pi\,x}{12}\right),\ -\coth\left(\frac{12\,\pi\,x\,y}{17}\right)$$

Example 3

The `expand` function implements the addition theorems:

```
expand(sinh(x + PI*I)), expand(cosh(x + y))
```

$$-\sinh(x),\ \cosh(x)\cosh(y)+\sinh(x)\sinh(y)$$

The `combine` function uses these theorems in the other direction, trying to rewrite products of hyperbolic functions:

```
combine(sinh(x)*sinh(y), sinhcosh)
```

$$\frac{\cosh(x+y)}{2}-\frac{\cosh(x-y)}{2}$$

Example 4

Various relations exist between the hyperbolic functions:

```
csch(x), sech(x)
```

$$\frac{1}{\sinh(x)},\ \frac{1}{\cosh(x)}$$

Use `rewrite` to obtain a representation in terms of a specific target function:

```
rewrite(tanh(x)*exp(2*x), sinhcosh), rewrite(sinh(x), tanh)
```

$$\frac{\sinh(x)\,(\cosh(2\,x)+\sinh(2\,x))}{\cosh(x)},\ -\frac{2\tanh\left(\frac{x}{2}\right)}{\tanh\left(\frac{x}{2}\right)^2-1}$$

```
rewrite(sinh(x)*coth(y), exp), rewrite(exp(x), coth)
```

$$-\frac{\left(\frac{e^{-x}}{2}-\frac{e^{x}}{2}\right)\left(e^{2\,y}+1\right)}{e^{2\,y}-1},\ \frac{\coth\left(\frac{x}{2}\right)+1}{\coth\left(\frac{x}{2}\right)-1}$$

Example 5

The inverse functions are implemented by `arcsinh`, `arccosh` etc.:

```
sinh(arcsinh(x)), sinh(arccosh(x)), cosh(arctanh(x))
```

$$x,\ \sqrt{x^2-1},\ \frac{1}{\sqrt{1-x}\,\sqrt{x+1}}$$

Note that `arcsinh(sinh(x))` does not necessarily yield `x`, because `arcsinh` produces values with imaginary parts in the interval $\left[-\frac{\pi}{2},\ \frac{\pi}{2}\right]$:

```
arcsinh(sinh(3)), arcsinh(sinh(1.6 + 100*I))
```

$$3,\ 1.6-0.5309649149\,i$$

Example 6

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the hyperbolic functions:

```
diff(sinh(x^2), x), float(sinh(3)*coth(5 + I))
```

$$2\,x\,\cosh(x^2),\ 10.01749636 - 0.0008270853591\,i$$

```
limit(x*sinh(x)/tanh(x^2), x = 0)
```

$$1$$

```
series((tanh(sinh(x)) - sinh(tanh(x)))/sinh(x^7), x = 0)
```

$$-\frac{1}{30} + \frac{29\,x^2}{756} - \frac{1913\,x^4}{75600} + O(x^6)$$

```
series(tanh(x), x = infinity)
```

$$1 - 2\,e^{-2x} + 2\,e^{-4x} - 2\,e^{-6x} + 2\,e^{-8x} - 2\,e^{-10x} + O\left(e^{-12x}\right)$$

Parameters

x

An arithmetical expression or a floating-point interval

Return Values

Arithmetical expression or a floating-point interval

Overloaded By

x

See Also

MuPAD Functions

`arccosh` | `arccoth` | `arccsch` | `arcsech` | `arcsinh` | `arctanh` | `coth` | `csch` | `sech` | `sinh` | `tanh`

tanh

Hyperbolic tangent function

Syntax

```
tanh(x)
```

Description

`tanh(x)` represents the hyperbolic tangent function `sinh(x)/cosh(x)`.

This function is defined for complex arguments.

Floating point values are returned for floating-point arguments. Floating point intervals are returned for floating-point interval arguments. Unevaluated function calls are returned for most exact arguments.

Arguments that are integer multiples of $\frac{i\pi}{2}$ lead to simplified results. If the argument involves a negative numerical factor of `Type::Real`, then symmetry relations are used to make this factor positive. Cf. “Example 2” on page 1-1926.

The special values $tanh(0) = 0$, $tanh(\infty) = 1$, $tanh(-\infty) = -1$ are implemented.

The functions `expand` and `combine` implement the addition theorems for the hyperbolic functions. Cf. “Example 3” on page 1-1927.

Use `expand` or `rewrite` to rewrite expressions involving `tanh` and `coth` in terms of `sinh` and `cosh`. Cf. “Example 4” on page 1-1927.

The inverse function is implemented by `arctanh`. Cf. “Example 5” on page 1-1928.

The float attributes are kernel functions, i.e., floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
sinh(I*PI), cosh(1), tanh(5 + I), csch(PI), sech(1/11), coth(8)
```

$$0,\ \cosh(1),\ \tanh(5+i),\ \frac{1}{\sinh(\pi)},\ \frac{1}{\cosh\left(\frac{1}{11}\right)},\ \coth(8)$$

```
sinh(x), cosh(x + I*PI), tanh(x^2 - 4)
```

$$\sinh(x),\ -\cosh(x),\ \tanh(x^2-4)$$

Floating point values are computed for floating-point arguments:

```
sinh(123.4), cosh(5.6 + 7.8*I), coth(1.0/10^20)
```

$$1.953930316\ 10^{53},\ 7.295585032 + 135.0143985\ i,\ 1.0\ 10^{20}$$

For floating-point intervals, intervals enclosing the image are calculated:

```
cosh(-1 ... 1), tanh(-1 ... 1)
```

$$1.0 \ldots 1.543080635,\ -0.761594156 \ldots 0.761594156$$

For functions with discontinuities, evaluation over an interval may result in a union of intervals:

```
coth(-1 ... 1)
```

$$\text{RD\_NINF} \ldots -1.313035285 \cup 1.313035285 \ldots \text{RD\_INF}$$

Example 2

Simplifications are implemented for arguments that are integer multiples of $\frac{i\,\pi}{2}$:

```
sinh(I*PI/2), cosh(40*I*PI), tanh(-10^100*I*PI),
coth(-17/2*I*PI)
```

$$i,\ 1,\ 0,\ 0$$

Negative real numerical factors in the argument are rewritten via symmetry relations:

```
sinh(-5), cosh(-3/2*x), tanh(-x*PI/12), coth(-12/17*x*y*PI)
```

$$-\sinh(5),\ \cosh\left(\frac{3x}{2}\right),\ -\tanh\left(\frac{\pi x}{12}\right),\ -\coth\left(\frac{12\pi xy}{17}\right)$$

Example 3

The `expand` function implements the addition theorems:

```
expand(sinh(x + PI*I)), expand(cosh(x + y))
```

$$-\sinh(x),\ \cosh(x)\cosh(y)+\sinh(x)\sinh(y)$$

The `combine` function uses these theorems in the other direction, trying to rewrite products of hyperbolic functions:

```
combine(sinh(x)*sinh(y), sinhcosh)
```

$$\frac{\cosh(x+y)}{2}-\frac{\cosh(x-y)}{2}$$

Example 4

Various relations exist between the hyperbolic functions:

```
csch(x), sech(x)
```

$$\frac{1}{\sinh(x)},\ \frac{1}{\cosh(x)}$$

Use `rewrite` to obtain a representation in terms of a specific target function:

```
rewrite(tanh(x)*exp(2*x), sinhcosh), rewrite(sinh(x), tanh)
```

$$\frac{\sinh(x)\,(\cosh(2\,x)+\sinh(2\,x))}{\cosh(x)},\ -\frac{2\tanh\left(\frac{x}{2}\right)}{\tanh\left(\frac{x}{2}\right)^2-1}$$

```
rewrite(sinh(x)*coth(y), exp), rewrite(exp(x), coth)
```

$$-\frac{\left(\frac{e^{-x}}{2}-\frac{e^{x}}{2}\right)\left(e^{2\,y}+1\right)}{e^{2\,y}-1},\ \frac{\coth\left(\frac{x}{2}\right)+1}{\coth\left(\frac{x}{2}\right)-1}$$

Example 5

The inverse functions are implemented by `arcsinh`, `arccosh` etc.:

```
sinh(arcsinh(x)), sinh(arccosh(x)), cosh(arctanh(x))
```

$$x,\ \sqrt{x^2-1},\ \frac{1}{\sqrt{1-x}\,\sqrt{x+1}}$$

Note that `arcsinh(sinh(x))` does not necessarily yield `x`, because `arcsinh` produces values with imaginary parts in the interval $\left[-\frac{\pi}{2},\ \frac{\pi}{2}\right]$:

```
arcsinh(sinh(3)), arcsinh(sinh(1.6 + 100*I))
```

$$3,\ 1.6-0.5309649149\,i$$

Example 6

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the hyperbolic functions:

```
diff(sinh(x^2), x), float(sinh(3)*coth(5 + I))
```

$$2\,x\,\cosh(x^2),\ 10.01749636 - 0.0008270853591\,i$$

```
limit(x*sinh(x)/tanh(x^2), x = 0)
```

$$1$$

```
series((tanh(sinh(x)) - sinh(tanh(x)))/sinh(x^7), x = 0)
```

$$-\frac{1}{30} + \frac{29\,x^2}{756} - \frac{1913\,x^4}{75600} + O(x^6)$$

```
series(tanh(x), x = infinity)
```

$$1 - 2\,e^{-2x} + 2\,e^{-4x} - 2\,e^{-6x} + 2\,e^{-8x} - 2\,e^{-10x} + O\left(e^{-12x}\right)$$

Parameters

x

An arithmetical expression or a floating-point interval

Return Values

Arithmetical expression or a floating-point interval

Overloaded By

x

See Also

MuPAD Functions

`arccosh` | `arccoth` | `arccsch` | `arcsech` | `arcsinh` | `arctanh` | `cosh` | `coth` | `csch` | `sech` | `sinh`

csch

Hyperbolic cosecant function

Syntax

```
csch(x)
```

Description

`csch(x)` represents the hyperbolic cosecant function `1/sinh(x)`.

This function is defined for complex arguments.

Floating point values are returned for floating-point arguments. Floating point intervals are returned for floating-point interval arguments. Unevaluated function calls are returned for most exact arguments.

Arguments that are integer multiples of $\frac{i\pi}{2}$ lead to simplified results. If the argument involves a negative numerical factor of `Type::Real`, then symmetry relations are used to make this factor positive. Cf. “Example 2” on page 1-1931.

The functions `expand` and `combine` implement the addition theorems for the hyperbolic functions. Cf. “Example 3” on page 1-1932.

`csch(x)` is rewritten as `1/sinh(x)`. Cf. “Example 4” on page 1-1932.

The inverse function is implemented by `arccsch`. Cf. “Example 5” on page 1-1933.

The float attributes are kernel functions, i.e., floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
sinh(I*PI), cosh(1), tanh(5 + I), csch(PI), sech(1/11), coth(8)
```

$$0,\ \cosh(1),\ \tanh(5+i),\ \frac{1}{\sinh(\pi)},\ \frac{1}{\cosh\left(\frac{1}{11}\right)},\ \coth(8)$$

```
sinh(x), cosh(x + I*PI), tanh(x^2 - 4)
```

$$\sinh(x),\ -\cosh(x),\ \tanh(x^2-4)$$

Floating point values are computed for floating-point arguments:

```
sinh(123.4), cosh(5.6 + 7.8*I), coth(1.0/10^20)
```

$$1.953930316\ 10^{53},\ 7.295585032 + 135.0143985\ i,\ 1.0\ 10^{20}$$

For floating-point intervals, intervals enclosing the image are calculated:

```
cosh(-1 ... 1), tanh(-1 ... 1)
```

$$1.0 \ldots 1.543080635,\ -0.761594156 \ldots 0.761594156$$

For functions with discontinuities, evaluation over an interval may result in a union of intervals:

```
coth(-1 ... 1)
```

$$\text{RD\_NINF} \ldots -1.313035285 \cup 1.313035285 \ldots \text{RD\_INF}$$

Example 2

Simplifications are implemented for arguments that are integer multiples of $\frac{i\,\pi}{2}$:

```
sinh(I*PI/2), cosh(40*I*PI), tanh(-10^100*I*PI),
coth(-17/2*I*PI)
```

$$i,\ 1,\ 0,\ 0$$

Negative real numerical factors in the argument are rewritten via symmetry relations:

```
sinh(-5), cosh(-3/2*x), tanh(-x*PI/12), coth(-12/17*x*y*PI)
```

$$-\sinh(5),\ \cosh\left(\frac{3x}{2}\right),\ -\tanh\left(\frac{\pi x}{12}\right),\ -\coth\left(\frac{12\pi xy}{17}\right)$$

Example 3

The `expand` function implements the addition theorems:

```
expand(sinh(x + PI*I)), expand(cosh(x + y))
```

$$-\sinh(x),\ \cosh(x)\cosh(y)+\sinh(x)\sinh(y)$$

The `combine` function uses these theorems in the other direction, trying to rewrite products of hyperbolic functions:

```
combine(sinh(x)*sinh(y), sinhcosh)
```

$$\frac{\cosh(x+y)}{2}-\frac{\cosh(x-y)}{2}$$

Example 4

Various relations exist between the hyperbolic functions:

```
csch(x), sech(x)
```

$$\frac{1}{\sinh(x)},\ \frac{1}{\cosh(x)}$$

Use `rewrite` to obtain a representation in terms of a specific target function:

```
rewrite(tanh(x)*exp(2*x), sinhcosh), rewrite(sinh(x), tanh)
```

$$\frac{\sinh(x)\,(\cosh(2\,x)+\sinh(2\,x))}{\cosh(x)},\ -\frac{2\tanh\left(\frac{x}{2}\right)}{\tanh\left(\frac{x}{2}\right)^2-1}$$

```
rewrite(sinh(x)*coth(y), exp), rewrite(exp(x), coth)
```

$$-\frac{\left(\frac{e^{-x}}{2}-\frac{e^{x}}{2}\right)\left(e^{2\,y}+1\right)}{e^{2\,y}-1},\ \frac{\coth\left(\frac{x}{2}\right)+1}{\coth\left(\frac{x}{2}\right)-1}$$

Example 5

The inverse functions are implemented by `arcsinh`, `arccosh` etc.:

```
sinh(arcsinh(x)), sinh(arccosh(x)), cosh(arctanh(x))
```

$$x,\ \sqrt{x^2-1},\ \frac{1}{\sqrt{1-x}\,\sqrt{x+1}}$$

Note that `arcsinh(sinh(x))` does not necessarily yield `x`, because `arcsinh` produces values with imaginary parts in the interval $\left[-\frac{\pi}{2},\ \frac{\pi}{2}\right]$:

```
arcsinh(sinh(3)), arcsinh(sinh(1.6 + 100*I))
```

$$3,\ 1.6-0.5309649149\,i$$

Example 6

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the hyperbolic functions:

```
diff(sinh(x^2), x), float(sinh(3)*coth(5 + I))
```

$$2\,x\,\cosh(x^2),\ 10.01749636 - 0.0008270853591\,i$$

```
limit(x*sinh(x)/tanh(x^2), x = 0)
```

$$1$$

```
series((tanh(sinh(x)) - sinh(tanh(x)))/sinh(x^7), x = 0)
```

$$-\frac{1}{30} + \frac{29\,x^2}{756} - \frac{1913\,x^4}{75600} + O(x^6)$$

```
series(tanh(x), x = infinity)
```

$$1 - 2\,e^{-2x} + 2\,e^{-4x} - 2\,e^{-6x} + 2\,e^{-8x} - 2\,e^{-10x} + O\left(e^{-12x}\right)$$

Parameters

x

An arithmetical expression or a floating-point interval

Return Values

Arithmetical expression or a floating-point interval

Overloaded By

x

See Also

MuPAD Functions

`arccosh` | `arccoth` | `arccsch` | `arcsech` | `arcsinh` | `arctanh` | `cosh` | `coth` | `sech` | `sinh` | `tanh`

sech

Hyperbolic secant function

Syntax

```
sech(x)
```

Description

`sech(x)` represents the hyperbolic secant function `1/cosh(x)`.

This function is defined for complex arguments.

Floating point values are returned for floating-point arguments. Floating point intervals are returned for floating-point interval arguments. Unevaluated function calls are returned for most exact arguments.

Arguments that are integer multiples of $\frac{i\pi}{2}$ lead to simplified results. If the argument involves a negative numerical factor of `Type::Real`, then symmetry relations are used to make this factor positive. Cf. “Example 2” on page 1-1936.

The functions `expand` and `combine` implement the addition theorems for the hyperbolic functions. Cf. “Example 3” on page 1-1937.

`sech(x)` is rewritten as `1/cosh(x)`. Cf. “Example 4” on page 1-1937.

The inverse function is implemented by `arcsech`. Cf. “Example 5” on page 1-1938.

The float attributes are kernel functions, i.e., floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
sinh(I*PI), cosh(1), tanh(5 + I), csch(PI), sech(1/11), coth(8)
```

$$0,\ \cosh(1),\ \tanh(5+i),\ \frac{1}{\sinh(\pi)},\ \frac{1}{\cosh\left(\frac{1}{11}\right)},\ \coth(8)$$

```
sinh(x), cosh(x + I*PI), tanh(x^2 - 4)
```

$$\sinh(x),\ -\cosh(x),\ \tanh(x^2-4)$$

Floating point values are computed for floating-point arguments:

```
sinh(123.4), cosh(5.6 + 7.8*I), coth(1.0/10^20)
```

$$1.953930316\ 10^{53},\ 7.295585032 + 135.0143985\ i,\ 1.0\ 10^{20}$$

For floating-point intervals, intervals enclosing the image are calculated:

```
cosh(-1 ... 1), tanh(-1 ... 1)
```

$$1.0 \ldots 1.543080635,\ -0.761594156 \ldots 0.761594156$$

For functions with discontinuities, evaluation over an interval may result in a union of intervals:

```
coth(-1 ... 1)
```

$$\text{RD\_NINF} \ldots -1.313035285 \cup 1.313035285 \ldots \text{RD\_INF}$$

Example 2

Simplifications are implemented for arguments that are integer multiples of $\frac{i\,\pi}{2}$:

```
sinh(I*PI/2), cosh(40*I*PI), tanh(-10^100*I*PI),
coth(-17/2*I*PI)
```

$$i,\ 1,\ 0,\ 0$$

Negative real numerical factors in the argument are rewritten via symmetry relations:

```
sinh(-5), cosh(-3/2*x), tanh(-x*PI/12), coth(-12/17*x*y*PI)
```

$$-\sinh(5),\ \cosh\left(\frac{3x}{2}\right),\ -\tanh\left(\frac{\pi x}{12}\right),\ -\coth\left(\frac{12\pi xy}{17}\right)$$

Example 3

The `expand` function implements the addition theorems:

```
expand(sinh(x + PI*I)), expand(cosh(x + y))
```

$$-\sinh(x),\ \cosh(x)\cosh(y)+\sinh(x)\sinh(y)$$

The `combine` function uses these theorems in the other direction, trying to rewrite products of hyperbolic functions:

```
combine(sinh(x)*sinh(y), sinhcosh)
```

$$\frac{\cosh(x+y)}{2}-\frac{\cosh(x-y)}{2}$$

Example 4

Various relations exist between the hyperbolic functions:

```
csch(x), sech(x)
```

$$\frac{1}{\sinh(x)},\ \frac{1}{\cosh(x)}$$

Use `rewrite` to obtain a representation in terms of a specific target function:

```
rewrite(tanh(x)*exp(2*x), sinhcosh), rewrite(sinh(x), tanh)
```

$$\frac{\sinh(x)\,(\cosh(2\,x)+\sinh(2\,x))}{\cosh(x)},\ -\frac{2\tanh\left(\frac{x}{2}\right)}{\tanh\left(\frac{x}{2}\right)^2-1}$$

```
rewrite(sinh(x)*coth(y), exp), rewrite(exp(x), coth)
```

$$-\frac{\left(\frac{e^{-x}}{2}-\frac{e^{x}}{2}\right)\left(e^{2\,y}+1\right)}{e^{2\,y}-1},\ \frac{\coth\left(\frac{x}{2}\right)+1}{\coth\left(\frac{x}{2}\right)-1}$$

Example 5

The inverse functions are implemented by `arcsinh`, `arccosh` etc.:

```
sinh(arcsinh(x)), sinh(arccosh(x)), cosh(arctanh(x))
```

$$x,\ \sqrt{x^2-1},\ \frac{1}{\sqrt{1-x}\,\sqrt{x+1}}$$

Note that `arcsinh(sinh(x))` does not necessarily yield `x`, because `arcsinh` produces values with imaginary parts in the interval $\left[-\frac{\pi}{2},\frac{\pi}{2}\right]$:

```
arcsinh(sinh(3)), arcsinh(sinh(1.6 + 100*I))
```

$$3,\ 1.6-0.5309649149\,i$$

Example 6

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the hyperbolic functions:

```
diff(sinh(x^2), x), float(sinh(3)*coth(5 + I))
```

$$2\,x\,\cosh(x^2),\ 10.01749636 - 0.0008270853591\,i$$

```
limit(x*sinh(x)/tanh(x^2), x = 0)
```

$$1$$

```
series((tanh(sinh(x)) - sinh(tanh(x)))/sinh(x^7), x = 0)
```

$$-\frac{1}{30} + \frac{29\,x^2}{756} - \frac{1913\,x^4}{75600} + O(x^6)$$

```
series(tanh(x), x = infinity)
```

$$1 - 2\,e^{-2x} + 2\,e^{-4x} - 2\,e^{-6x} + 2\,e^{-8x} - 2\,e^{-10x} + O\left(e^{-12x}\right)$$

Parameters

x

An arithmetical expression or a floating-point interval

Return Values

Arithmetical expression or a floating-point interval

Overloaded By

x

See Also

MuPAD Functions

`arccosh` | `arccoth` | `arccsch` | `arcsech` | `arcsinh` | `arctanh` | `cosh` | `coth` | `csch` | `sinh` | `tanh`

coth

Hyperbolic cotangent function

Syntax

```
coth(x)
```

Description

`coth(x)` represents the hyperbolic cotangent function `cosh(x)/sinh(x)`.

This function is defined for complex arguments.

Floating point values are returned for floating-point arguments. Floating point intervals are returned for floating-point interval arguments. Unevaluated function calls are returned for most exact arguments.

Arguments that are integer multiples of $\frac{i\pi}{2}$ lead to simplified results. If the argument involves a negative numerical factor of `Type::Real`, then symmetry relations are used to make this factor positive. Cf. “Example 2” on page 1-1941.

The special values $coth(\infty) = 1$, $coth(-\infty) = -1$ are implemented.

The functions `expand` and `combine` implement the addition theorems for the hyperbolic functions. Cf. “Example 3” on page 1-1942.

Use `expand` or `rewrite` to rewrite expressions involving `tanh` and `coth` in terms of `sinh` and `cosh`. Cf. “Example 4” on page 1-1942.

The inverse function is implemented by `arccoth`. Cf. “Example 5” on page 1-1943.

The float attributes are kernel functions, i.e., floating-point evaluation is fast.

Environment Interactions

When called with a floating-point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
sinh(I*PI), cosh(1), tanh(5 + I), csch(PI), sech(1/11), coth(8)
```

$$0,\ \cosh(1),\ \tanh(5+i),\ \frac{1}{\sinh(\pi)},\ \frac{1}{\cosh\left(\frac{1}{11}\right)},\ \coth(8)$$

```
sinh(x), cosh(x + I*PI), tanh(x^2 - 4)
```

$$\sinh(x),\ -\cosh(x),\ \tanh(x^2-4)$$

Floating point values are computed for floating-point arguments:

```
sinh(123.4), cosh(5.6 + 7.8*I), coth(1.0/10^20)
```

$$1.953930316\ 10^{53},\ 7.295585032 + 135.0143985\ i,\ 1.0\ 10^{20}$$

For floating-point intervals, intervals enclosing the image are calculated:

```
cosh(-1 ... 1), tanh(-1 ... 1)
```

$$1.0 \ldots 1.543080635,\ -0.761594156 \ldots 0.761594156$$

For functions with discontinuities, evaluation over an interval may result in a union of intervals:

```
coth(-1 ... 1)
```

$$\text{RD\_NINF} \ldots -1.313035285 \cup 1.313035285 \ldots \text{RD\_INF}$$

Example 2

Simplifications are implemented for arguments that are integer multiples of $\frac{i\pi}{2}$:

```
sinh(I*PI/2), cosh(40*I*PI), tanh(-10^100*I*PI),
coth(-17/2*I*PI)
```

$$i, 1, 0, 0$$

Negative real numerical factors in the argument are rewritten via symmetry relations:

```
sinh(-5), cosh(-3/2*x), tanh(-x*PI/12), coth(-12/17*x*y*PI)
```

$$-\sinh(5),\ \cosh\left(\frac{3x}{2}\right),\ -\tanh\left(\frac{\pi x}{12}\right),\ -\coth\left(\frac{12\,\pi\,x\,y}{17}\right)$$

Example 3

The `expand` function implements the addition theorems:

```
expand(sinh(x + PI*I)), expand(cosh(x + y))
```

$$-\sinh(x),\ \cosh(x)\cosh(y)+\sinh(x)\sinh(y)$$

The `combine` function uses these theorems in the other direction, trying to rewrite products of hyperbolic functions:

```
combine(sinh(x)*sinh(y), sinhcosh)
```

$$\frac{\cosh(x+y)}{2}-\frac{\cosh(x-y)}{2}$$

Example 4

Various relations exist between the hyperbolic functions:

```
csch(x), sech(x)
```

$$\frac{1}{\sinh(x)},\ \frac{1}{\cosh(x)}$$

Use `rewrite` to obtain a representation in terms of a specific target function:

```
rewrite(tanh(x)*exp(2*x), sinhcosh), rewrite(sinh(x), tanh)
```

$$\frac{\sinh(x)\,(\cosh(2\,x)+\sinh(2\,x))}{\cosh(x)},\ -\frac{2\tanh\left(\frac{x}{2}\right)}{\tanh\left(\frac{x}{2}\right)^2-1}$$

```
rewrite(sinh(x)*coth(y), exp), rewrite(exp(x), coth)
```

$$-\frac{\left(\frac{e^{-x}}{2}-\frac{e^{x}}{2}\right)\left(e^{2\,y}+1\right)}{e^{2\,y}-1},\ \frac{\coth\left(\frac{x}{2}\right)+1}{\coth\left(\frac{x}{2}\right)-1}$$

Example 5

The inverse functions are implemented by `arcsinh`, `arccosh` etc.:

```
sinh(arcsinh(x)), sinh(arccosh(x)), cosh(arctanh(x))
```

$$x,\ \sqrt{x^2-1},\ \frac{1}{\sqrt{1-x}\,\sqrt{x+1}}$$

Note that `arcsinh(sinh(x))` does not necessarily yield `x`, because `arcsinh` produces values with imaginary parts in the interval $\left[-\frac{\pi}{2},\frac{\pi}{2}\right]$:

```
arcsinh(sinh(3)), arcsinh(sinh(1.6 + 100*I))
```

$$3,\ 1.6-0.5309649149\,i$$

Example 6

Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the hyperbolic functions:

```
diff(sinh(x^2), x), float(sinh(3)*coth(5 + I))
```

$$2\,x\,\cosh(x^2),\ 10.01749636 - 0.0008270853591\,i$$

```
limit(x*sinh(x)/tanh(x^2), x = 0)
```

$$1$$

```
series((tanh(sinh(x)) - sinh(tanh(x)))/sinh(x^7), x = 0)
```

$$-\frac{1}{30} + \frac{29\,x^2}{756} - \frac{1913\,x^4}{75600} + O(x^6)$$

```
series(tanh(x), x = infinity)
```

$$1 - 2\,e^{-2x} + 2\,e^{-4x} - 2\,e^{-6x} + 2\,e^{-8x} - 2\,e^{-10x} + O\left(e^{-12x}\right)$$

Parameters

x

An arithmetical expression or a floating-point interval

Return Values

Arithmetical expression or a floating-point interval

Overloaded By

x

See Also

MuPAD Functions

`arccosh` | `arccoth` | `arccsch` | `arcsech` | `arcsinh` | `arctanh` | `cosh` | `csch` | `sech` | `sinh` | `tanh`

slot

Method or entry of a domain or a function environment

Syntax

```
d::n
```

```
slot(d, "n")
```

```
d::n := v
```

```
slot(d, "n", v)
```

```
object::dom
```

```
slot(object, "dom")
```

Description

`d::n` returns the value of the slot named `"n"` of the object `d`.

`d::n := v` creates or changes the slot `"n"`. The value `v` is assigned to the slot.

The function `slot` is used for defining methods and entries of data types (domains) or for defining attributes of function environments. Such methods, entries, or attributes are called *slots*. They allow to overload system functions by user defined domains and function environments. See the “Background” section below for further information.

Any MuPAD object has a special slot named `"dom"`. It holds the domain the object belongs to: `slot(object, "dom")` is equivalent to `domtype(object)`. The value of this special slot cannot be changed. Cf. “Example 1” on page 1-1946.

Apart from the special slot `"dom"`, only domains and function environments may have further slots.

The call `slot(d, "n")` is equivalent to `d::n`. It returns the value of the slot.

The call `slot(d, "n", v)` returns the object `d` with an added or changed slot `"n"` bearing the value `v`.

For a *function environment*`d`, the call `slot(d, "n", v)` returns `d` with the changed slot `"n"` and changes the function environment `d` as a side-effect. This is the so-called "reference effect" of function environments. Cf. "Example 2" on page 1-1947.

For a *domain*`d`, however, the call `slot(d, "n", v)` modifies `d` as a side-effect and returns the domain. This is the so-called "reference effect" of domains. Cf. "Example 3" on page 1-1947.

If a non-existing slot is accessed, `FAIL` is returned as the value of the slot. Cf. "Example 4" on page 1-1948.

The `::`-operator is a shorthand notation to access a slot.

The expression `d::n`, when not appearing on the left hand side of an assignment, is equivalent to `slot(d, "n")`.

The command `d::n := v` assigns the value `v` to the slot `"n"` of `d`. This assignment is almost equivalent to changing or creating a slot via `slot(d, "n", v)`. Note the following subtle semantical difference between these assignments: in `d::n := v`, the identifier `d` is evaluated with level 1, i.e., the slot `"n"` is attached to the *value* of `d`. In `slot(d, "n", v)`, the identifier `d` is *fully evaluated*. See "Example 6" on page 1-1949.

With `delete d::n` or `delete slot(d,"n")`, the slot `"n"` of the function environment or the domain `d` is deleted. Cf. "Example 5" on page 1-1948. The special slot `"dom"` cannot be deleted.

The first argument of `slot` is not flattened. This allows to access the slots of expression sequences and `null()` objects. Cf. "Example 7" on page 1-1949.

For domains, there is a special mechanism to create new values for slots on demand. If a non existing slot is read, the method `"make_slot"` of the domain is called in order to create the slot. If such a method does not exist, `FAIL` is returned. Cf. "Example 8" on page 1-1950.

Examples

Example 1

Every object has the slot `"dom"`:

```
x::dom = domtype(x),
slot(45, "dom") = domtype(45),
sin::dom = domtype(sin)
```

$$\text{DOM\_IDENT} = \text{DOM\_IDENT},\ \text{DOM\_INT} = \text{DOM\_INT},\ \text{DOM\_FUNC\_ENV} = \text{DOM\_FUNC\_ENV}$$

Example 2

Here we access the existing `"float"` slot of the function environment `sin` implementing the sine function. The float slot is again a function environment and may be called like any MuPAD function. Note, however, the different functionality: in contrast to `sin`, the float slot always tries to compute a floating-point approximation:

```
s := sin::float:  s(1), sin(1)
```

$$0.8414709848,\ \sin(1)$$

With the following commands, `s` becomes the function environment `sin` apart from a changed `"float"` slot. The `slot` call has no effect on the original `sin` function because `slot` returns a copy of the function environment:

```
s := funcenv(sin):
s::float := x -> float(x - x^3/3!):
s(PI/3) = sin(PI/3), s::float(1) <> sin::float(1)
```

$$\frac{\sqrt{3}}{2} = \frac{\sqrt{3}}{2},\ 0.8333333333 \neq 0.8414709848$$

```
delete s:
```

Example 3

If you are using the `slot` function to change slot entries in a domain, you must be aware that you are modifying the domain:

```
old_one := Dom::Float::one
```

$$1.0$$

```
newDomFloat := slot(Dom::Float, "one", 1):
newDomFloat::one, Dom::Float::one
```

```
1, 1
```

We restore the original state:

```
slot(Dom::Float, "one", old_one):  Dom::Float::one
```

```
1.0
```

```
delete old_one, newDomFloat:
```

Example 4

The function environment `sin` does not contain a `"sign"` slot. So accessing this slot yields `FAIL`:

```
slot(sin, "sign"), sin::sign
```

```
FAIL, FAIL
```

Example 5

We define a function environment for a function computing the logarithm to the base `3`:

```
log3 := funcenv(x -> log(3, x)):
```

If the function `info` is to give some information about `log3`, we have to define the `"info"` slot for this function:

```
log3::info := "log3 -- the logarithm to the base 3":

info(log3)
```

```
log3 -- the logarithm to the base 3
```

The `delete` statement is used for deleting a slot:

```
delete log3::info: info(log3)
```

```
log3(x) -- a library procedure [try ?log3 for help]
```

It is not possible to delete the special slot `"dom"`:

```
delete log3::dom
```

```
Error: The argument is invalid. [delete]
```

```
delete log3:
```

Example 6

Here we demonstrate the subtle difference between the `slot` function and the use of the `::`-operator in assignments. The following call adds a `"xyz"` slot to the domain `DOM_INT` of integer numbers:

```
delete b: d := b: b := DOM_INT: slot(d, "xyz", 42):
```

The slot `"xyz"` of `DOM_INT` is changed, because `d` is fully evaluated with the result `DOM_INT`. Hence, the slot `DOM_INT::xyz` is set to 42:

```
slot(d, "xyz"), slot(DOM_INT, "xyz")
```

```
42, 42
```

Here is the result when using the `::`-operator: `d` is only evaluated with level 1, i.e., it is evaluated to the identifier `b`. However, there is no slot `b::xyz`, and an error occurs:

```
delete b: d := b: b := DOM_INT: d::xyz := 42
```

```
Error: Slot 'd::xyz' is unknown. [slot]
```

```
delete b, d:
```

Example 7

The first argument of `slot` is not flattened. This allows access to the slots of expression sequences and `null()` objects:

```
slot((a, b), "dom") = (a,b)::dom,
slot(null(), "dom") = (null())::dom
```

```
DOM_EXPR = DOM_EXPR, DOM_NULL = DOM_NULL
```

Example 8

We give an example for the use of the function `make_slot`. The element `undefined` of the domain `stdlib::Undefined` represents an undefined value. Any function `f` should yield `f(undefined) = undefined`. Inside the implementation of `stdlib::Undefined`, we find:

```
undef := newDomain("stdlib::Undefined"):
undefined := new(undef):
undef::func_call := proc() begin undefined end_proc;
undef::make_slot := undef::func_call:
```

The following mechanism takes place automatically for a function `f` that is overloadable by its first argument: in the call `f(undefined)`, it is checked whether the slot `undef::f` exists. If this is not the case, the `make_slot` function creates this slot "on the fly", producing the value `undefined`. Thus, via overloading, `f(undefined)` returns the value `undefined`.

Example 9

The following example is rather advanced and technical. It demonstrates overloading of the `slot` function to implement slot access and slot assignments for other objects than domains (`DOM_DOMAIN`) or function environments (`DOM_FUNC_ENV`). The following example defines the slots `"numer"` and `"denom"` for rational numbers. The domain `DOM_RAT` of such numbers does not have slots `"numer"` and `"denom"`:

```
domtype(3/4)
```

```
DOM_RAT
```

```
slot(3/4, "numer")
```

```
Error: Slot '(3/4)::numer' is unknown. [slot]
```

We can change `DOM_RAT`, however. For this, we have to unprotect`DOM_RAT` temporarily:

```
unprotect(DOM_RAT):
```

```
DOM_RAT::slot :=
  proc(r : DOM_RAT, n : DOM_STRING, v=null(): DOM_INT)
      local i : DOM_INT;
  begin
      i := contains(["numer", "denom"], n);
      if i = 0 then
         error("Unknown slot \"".expr2text(r)."::".n."\"")
      end;
      if args(0) = 3 then
         subsop(r, i = v)
      else
         op(r, i)
      end
  end_proc:
```

Now, we can access the operands of rational numbers, which are the numerator and the denominator respectively, via our new slots:

```
slot(3/4, "numer"), (3/4)::numer,
slot(3/4, "denom"), (3/4)::denom
```

$$3, 3, 4, 4$$

```
a := 3/4:  slot(a, "numer", 7)
```

$$\frac{7}{4}$$

```
a::numer := 11:  a
```

$$\frac{11}{4}$$

We restore the original behavior:

```
delete DOM_RAT::slot, a: protect(DOM_RAT, Error):
```

Parameters

d

A domain or a function environment

n

The name of the slot: an identifier

v

The new value of the slot: an arbitrary MuPAD object

object

An arbitrary MuPAD object

Return Values

`slot(d, "n")` returns the value of the slot; `slot(d, "n", v)` returns the object `d` with the added or changed slot; `slot(object, "dom")` returns the domain type of the object.

Overloaded By

d

Algorithms

Overloading of system functions by domain elements is typically implemented as follows. If a library function `f`, say, is to be overloadable by user defined data types, a code segment as indicated by the following lines is appropriate. It tests whether the domain `x::dom` of the argument `x` contains a method `f`. If this is the case, this domain method is called:

```
f:= proc(x)
    begin
      // check if f is overloaded by x
      if x::dom::f <> FAIL then
           // use the method of the domain of x
           return(x::dom::f(args()))
      else
```

```
        // execute the code for the function f
    endif
  end_proc:
```

By overloading the function `slot`, slot access and slot assignment can be implemented for other objects than domains or function environments. Cf. “Example 9” on page 1-1950.

In principle, the name `n` of a slot may be an arbitrary MuPAD object. Note, however, that the `::`-operator cannot access slots defined by `slot(d, n, v)` if the the name `n` is not a string.

Strings may be used in conjunction with the `::`-operator: the calls `d::"n"` and `d::n` are equivalent.

See Also

MuPAD Domains

`DOM_DOMAIN` | `DOM_FUNC_ENV`

MuPAD Functions

`funcenv` | `newDomain` | `slotAssignCounter`

slotAssignCounter

Counts slot assignments

Syntax

```
slotAssignCounter(key)
```

Description

`slotAssignCounter(key)` returns the number of `slot` assignments with the key `key` since the initialization of the counter. The counter for `key` is initialized with `0` on the first call of `slotAssignCounter(key)`. Previous assignments with the key `key` are not counted.. This function serves a highly technical purpose. Usually, there should be no need for a user to call this function.

`slotAssignCounter` only counts assignments to slots of domains and function environments.

`slotAssignCounter` was introduced as a dependency check function for `prog::remember`. See “Example 2” on page 1-1955.

Examples

Example 1

We initialize slot assignment counting of the slot `"foo"`:

```
slotAssignCounter("foo")
```

```
     0
```

Then we define a function `f` with a slot `"foo"`:

```
f := funcenv(f):
f::foo := bar:
```

Now the counter has the value 1:

```
slotAssignCounter("foo")
```

```
     1
```

Example 2

Here we define a recursing function `foo` which overloads for domain elements. The function remembers computed values with `prog::remember`:

```
foo := x -> (if x::dom::foo <> FAIL then return(x::dom::foo(x)) end_if;
             if x = op(x) then procname(x) else map(x, foo) end_if):
foo := prog::remember(foo):
```

Then we define a domain `bar` which does not overload the slot `"foo"`:

```
bar := newDomain("bar"):
bar::new := x -> new(bar, x):
bar::print := x -> hold(bar)(extop(x)):
bar::op := id:
foo(bar(2))
```

```
     foo(bar(2))
```

Now we add a `"foo"` slot to `bar`:

```
bar::foo := x -> 4:
foo(bar(2))
```

```
     foo(bar(2))
```

The new slot was not used, because `foo` took the result from its remember table. If we use a dependency function with `slotAssignCounter` in `prog::remember`, we can make `foo` aware of changes in `"foo"`-slots of other functions and domains:

```
foo := x -> (if x::dom::foo <> FAIL then return(x::dom::foo(x)) end_if;
             if x = op(x) then x else map(x, foo) end_if):
foo := prog::remember(foo, () -> slotAssignCounter("foo")):
foo(bar(2));
bar::foo := x -> 5:
```

```
foo(bar(2))
```

```
    4
```

```
    5
```

```
delete foo, bar:
```

Parameters

`key`

Any MuPAD object

Return Values

Non-negative number of type `DOM_INT`.

See Also

MuPAD Functions

`prog::remember` | `slot`

solve

Solve equations and inequalities

Syntax

```
solve(eq, x, options)

solve(eq, x = a .. b, options)

solve(eq, vars, options)

solve(eq, options)

solve(eqs, x, options)

solve(eqs, vars, options)

solve(eqs, options)

solve(ODE)

solve(REC)
```

Description

`solve(eq, x)` returns the set of all complex solutions of an equation or inequality `eq` with respect to `x`.

`solve(eq, x = a..b)` returns the set of all solutions in the closed interval `Dom::Interval([a, b])`.

`solve(eq, vars)` solves an equation for the variables `vars`.

`solve(eqs, x)` solves a system `eqs` for the variable `x`.

`solve(eqs, vars)` solves a system `eqs` of equations for the variables `vars`.

The `solve` function provides a unified interface to a variety of specialized solvers. See Choosing a Solver.

If you do not specify indeterminates for which you want to solve an equation, inequality or system, the solver uses a set of all indeterminates. Indeterminates must be identifiers or indexed identifiers. You cannot use mathematical constants, such as PI, EULER, and so on, as indeterminates. The solver discards indeterminates that appear only inside function names or indices. See “Example 12” on page 1-1966.

If you specify a list of indeterminates for which you want to solve an equation, an inequality, or a system, the solver sorts the components of the resulting solution vectors according to the order of the indeterminates that you used. If you specify indeterminates as a set, MuPAD can change the order of the indeterminates.

`solve(eq, vars)` is equivalent to `solve([eq], vars)`.

The solver can return the following types of sets:

- Finite sets (type `DOM_SET`).
- Symbolic calls to `solve`.
- Zero sets of polynomials (type `RootOf`). The solver returns a set of this type if it cannot solve an equation explicitly in terms of radicals. The solver also can return this type of set when you use the `MaxDegree` option.
- Set-theoretic expressions, such as `"_union"`, `"_intersect"`, and `"_minus"`.
- Symbolic calls to `solvelib::Union`. These calls represent unions over parametrized systems of sets.
- The $\mathbb{C}$, $\mathbb{R}$, $\mathbb{Q}$, and $\mathbb{Z}$ (type `solvelib::BasicSet`) sets.
- Intervals (type `Dom::Interval`).
- Image sets of functions (type `Dom::ImageSet`).
- Piecewise objects in which every branch defines a set of one of the valid types (type `piecewise`).

MuPAD can use sets of these types, excluding intervals and basic sets, to represent sets of vectors (for solutions of systems). When solving a system, MuPAD also can return a solution in the form S^n (the n-fold cartesian power of the set S of scalars). Here S is a set of any type returned by `solve`.

For returned solution sets, you can use the set-theoretic operations, such as `intersect`, `union`, and `minus`. Also, you can use pointwise-defined arithmetical operations, such as `+`, `*`, and so on. To extract elements of a set, use the `solvelib::getElement` function. To test whether the solution set returned by `solve` is finite, use the function `solvelib::isFinite`. See “Example 2” on page 1-1960

For systems, the solver returns a set of vectors or a set of lists of equations. To specify that the solver must return a set of vectors, use the `VectorFormat` option. See “Example 10” on page 1-1964.

By default, `solve(eq, x)` returns only the solutions consistent with the properties of `x`. To ignore the properties of `x`, use the `IgnoreProperties` option. This option is helpful when you solve a system of equations for more than one variable. See “Example 13” on page 1-1966.

An inequality `a <= b` or `a < b` holds only when both sides represent real numbers. In particular, `a = b` does not imply that `a <= b` for complex numbers.

You can write custom domains for equations of special types, and then overload `solve` for these domains. MuPAD uses this feature for differential and recurrence equations. See the `ode`, `ode::solve`, and `rec` help pages.

The `solve` function is a symbolic solver. If you want to use numeric methods, see the `numeric::solve` help page for available options and examples.

If the input contains floating-point numbers, the solver replaces them by approximate rational values. The accuracy of these approximate values depends on the environment variable `DIGITS`. If `solve` finds a solution, MuPAD internally calls the `float` function for that solution, and then returns the result. If the symbolic solver returns `unevaluated`, MuPAD calls `numeric::solve`. See “Example 16” on page 1-1967.

If a numerator contains a factored polynomial with the multiplicities greater than 1, the solver does not check the multiple roots for zeros in the denominator. See “Example 17” on page 1-1968.

Environment Interactions

`solve` reacts to properties of identifiers.

Examples

Example 1

Solve the following equation. Typically, for equations with a finite number of solutions, the solver returns a set of the `DOM_SET` type:

```
solve(x^4 - 5*x^2 + 6*x = 2, x)
```

$$\{1, -\sqrt{3} - 1, \sqrt{3} - 1\}$$

Example 2

The solver can also return an infinite discrete set of solutions:

```
S := solve(sin(x*PI/7) = 0, x)
```

$$\{7\,k \mid k \in \mathbb{Z}\}$$

To select the solutions in a particular finite interval, find the intersection of the solution set with the interval:

```
S intersect Dom::Interval(-22, 22)
```

$$\{-21, -14, -7, 0, 7, 14, 21\}$$

Alternatively, specify the interval when calling the solver. For example, compute the solutions in the interval [- 22, 22]:

```
solve(sin(x*PI/7) = 0, x = -22..22)
```

$$\{-21, -14, -7, 0, 7, 14, 21\}$$

```
delete S:
```

Example 3

Use the `solve` function to solve inequalities. Typically, the solution set of an inequality is an interval or a union of intervals:

```
solve(x^2 > 5, x)
```

$$(-\infty, -\sqrt{5}) \cup (\sqrt{5}, \infty)$$

Example 4

Solve the following inequality. The solution includes the set of all complex numbers, excluding $\sqrt{7}$ and $-\sqrt{7}$:

```
solve(x^2 <> 7, x)
```

$$\mathbb{C} \setminus \{\sqrt{7}, -\sqrt{7}\}$$

Example 5

The solver can return a solution as a union of an infinite family of sets. The `solvelib::Union` function represents such infinite unions in MuPAD:

```
solve(sin(x)*cos(x) > 1/4, x, Real)
```

$$\bigcup_{k \in \mathbb{Z}} \left(\frac{\pi}{12} + \pi\, k, \frac{5\,\pi}{12} + \pi\, k \right)$$

Example 6

If an equation contains symbolic parameters, the solver returns a piecewise solution. For example, solve the quadratic equation $ax^2 + bx + c = 0$:

```
S := solve(a*x^2 + b*x + c, x)
```

$$\begin{cases} \left\{ -\frac{b+\sigma_1}{2\,a}, -\frac{b-\sigma_1}{2\,a} \right\} & \text{if } a \neq 0 \\ \left\{ -\frac{c}{b} \right\} & \text{if } a = 0 \wedge b \neq 0 \\ \mathbb{C} & \text{if } a = 0 \wedge b = 0 \wedge c = 0 \\ \varnothing & \text{if } a = 0 \wedge b = 0 \wedge c \neq 0 \end{cases}$$

where

$$\sigma_1 = \sqrt{b^2 - 4\,a\,c}$$

Now, evaluate the solution assuming that a is not equal to 0:

```
assume(a <> 0): S
```

$$\left\{-\frac{b+\sqrt{b^2-4\,a\,c}}{2\,a}, -\frac{b-\sqrt{b^2-4\,a\,c}}{2\,a}\right\}$$

```
delete S: unassume(a):
```

Example 7

By default, the solver tries to find all possible solutions. The following inequality has both real and complex solutions. For example, $x = \sqrt{\frac{3}{4}} + \frac{i}{2}$ is one of the solutions. The solver cannot find a closed-form representation of all possible solutions:

```
solve(x + 1/x > 0, x)
```

$$\text{solve}\left(0 < x + \frac{1}{x}, x\right)$$

With the `Real` option, the solver computes only real solutions. The closed-form representation of all real solutions of that equation is an interval of all real numbers from 0 to infinity:

```
solve(x + 1/x > 0, x, Real)
```

$$(0, \infty)$$

Example 8

Solve this equation. By default, the solver returns a complete, but rather long and complicated solution:

```
solve(x^(7/2) + 1/x^(7/2) = 1, x)
```

$$\left\{\frac{1}{\sigma_1}, \frac{1}{\sigma_2}, \frac{e^{\frac{4\pi i}{7}}}{\sigma_1}, \frac{e^{\frac{4\pi i}{7}}}{\sigma_2}, -\frac{e^{\frac{3\pi i}{7}}}{\sigma_1}, -\frac{e^{\frac{3\pi i}{7}}}{\sigma_2}\right\}$$

where

$$\sigma_1 = \left(\frac{1}{2} - \frac{\sqrt{3}\,i}{2}\right)^{2/7}$$

$$\sigma_2 = \left(\frac{1}{2} + \frac{\sqrt{3}\,i}{2}\right)^{2/7}$$

Using `IgnoreAnalyticConstraints`, you often can get simpler results:

```
solve(x^(7/2) + 1/x^(7/2) = 1, x, IgnoreAnalyticConstraints)
```

$$\left\{\frac{1}{\left(\frac{1}{2} - \frac{\sqrt{3}\,i}{2}\right)^{2/7}}, \frac{1}{\left(\frac{1}{2} + \frac{\sqrt{3}\,i}{2}\right)^{2/7}}\right\}$$

Using this option, you also can get wrong results:

```
solve(arcsin(x) = C, x, IgnoreAnalyticConstraints) assuming C > 10
```

$$\{\sin(C)\}$$

Always check the results obtained with this option:

```
testeq(arcsin(sin(C)), C)
```

FALSE

The `IgnoreAnalyticConstraints` option also can lead to incomplete results:

```
solve(x^(5/2) = 1, x)
```

$$\left\{1, -\frac{\sqrt{5}}{4} - \frac{1}{4} - \frac{\sqrt{2}\sqrt{5-\sqrt{5}}\,i}{4}, -\frac{\sqrt{5}}{4} - \frac{1}{4} + \frac{\sqrt{2}\sqrt{5-\sqrt{5}}\,i}{4}\right\}$$

```
solve(x^(5/2) = 1, x, IgnoreAnalyticConstraints)
```

$$\{1\}$$

Example 9

With the `IgnoreAnalyticConstraints` option, the solver can multiply both sides of an equation by any expression, except `0`. In the following example, the solver multiplies both sides of the equation by $\sqrt{x}\,\sqrt{y}$. The solver does not consider the special case `x = y = 0`:

```
solve(1/sqrt(x) = 1/sqrt(y), IgnoreAnalyticConstraints)
```

$$\{[x = z,\ y = z]\}$$

The result is not valid for `x = y = 0`.

Example 10

When you solve a system of equations, MuPAD tries to represent the solutions as a set of lists of substitutions:

```
solve([x^2 + y = 1, x + y^2 = 1], [x, y])
```

$$\left\{[x = 0,\ y = 1],\ [x = 1,\ y = 0],\ \left[x = -\frac{\sqrt{5}}{2} - \frac{1}{2},\ y = -\frac{\sqrt{5}}{2} - \frac{1}{2}\right],\ \left[x = \frac{\sqrt{5}}{2} - \frac{1}{2},\ y = \frac{\sqrt{5}}{2} - \frac{1}{2}\right]\right\}$$

If you use the `VectorFormat` option, MuPAD returns a solution as a set of vectors:

```
solve([x^2 + y = 1, x + y^2 = 1], [x, y], VectorFormat)
```

$$\left\{\begin{pmatrix}1\\0\end{pmatrix},\ \begin{pmatrix}0\\1\end{pmatrix},\ \begin{pmatrix}-\frac{\sqrt{5}}{2} - \frac{1}{2}\\ -\frac{\sqrt{5}}{2} - \frac{1}{2}\end{pmatrix},\ \begin{pmatrix}\frac{\sqrt{5}}{2} - \frac{1}{2}\\ \frac{\sqrt{5}}{2} - \frac{1}{2}\end{pmatrix}\right\}$$

Right sides of the returned substitutions can contain generated identifiers. In this case, substituting each of these identifiers with a complex number gives a solution of

the system. You can obtain all solutions by substituting generated identifiers with all complex numbers:

```
sys:= [x + y + z = 2, x + y^2 + z^2 = 4]:
solve(sys, [x, y, z])
```

$$\left\{\left[x = \frac{3}{2} - \sigma_1 - z1,\ y = \sigma_1 + \frac{1}{2},\ z = z1\right],\ \left[x = \sigma_1 - z1 + \frac{3}{2},\ y = \frac{1}{2} - \sigma_1,\ z = z1\right]\right\}$$

where

$$\sigma_1 = \frac{\sqrt{-4\,z1^2 + 4\,z1 + 9}}{2}$$

If you use the `VectorFormat` option, the solver returns a solution as an infinite set of vectors, in the usual mathematical notation:

```
solve(sys, [x, y, z], VectorFormat); delete sys:
```

$$\left\{\begin{pmatrix} -z1 + \sigma_1 + \frac{3}{2} \\ -\sigma_1 + \frac{1}{2} \\ z1 \end{pmatrix} \middle|\ z1 \in \mathbb{C}\right\} \cup \left\{\begin{pmatrix} -z1 - \sigma_1 + \frac{3}{2} \\ \sigma_1 + \frac{1}{2} \\ z1 \end{pmatrix} \middle|\ z1 \in \mathbb{C}\right\}$$

where

$$\sigma_1 = \frac{\sqrt{-4\,z1^2 + 4\,z1 + 9}}{2}$$

Example 11

You can specify the variable, for which you want to solve an equation, as a list of one entry. In this case, the solver returns the solution using the output format typically used for systems:

```
solve(x = x, x), solve(x = x, [x])
```

$$\mathbb{C},\ \{[x = z]\}$$

Example 12

If you do not specify indeterminates (the variables for which you want to solve an equation), the solver uses the set of all indeterminates that it can find in that equation:

```
solve(x^2 = 3)
```

$$\{[x = \sqrt{3}], [x = -\sqrt{3}]\}$$

The solver does not regard operators and indices as indeterminates. Therefore, the solver does not treat `f` and `y` as indeterminates in the following equation:

```
solve(f(x[y]) = 7)
```

$$\text{solve}\left(\left[f\left(x_y\right) = 7\right], \left[x_y\right]\right)$$

Example 13

If you set an assumption on the variable for which you want to solve an equation, the solver returns only the results compatible with that assumption. For example, assume that `x` represents a real positive number. Then, solve the following equation:

```
assume(x, Type::Positive): solve(x^4 = 1, x)
```

$$\{1\}$$

Without that assumption, the solver returns all complex solutions:

```
unassume(x): solve(x^4 = 1, x)
```

$$\{-1, 1, -i, i\}$$

Example 14

To obtain the multiplicities of the roots of a polynomial, use the `Multiple` option. For example, the polynomial $x^3 + 2\,x^2 + x$ has two roots. The multiplicity of $x = -1$ is 2. The multiplicity of $x = 0$ is 1:

```
solve(x^3 + 2*x^2 + x, x, Multiple)
```

$$\{[-1, 2], [0, 1]\}$$

Example 15

Suppose, you want to solve the following system of equations for two variables, `x` and `y`. Suppose, you want to avoid backward substitutions while solving this system. To disable backward substitutions, use the option `BackSubstitution = FALSE`. Specify the list of variables so that `x` appears to the right of `y`. Now, the solution for the variable `y` can contain the variable `x`:

```
solve({x^2 + y = 1, x - y = 2}, [y, x], BackSubstitution = FALSE)
```

$$\left\{\left[y = x - 2, x = -\frac{\sqrt{13}}{2} - \frac{1}{2}\right], \left[y = x - 2, x = \frac{\sqrt{13}}{2} - \frac{1}{2}\right]\right\}$$

```
solve({x^2 + y = 1, x - y = 2}, {x, y})
```

$$\left\{\left[x = -\frac{\sqrt{13}}{2} - \frac{1}{2}, y = -\frac{\sqrt{13}}{2} - \frac{5}{2}\right], \left[x = \frac{\sqrt{13}}{2} - \frac{1}{2}, y = \frac{\sqrt{13}}{2} - \frac{5}{2}\right]\right\}$$

If MuPAD cannot express the result as a set of lists, then `BackSubstitution` has no effect:

```
solve({x^2 + y = 1, x - y = 2}, [y, x],
BackSubstitution = FALSE, MaxDegree = 1)
```

$$\begin{pmatrix} y \\ x \end{pmatrix} \in \left\{ \begin{pmatrix} x - 2 \\ z1 \end{pmatrix} \;\middle|\; z1 \in \mathrm{RootOf}(z^2 + z - 3, z) \right\}$$

Example 16

If the input contains floating-point numbers, MuPAD uses the symbolic solver `solve`, and then calls the `float` function for the obtained solution:

```
solve(x^3 + 3.0*x + 1, x)
```

$$\{-0.3221853546, 0.1610926773 + 1.75438096\,i, 0.1610926773 - 1.75438096\,i\}$$

If the symbolic solver fails to solve such equation or system, MuPAD calls the numeric solver `numeric::solve`:

```
solve({sin(x) + 1/2*cos(sqrt(2)*y) = 1, cos(x) + sin(y) = 0.1}, {x, y})
```

$$\{[x = 0.7780082473,\ y = -0.6589827125]\}$$

The numeric solver can return an incomplete set of solutions. For details, see the `numeric::solve` help page.

Example 17

If a numerator contains a factored polynomial with the multiplicities greater than 1, the solutions might give zeros in a denominator:

```
solve((x - 1)^2/(x - 1) = 0, x)
```

$$\{1\}$$

To eliminate these solutions, expand a numerator:

```
f := expand((x - 1)^2): solve(f/(x - 1) = 0, x)
```

$$\varnothing$$

Example 18

You can use the `solve` function to solve Diophantine equations. For example, solve the following linear Diophantine equation:

```
S := solve(30*x + 56*y = 2, [x, y], Domain = Z_)
```

$$\binom{x}{y} \in \left\{ \begin{pmatrix} -28\,k - 13 \\ 15\,k + 7 \end{pmatrix} \,\middle|\, k \in \mathbb{Z} \right\}$$

Example 19

You can use the `solve` function to solve equation given in the form of memberships. For example, solve the following equation:

```
solve(x^2 in Z_, x)
```

$$\{\sqrt{k} \mid k \in \mathbb{Z}\} \cup \{-\sqrt{k} \mid k \in \mathbb{Z}\}$$

Example 20

You can solve an equation with symbolic parameters, thus finding its general solution. Then you can evaluate the solution for any particular values of parameters or plot the solution with respect to the parameter values.

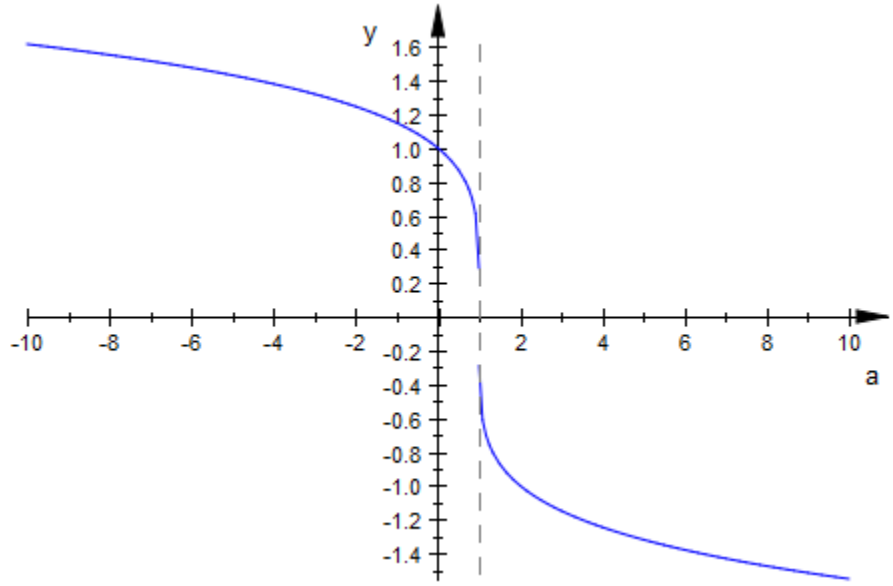
Solve this equation:

```
S := solve(x^5 + a = 1, x, Real)
```

$$\left\{-|a-1|^{1/5}\,\mathrm{sign}(a-1)\right\}$$

Plot the result for the values `-10 < a < 10`:

```
plot(S, a = -10..10)
```



Evaluate the result for `a = 5` using the operator `|` or its functional form `evalAt`:

```
S | a = 5
```

$$\left\{-4^{1/5}\right\}$$

Approximate the result with a floating-point value using `float`:

```
float(%)
```

$$\{-1.319507911\}$$

Parameters

`eq`

A single equation or an inequality of type `"_equal"`, `"_less"`, `"_leequal"`, or `"_unequal"`, or an equation in the form of membership (`_in`). Alternatively, any Boolean expression composed of equations or inequalities by the operators `"_and"`, `"_or"`, and `"_not"`. Also, the solver accepts an arithmetical expression and regards such expression as an equation without the right side. (Internally, the solver assumes that the right side is equal to 0.)

`x`

The indeterminate for which you solve an equation, an inequality of a system: an identifier or an indexed identifier

`a, b`

Arithmetical expressions

`vars`

A nonempty set or list of indeterminates for which you solve an equation, an inequality, or a system

eqs

A set, list, array, or table of equations, inequalities, arithmetical expressions, or any combination of these objects. The solver regards expressions as equations without the right side. (Internally, the solver assumes that the right side is equal to 0.)

ODE

An ordinary differential equation: an object of the `ode` type.

REC

A recurrence equation: an object of the `rec` type.

Options

MaxDegree

Option, specified as `MaxDegree = n`

Do not use explicit formulas that involve radicals when solving polynomial equations of degree larger than `n`. Here `n` is a positive integer. By default, `n = 2`.

This option enables and disables the use of explicit formulas for the roots of polynomials. This option does not affect other methods, such as factorization. For polynomial equations, the given maximal degree `n` refers to the factors of the polynomials, not to the input polynomial.

When you solve a fifth- or higher-order polynomial equation, the solver might be unable to return the solution explicitly. In general, there are no explicit expressions for the roots of polynomials of degrees higher than 4. Setting the `MaxDegree` option to 4 or a higher value makes no difference.

BackSubstitution

Option, specified as `BackSubstitution = b`

Enable or disable backward substitutions when solving algebraic systems. The value `b` must be `TRUE` or `FALSE`. By default, `b = TRUE`.

`BackSubstitution` only affects the results returned as sets of lists.

`Multiple`

With this option, `solve` returns a set of type `Dom::Multiset`, indicating the multiplicity of polynomial roots.

The solver ignores this option if the input is not a polynomial expression or equation, or if the solution is of the `RootOf` type.

`VectorFormat`

Return a set of vectors when solving a system of equations for a list of variables.

`PrincipalValue`

With this option, the solver returns only one solution. The solver returns this solution as a set with one element. If an equation does not have a solution, the solver returns an empty set.

If the solver cannot find any solution and cannot prove that solutions do not exist, it returns an unresolved symbolic call to `solve`. For example, if the set of solutions is a piecewise function, and there are no elements that belongs to all cases, the solver cannot find a solution.

You also can use this option to solve equations for more than one variable. In this case, the solver returns a set that contains one list. This nested structure represents a solution vector.

`Domain`

Option, specified as `Domain = d`

Return the set of all solutions that are elements of `d`. Here `d` must represent a subset of the complex numbers (for example, real numbers `Dom::Real` or integers `Dom::Integer`). Alternatively, `d` can be a domain over which you can factor polynomials (for example, `d` can be a finite field). In this case, you can use this option only when solving polynomial equations. Without this option, the solver returns all solutions in the set of complex numbers.

You can solve an equation or a system over the following domains:

- Subsets of the set of complex numbers `C_`.
- Domains over which you can factor polynomials. You can use these domains only when solving polynomial equations.

A subset of `C_` is any kind of set returned by `solve`. Instead of `C_`, `R_`, `Q_`, and `Z_`, you also can use the corresponding domains of the domains package `Dom::Complex`, `Dom::Real`, `Dom::Rational`, and `Dom::Integer`.

You can overload the solver for your custom domains by adding the `domsolve` method to those domains. If this method does not exist, MuPAD uses the `solve_eq` method to solve equations. The `solve_eq` method does not accept systems as arguments. Finally, if the `solve_eq` method does not exist, MuPAD uses the `solve_poly` method to solve polynomials. The `solve_poly` method accepts only polynomials as first arguments. This method regards any first argument of `solve` that cannot be converted to a polynomial as illegal.

The calling syntax for the `domsolve`, `solve_eq`, and `solve_poly` methods is `domsolve(eq, var, options)`. Here `var` is the same argument as in `solve`, and `options` is a table of options. For the `domsolve` method, `eq` is also the same as in `solve`. For the `solve_eq` method, `eq` must be an arithmetical expression. For `solve_poly`, `eq` must be a polynomial.

You cannot solve equations and systems in more than one variable over domains.

`IgnoreProperties`

Include solutions that are not consistent with the properties of the variable `x`.

`Real`

Return only the solutions for which every subexpression of `eq` represents a real number. Also, assume that every subexpression independent of `x` represents a real number.

With this option, the solver assumes that every subexpression independent of `x` represents a real number. In particular, the solver assumes that all symbolic parameters are real. When you use `Real`, the solver returns only the solutions for which every subexpression of `eq` is real. See “Example 7” on page 1-1962.

When you use this option, MuPAD restricts the domain of every function to real numbers. For example, it does not support the logarithms of negative numbers. For all returned solutions `x`, the input is defined over the real numbers.

This option is particularly useful for solving inequalities. Inequalities hold only when both sides represent real values.

This option does not affect some systems.

IgnoreSpecialCases

If a solution requires case analysis, ignore cases for which one or more parameters in the equation are supposed to be an element of a comparatively small set (for example, with this option, MuPAD can ignore a membership in a fixed finite set or a set of integers $\mathbb{Z}$).

With this option, the solver tries to reduce the number of branches in piecewise objects. MuPAD finds equations and memberships in comparatively small sets. First, MuPAD tries to prove such equations and memberships by using the property mechanism. If the property mechanism proves an equation or a membership is true, MuPAD keeps that statement. Otherwise, MuPAD can replace that statement with the value FALSE. For example, if the property mechanism cannot prove that a denominator is equal to zero, MuPAD regards this denominator as nonzero. This option can significantly reduce the number of piecewise objects in a solution.

IgnoreAnalyticConstraints

Apply purely algebraic simplifications to expressions and equations. With this option, the solver applies the following rules to the expressions on both sides of an equation:

- $ln(a) + ln(b) = ln(a\,b)$ for all values of a and b. In particular:

  $(a\,b)^c = e^{c\,\ln(a\,b)} = e^{c\,(\ln(a)+\ln(b))} = a^c\,b^c$ for all values of a, b, and c.
- $ln(a^b) = b\,ln(a)$ for all values of a and b. In particular:

  $(a^b)^c = e^{b\,c\,\ln(a)} = e^{\ln(a)^{b\,c}} = a^{b\,c}$ for all values of a, b, and c.
- If f and g are standard mathematical functions and $f(g(x)) = x$ for all small positive numbers, $f(g(x)) = x$ is assumed to be valid for all complex x. In particular:
- - $\ln(e^x) = x$.
  - $arcsin(sin(x)) = x$, $arccos(cos(x)) = x$, $arctan(tan(x)) = x$.
  - $arcsinh(sinh(x)) = x$, $arccosh(cosh(x)) = x$, $arctanh(tanh(x)) = x$.
  - $W_k(x\,e^x) = x$ for all values of k.
- The solver can multiply both sides of an equation by any expression except 0.
- The solutions of polynomial equations must be complete.

Using this option, you can get simpler solutions for equations for which the direct call of the solver returns complicated results. Note that with this option the solver does not

verify the correctness and completeness of the result. See “Example 8” on page 1-1962 and “Example 9” on page 1-1964.

`DontRewriteBySystem`

Do not transform an equation to an equivalent system of equations. This option decreases the running time. With this option, the solver cannot solve some equations.

This option does not allow the solver to replace an equation with the equivalent system of equations. Typically, MuPAD replaces an equation by an equivalent system of equations when solving equations with nested roots. Solving the resulting system can be slow. Use this option to improve performance of the solver. When you use `DontRewriteBySystem`, the solver cannot solve some of the equations that it can solve without this option.

`NoWarning`

Suppress all warning messages.

Return Values

If `x` is an identifier, `solve(eq, x)` returns an object that represents a mathematical set (see the “Details” section). If `x` is a set or a list, or if you omit `x`, a call to `solve` returns a set of lists. Each list consists of equations. The left side of each equation is one of the variables for which you solve an equation, an inequality of a system. In this case, `solve` also can return an expression of the form `x in S`, where `x` is a list of variables, and `S` is a set of vectors. When you solve a system providing the list of variables and the `VectorFormat` option, the solver returns a set of vectors.

Overloaded By

eq

See Also

MuPAD Functions

`isolate` | `linsolve` | `numeric::linsolve` | `numeric::solve` | `RootOf`

sort

Sort a list

Syntax

```
sort(list, <f>)
```

Description

`sort(list)` returns a sorted copy of the list.

`sort` sorts the list in ascending order.

If you do not specify a procedure `f`, the `sort` command uses the following rules for sorting the lists:

- The command sorts a list of real numbers (`Type::Real`) numerically.
- The command sorts a list of character strings alphabetically.
- The command sorts an outer list containing inner lists with numeric first entries by these numeric first entries. See “Example 4” on page 1-1979.
- In all other cases, the command sorts a list according to the internal order: `sort(list)` is equivalent to `sort(list, sysorder)`. All MuPAD sessions use the same internal order. Between different versions of MuPAD, internal order might change.

When you sort strings, uppercase letters have a preference over lowercase letters. For example, `Z` appears before `abc`.

You can specify a procedure `f` to define the sorting criteria. `sort` calls the procedure `f` for every pair of the entries of the list. `f` must return a Boolean expression that the `bool` command can evaluate to `TRUE` or `FALSE`. If for the pair of entries the procedure `f(x, y)` returns `TRUE`, the sorted list displays `x` to the left of `y`. Otherwise, `x` appears to the right of `y`. The entries of the sorted list `L := sort(list, f)` satisfy `bool(f(L[i], L[j])) = TRUE` for `i < j`.

If two entries of a list are equal by the sorting criteria `f`, the `sort` command can swap these entries. For example, if you sort polynomials by their degrees, the `sort` command

can return the polynomials with the same degree in the order different from their order in the input.

`sort` can be overloaded by kernel domains. For example, use the function `DOM_SET::sort` to sort sets. See “Example 3” on page 1-1978

The average runtime to sort a list containing n entries is $O(n\ log(n))$.

Examples

Example 1

The `sort` command sorts real numbers (type `Type::Real`) numerically:

```
sort([4, -1, 2/3, 0.5])
```

$$\left[-1,\ 0.5,\ \frac{2}{3},\ 4\right]$$

The `sort` command sorts strings alphabetically:

```
sort(["chip", "alpha", "Zip"])
```

$$\text{["Zip", "alpha", "chip"]}$$

If a list contains other types of objects the `sort` command sorts a list according to the internal order. The command also applies internal order to sort the lists with mixed types of entries:

```
sort([4, -1, 2/3, 0.5, "alpha"])
```

$$\left[-1,\ 0.5,\ \frac{2}{3},\ 4,\ \text{"alpha"}\right]$$

```
sort([4, -1, 2/3, 0.5, I])
```

$$\left[-1,\ 0.5,\ \frac{2}{3},\ 4,\ i\right]$$

Example 2

Define your own criteria to sort a list. For example, sort the entries by their absolute values:

```
sort([-2, 1, -3, 4], (x, y) -> abs(x) < abs(y))
```

$$[1, -2, -3, 4]$$

Example 3

When sorting sets, the `sort` command returns a list as a result:

```
sort({3, 12, 5, 30, 6, 43})
```

$$[3, 5, 6, 12, 30, 43]$$

The sorted set is equivalent to the corresponding sorted list:

```
bool(sort({3, 12, 5, 30, 6, 43}) = sort([3, 12, 5, 30, 6, 43]))
```

$$\text{TRUE}$$

To sort other data types, implement a `sort`-slot for them:

```
unprotect(DOM_INT):
DOM_INT::sort :=
proc(n)
  local str, i;
begin
  str := expr2text(n);
  text2expr(_concat(op(sort([str[i] $ i = 1..length(str)]))))
end:
sort(1703936)
```

$$133679$$

```
delete DOM_INT::sort:  protect(DOM_INT):
```

Example 4

If the list contains lists as entries, and all the inner lists start with numbers, the `sort` command uses these numbers to sort the outer list:

```
sort([[10 - i, i*x^i] $ i = 1..9])
```

$$[[1, 9\,x^9], [2, 8\,x^8], [3, 7\,x^7], [4, 6\,x^6], [5, 5\,x^5], [6, 4\,x^4], [7, 3\,x^3], [8, 2\,x^2], [9, x]]$$

Compare the sorted list with the internal order of its entries:

```
sort([[10 - i, i*x^i] $ i = 1..9], sysorder)
```

$$[[8, 2\,x^2], [7, 3\,x^3], [6, 4\,x^4], [5, 5\,x^5], [4, 6\,x^6], [3, 7\,x^7], [2, 8\,x^8], [1, 9\,x^9], [9, x]]$$

Parameters

`list`

A list of arbitrary MuPAD objects

`f`

A procedure defining the ordering

Return Values

List.

Overloaded By

list

Algorithms

To implement the `sort` function, MuPAD uses a version of the Quicksort algorithm.

See Also

MuPAD Functions
`prog::sort` | `sysorder`

split

Split an object

Syntax

```
split(object, f, <p1, p2, …>)
```

Description

`split(object, f)` splits the object into a list of three objects. The first list entry is an object consisting of those operands of the input object that satisfy a criterion defined by the procedure `f`. The second list entry is built from the operands that violate the criterion. The third list entry is built from the operands for which it is unknown whether the criterion is satisfied.

The function `f` must return a value that can be evaluated to one of the Boolean values `TRUE`, `FALSE`, or `UNKNOWN`. It may either return one of these values directly, or it may return an equation or an inequality that can be simplified to one of these values by the function `bool`.

The function `f` is applied to all operands`x` of the input object via the call `f(x, p1, p2, ...)`. Depending on the result `TRUE`, `FALSE`, or `UNKNOWN`, this operand is inserted into the first, the second, or the third output object, respectively.

The output objects are of the same type as the input object, i.e., a list is split into three lists, a set into three sets, a table into three tables etc.

If the input object is an expression sequence, then neither the input sequence nor the output (a list containing three sequences) are flattened.

Also “atomic” objects such as numbers or identifiers can be passed to `split` as first argument. Such objects are handled like sequences with a single operand.

Examples

Example 1

The following command checks which of the integers in the list are prime:

```
split([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], isprime)
```

$$[[2, 3, 5, 7], [1, 4, 6, 8, 9, 10], []]$$

The return value is a list of three lists. The first list contains the prime numbers, the second list contains all other numbers. The third list is empty, because for any number of the input list, it can be decided whether it is prime or not.

Example 2

With the optional arguments `p1, p2, ...` one can use functions that need more than one argument. For example, `contains` is a handy function to be used with `split`. The following call splits a list of sets into those sets that contain `x` and those that do not:

```
split([{a, x, b}, {a}, {1, x}], contains, x)
```

$$[[\{a, b, x\}, \{1, x\}], [\{a\}], []]$$

The elements of the returned list are of of type `DOM_LIST`, because the given expression was a list. If the given expression is of another type, e.g., `DOM_SET`, also the elements of the result are of type `DOM_SET`, too:

```
split({{a, x, b}, {a}, {1, x}}, contains, x)
```

$$[\{\{1, x\}, \{a, b, x\}\}, \{\{a\}\}, \varnothing]$$

Example 3

We use the function `is` to split an expression sequence into sub-sequences. This function returns `UNKNOWN` if it cannot derive the queried property:

```
split((-2, -1, a, 0, b, 1, 2), is, Type::Positive)
```

$$[1, 2, -2, -1, 0, a, b]$$

Example 4

We split a table of people marked as male or female:

```
people := table("Tom" = "m", "Rita" = "f", "Joe" = "m"):
[male, female, dummy] := split(people, has, "m"):

male
```

"Joe"	"m"
"Tom"	"m"

```
female
```

"Rita"	"f"

```
dummy
```

$\top$

```
delete people, male, female, dummy:
```

Parameters

`object`

A list, a set, a table, an expression sequence, or an expression of type `DOM_EXPR`

`f`

A procedure returning a Boolean value

`p1, p2, …`

Any MuPAD objects accepted by `f` as additional parameters

Return Values

List with three objects of the same type as the input object.

Overloaded By

object

See Also

MuPAD Functions
`map` | `op` | `select` | `zip`

sqrt

Square root function

Syntax

```
sqrt(z)
```

Description

`sqrt(z)` represents the square root of *z*.

$x = \sqrt{z}$ represents the solution of $x^2 = z$ that has a nonnegative real part. In particular, it represents the positive root for real positive *z*. For real negative *z*, it represents the complex root with positive imaginary part.

A floating-point result is returned for floating-point arguments. Note that the branch cut is chosen as the negative real semi-axis. The values returned by `sqrt` jump when crossing this cut. Cf. “Example 2” on page 1-1986.

Certain simplifications of the argument may occur. In particular, positive integer factors are extracted from some symbolic products. Cf. “Example 3” on page 1-1987.

Note that $\sqrt{z}$ cannot be simplified to *x* for all complex numbers (e.g., $\sqrt{x^2} = -x$ for real $x < 0$). Cf. “Example 4” on page 1-1987.

Mathematically, `sqrt(z)` coincides with `z^(1/2) = _power(z,1/2)`. However, `sqrt` provides more simplifications than `_power`. Cf. “Example 5” on page 1-1987.

Environment Interactions

When called with a floating-point argument, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
sqrt(2), sqrt(4), sqrt(36*7), sqrt(127)
```

$$\sqrt{2},\ 2,\ 6\sqrt{7},\ \sqrt{127}$$

```
sqrt(1/4), sqrt(1/2), sqrt(3/4), sqrt(25/36/7), sqrt(4/127)
```

$$\frac{1}{2},\ \frac{\sqrt{2}}{2},\ \frac{\sqrt{3}}{2},\ \frac{5\sqrt{7}}{42},\ \frac{2\sqrt{127}}{127}$$

```
sqrt(-4), sqrt(-1/2), sqrt(1 + I)
```

$$2\,i,\ \frac{\sqrt{2}\,i}{2},\ \sqrt{1+i}$$

```
sqrt(x), sqrt(4*x^(4/7)), sqrt(4*x/3), sqrt(4*(x + I))
```

$$\sqrt{x},\ 2\,x^{2/7},\ \sqrt{\frac{4\,x}{3}},\ 2\sqrt{x+i}$$

Example 2

Floating point values are computed for floating-point arguments:

```
sqrt(1234.5), sqrt(-1234.5), sqrt(-2.0 + 3.0*I)
```

$$35.13545218,\ 35.13545218\,i,\ 0.8959774761 + 1.674149228\,i$$

A jump occurs when crossing the negative real semi axis:

```
sqrt(-4.0), sqrt(-4.0 + I/10^100), sqrt(-4.0 - I/10^100)
```

$$2.0\,i,\ 2.5\,10^{-101}+2.0\,i,\ 2.5\,10^{-101}-2.0\,i$$

Example 3

The square root of symbolic products involving positive integer factors is simplified:

```
sqrt(20*x*y*z)
```

$$2\sqrt{5}\sqrt{x\,y\,z}$$

Example 4

Square roots of squares are not simplified, unless the argument is real and its sign is known:

```
sqrt(x^2*y^4)
```

$$\sqrt{x^2\,y^4}$$

```
assume(x > 0): sqrt(x^2*y^4)
```

$$x\sqrt{y^4}$$

```
assume(x < 0): sqrt(x^2*y^4)
```

$$-x\sqrt{y^4}$$

Example 5

`sqrt` provides more simplifications than the `_power` function:

```
sqrt(4*x), (4*x)^(1/2) = _power(4*x, 1/2)
```

$$2\sqrt{x},\ \sqrt{4x} = \sqrt{4x}$$

Parameters

z

An arithmetical expression

Return Values

Arithmetical expression.

Overloaded By

z

See Also

MuPAD Functions

`_power` | `isqrt` | `numlib::issqr` | `surd`

strmatch

Match pattern in character string

Syntax

```
strmatch(text, pattern, <Index>, <ReturnMatches>, <All>)
```

Description

`strmatch(text, pattern)` checks whether `text` matches the regular expression `pattern`.

`strmatch` performs regular expression matching on strings, via the ICU library. The pattern can contain wildcards forming a perl-compatible regular expression. In these expressions, most characters represent themselves. For example, `"a"` matches `"a"`. For the list of exceptions, see “Algorithms” on page 1-2002.

The library `stringlib` provides more functions for handling strings. For details, see “Operations on Strings”.

Examples

Example 1

Most characters simply match themselves:

```
s := "Hamburg": strmatch(s, "Hamburg")
```

```
     TRUE
```

`strmatch` typically matches substrings:

```
strmatch(s, "Ham"), strmatch(s, "burg")
```

```
     TRUE, TRUE
```

```
strmatch("Ham", "Hamburg")
```

```
    FALSE
```

```
delete s:
```

Example 2

A dot (`.`) is a placeholder for any character except `"\n"`:

```
strmatch("abcd", "a.c"), strmatch("ab\ncd", "ab.")
```

```
    TRUE, FALSE
```

To match an actual dot, use `"\\."`:

```
strmatch("abcd", "a\\.c"),
strmatch("a.cd", "a\\.c")
```

```
    FALSE, TRUE
```

A dot, like all special characters, has its special role only in the second argument of `strmatch`:

```
strmatch("a.c", "abc")
```

```
    FALSE
```

With the `s` modifier, you can use a dot to match newlines:

```
strmatch("abcd", "(?s)a.c"), strmatch("ab\ncd", "(?s)ab.")
```

```
    TRUE, TRUE
```

A dot matches only a single character:

```
strmatch("abcd", "a.d"), strmatch("abcd", "a.b")
```

```
    FALSE, FALSE
```

Example 3

By default, `strmatch` only checks for a match and returns a Boolean value:

```
strmatch("aaaba", "a"), strmatch("aaaba", "c")
```

```
TRUE, FALSE
```

To return the first place where a match occurs, use `Index`:

```
strmatch("aaaba", "a", Index),
strmatch("aaaba", "c", Index)
```

```
[1, 1], FALSE
```

To return the matched substrings, use `ReturnMatches`. This option is helpful when you match complicated expressions.

```
strmatch("aaaba", "a", ReturnMatches),
strmatch("aaaba", "c", ReturnMatches)
```

```
"a", FALSE
```

To find more than one match, use `All`:

```
strmatch("aaaba", "a", All),
strmatch("aaaba", "c", All)
```

```
{"a"}, ∅
```

This expression has several matches because a dot matches any character:

```
strmatch("aaaba", "a.", All)
```

```
{"aa", "ab"}
```

`All` implies `ReturnMatches` unless you also use `Index`:

```
strmatch("aaaba", "a", All, Index)
```

```
{[1, 1], [2, 2], [3, 3], [5, 5]}
```

Combine all three options:

```
strmatch("aaaba", "a", All, Index, ReturnMatches)
```

```
{[1, 1, "a"], [2, 2, "a"], [3, 3, "a"], [5, 5, "a"]}
```

Example 4

By default, `strmatch` matches substrings. To look only for matches at the beginning and end of the string, use the caret (^) and dollar ($) characters, respectively:

```
strmatch("abcd", "a"),
strmatch("abcd", "c"),
strmatch("abcd", "d"),
strmatch("abcd", "abcd")
```

```
TRUE, TRUE, TRUE, TRUE
```

```
strmatch("abcd", "^a"),
strmatch("abcd", "^c"),
strmatch("abcd", "^d"),
strmatch("abcd", "^abcd")
```

```
TRUE, FALSE, FALSE, TRUE
```

```
strmatch("abcd", "a$"),
strmatch("abcd", "c$"),
strmatch("abcd", "d$"),
strmatch("abcd", "abcd$")
```

```
FALSE, FALSE, TRUE, TRUE
```

```
strmatch("abcd", "^a$"),
strmatch("abcd", "^c$"),
strmatch("abcd", "^d$"),
strmatch("abcd", "^abcd$")
```

```
FALSE, FALSE, FALSE, TRUE
```

Using the `m` modifier, you can change the meaning from the beginning or end of a string to the beginning or end of a line:

```
s := "ab\ncd":

strmatch(s, "b$"),
strmatch(s, "(?m)b$")
```

```
FALSE, TRUE
```

Example 5

Specify alternative patterns to match by using the vertical bar (|):

```
strmatch("abcd", "abc|xyz")
```

```
TRUE
```

```
strmatch("abcd", "a|f|j")
```

```
TRUE
```

`strmatch` treats all characters between the vertical bars as one of the alternative patterns. To limit the extent of alternatives, use parentheses:

```
strmatch("abcd", "ab(c|xy)z"),
strmatch("abcd", "ab(c|xy)(z|d)")
```

```
FALSE, TRUE
```

When you use the `ReturnMatches` option, `strmatch` returns the substrings matched by each pair of parentheses:

```
strmatch("abcd", "ab(c|xy)(z|d)", ReturnMatches)
```

```
["abcd", "c", "d"]
```

With alternatives, `strmatch` can find several matches:

```
strmatch("abracadabra", "a(b|c|d)", All)
```

```
{["ab", "b"], ["ac", "c"], ["ad", "d"]}
```

To group alternatives without returning matches, use `(?:...)`:

```
strmatch("abracadabra", "a(?:b|c|d)", All)
```

```
{"ab", "ac", "ad"}
```

To match for the characters `"|"`, `"("`, and `")"`, use `\\` before the character when you specify the pattern to match:

```
strmatch("ab(c)d", "\\((c|d)\\)", ReturnMatches)
```

```
["(c)", "c"]
```

Example 6

Use a question mark (?) to indicate that a subexpression (a single character or a group of characters in parentheses or brackets) is optional:

```
strmatch("abcd", "abc?d"),
strmatch("abd", "abc?d")
```

```
TRUE, TRUE
```

Use an asterisk (*) to indicate that a subexpression can be repeated an arbitrary number of times, including zero:

```
strmatch("abcd", "a.*d"),
strmatch("abcd", "a.*c")
```

```
TRUE, TRUE
```

Use a plus sign (+) to indicate that a subexpression can be repeated an arbitrary number of times, excluding zero:

```
strmatch("abcd", "a.+d"),
strmatch("abcd", "a.+b")
```

```
TRUE, FALSE
```

When you use the asterisk or the plus sign in the pattern to match, `strmatch` finds the first match going from left to right, and then returns the longest substring that satisfies the matching pattern:

```
strmatch("abracadabra", "a.*a", ReturnMatches)
```

```
"abracadabra"
```

By appending another question mark, you can switch the asterisk and plus sign to “non-greedy” matching:

```
strmatch("abracadabra", "a.*?a", ReturnMatches)
```

```
"abra"
```

This does not return the shortest match (which would have been `"aca"` or `"ada"`). The call returns the first match looking from left to right from the starting position.

Example 7

Use curly braces to specify a number of repetitions of a subexpression:

```
strmatch("abracadabra", "(a(b|c|d)){2}"),
strmatch("abracadabra", "(a(b|c|d)){3}"),
strmatch("abracadabra", "(a(b|c|d)){4}")
```

```
TRUE, TRUE, FALSE
```

These repetitions must be adjacent:

```
strmatch("abracadabra", "(abr){2}")
```

```
FALSE
```

To get nonadjacent repetitions, use `".*"`. This combination means "anything without newlines".

```
strmatch("abracadabra", "(abr.*){2}")
```

```
    TRUE
```

Example 8

To indicate a range of possible repetitions, use a comma inside curly braces. For example, select the expressions representing binary numbers with three to five digits:

```
select(["11001", "1100111", "11", "11021"], strmatch, "^((0|1){3,5})$")
```

```
    ["11001"]
```

Here `{3,5}` specifies the range. You can omit the second number to remove the upper bound. For example, `{3,}` indicates that there must be three or more repetitions.

The following regular expression checks whether there is an `"a"` followed by at least three letters `"b"` followed by a `"c"` somewhere in the input string:

```
strmatch("abcd", "ab{3,}c"),
strmatch("abbbcd", "ab{3,}c"),
strmatch("abcdabbbc", "ab{3,}c")
```

```
    FALSE, TRUE, TRUE
```

By default, when `strmatch` looks for repetitions, it returns the longest matching substring. Use a question mark to return the first match instead of the longest one:

```
strmatch("abcdabcdabcd", "a.{2,8}d", ReturnMatches),
strmatch("abcdabcdabcd", "a.{2,8}?d", ReturnMatches)
```

```
    "abcdabcd", "abcd"
```

Example 9

Characters enclosed in brackets ([ ]) form a "character class", which matches any of the characters in the class. This behavior is similar to an alternation between these characters.

```
strmatch("abc", "ab[cde]"),
strmatch("abd", "ab[cde]"),
strmatch("aba", "ab[cde]")
```

```
TRUE, TRUE, FALSE
```

Inside character classes, special characters are completely different. Dots, asterisks, plus and dollar signs, parentheses, and curly braces match themselves, but a character class starting with a caret is "negated" and matches any character *not* listed:

```
strmatch("abcd", "[^ab]", All)
```

```
{"c", "d"}
```

If a caret is not the first character in a class, then it represents itself:

```
strmatch("x^2", "[*^]2")
```

```
TRUE
```

If a dash (-) is not the first character in a class (apart from the caret), then it specifies a range of characters. Thus, to find a number with at least five digits, you can specify the pattern as follows:

```
strmatch("x = 123456...", "[0-9]{5,}", ReturnMatches)
```

```
"123456"
```

The exact meaning of a range depends on the language settings of your computer. Technically, it depends on the "collating", which can be different for the same language on different versions of the same operating system. For example, `"[a-z]"` can match only lowercase ASCII characters on one computer, while on the second one it also matches the uppercase characters from A through Y, and on the third one includes the uppercase characters from B through Z. For this reason, the best practice is using the named character classes instead:

```
strmatch("some words", "[[:word:]]+", All)
```

```
{"some", "words"}
```

Example 10

Some character classes have a short form, such as `"\\x"`, where `x` is `w`, `W`, `s`, `S`, `d`, or `D`. The uppercase letters mean the negation of the lowercase letters.

```
strmatch("abcd", "\\w"),
strmatch("abcd", "\\W"),
strmatch("abcd", "\\d"),
strmatch("abcd", "\\D")
```

```
TRUE, FALSE, FALSE, TRUE
```

Here, negation means that a character does not match whatever is negated:

```
strmatch("abcd 1", "\\w"),
strmatch("abcd 1", "\\W"),
strmatch("abcd 1", "\\d"),
strmatch("abcd 1", "\\D")
```

```
TRUE, TRUE, TRUE, TRUE
```

Use `"\\b"` to look for words starting with an `a`. The pattern `"\\b"` is a zero-width expression matching the place between a "word" and the spaces surrounding it (or the beginning and end of string).

```
strmatch("abc cbd cba (aa) b", "\\ba\\w*", All)
```

```
{"aa", "abc"}
```

You can also use `"\\b"` to match the end of a word:

```
strmatch("abc cbd cba (aa) b", "\\w*a\\b", All)
```

```
{"aa", "cba"}
```

Example 11

You can change the behavior of `strmatch` with modifier flags. For example, the `i` modifier enables case-insensitive matching. (The precise effects of case-insensitive

matching depend on your language settings, for example, most English computers do not treat the German umlauts ä and Ä as being the same up to case.) To enable case-insensitive matching for the whole expression, prefix it with "(?i)":

```
strmatch("ABC", "(?i)ab")
```

```
TRUE
```

To limit the effect of the modifier to some part of the expression, use "(?i:...)":

```
strmatch("ABC", "(?i:a)b"),
strmatch("abc", "(?i:a)b"),
strmatch("Abc", "(?i:a)b")
```

```
FALSE, TRUE, TRUE
```

Example 12

`strmatch` with `ReturnMatches` or `All` (without `Index`) returns the matched substrings. You can also return parts of those substrings. For example, extract all function names from this expression. To identify function names, note that an opening parenthesis or a space and an opening parenthesis follows every function name.

```
s := "f(sin (x) + abc + def(x))":
strmatch(s, "\\b\\w+\\s*\\(", All)
```

```
{"def(", "f(", "sin ("}
```

To extract the function names themselves, use this command:

```
map(strmatch(s, "\\b(\\w+)\\s*\\(", All), op, 2)
```

```
{"def", "f", "sin"}
```

Regular expressions can contain zero-width assertions. These assertions ensure that something does or does not follow, without actually including it or moving the conceptual pointer behind it. Therefore, the more efficient approach is to wrap the corresponding expression in "(?=...)":

```
strmatch(s, "\\b\\w+(?=\\s*\\()", All)
```

```
{"def", "f", "sin"}
```

Example 13

Regular expressions can also make zero-width assertions with respect to the preceding text. Such assertions must have a fixed width. For example, extract the amount of money mentioned in this string:

```
s := "In March 2005, we've spent $1192.23 on light.":
strmatch(s, "(?<=\\$)\\d+(?:\\.\\d\\d)?", All)
```

```
{"1192.23"}
```

Example 14

To detect the positions of the matches in the input string, use the `Index` option. The returned list contains two numbers: the beginning and end of the match.

```
strmatch("abc", "b", Index)
```

```
[2, 2]
```

If no match is found, `strmatch` returns `FALSE`:

```
strmatch("abc", "d", Index)
```

```
FALSE
```

If you use both `Index` and `ReturnMatches`, then `strmatch` returns indices followed by the matched subexpressions:

```
strmatch("abc", "b.", ReturnMatches, Index)
```

```
[2, 3, "bc"]
```

Example 15

If you use `All`, then the return value is a set:

```
strmatch("abc", ".", All),
strmatch("abc", ".", Index, All),
strmatch("abc", ".", ReturnMatches, All)
```

```
{"a", "b", "c"}, {[1, 1], [2, 2], [3, 3]}, {"a", "b", "c"}
```

Parameters

`text`, `pattern`

character strings

Options

`Index`

Return the position of the match. If there are no matches, `strmatch` returns `FALSE`. Otherwise, it returns the position of the match as a list of two integers, `[i, j]`, such that `text[i..j]` is the matched substring.

`ReturnMatches`

Return the matched substrings. If the regular expression contains groups (subexpressions in parentheses), then `strmatch` returns lists containing the matched substring and the strings matched by the groups, in order of opening parentheses.

`All`

Return all matches that `strprint` can find. By default, `strmatch` returns only the first match. If you do not use `Index`, then the `All` option also implies `ReturnMatches`.

Return Values

Without options, `TRUE` or `FALSE` is returned. With `Index`, a list of two nonnegative integers or `FALSE` is returned. With option `ReturnMatches`, a string or a list of strings is returned, depending on whether the pattern contains groups. With both `Index` and

`ReturnMatches`, a list starting with the indices of the match, followed by the string or strings of `ReturnMatches`, is returned. With option `All`, a set is returned.

Overloaded By

pattern, text

Algorithms

- A dot (`.`) matches any character, except `"\n"`. With the `s` modifier, a dot matches any character. See “Example 2” on page 1-1990.
- A caret (`^`) matches the beginning of a line. A dollar (`$`) matches the end of a line. Typically, `^` and `$` mark the beginning and end of the string, but with the `m` modifier they also can appear after or before a `"\n"`. See “Example 4” on page 1-1992.
- A pattern enclosed in parentheses (`()`) is considered “grouped.”
- A vertical bar (`|`) between two characters or groups (sub-regexes) lets you specify alternative matching patterns. Any one of the alternatives matching is sufficient. See “Example 5” on page 1-1993.
- A sub-regex followed by a number *n* enclosed in `{}` must match exactly *n* times.

  A sub-regex followed by `{n,}` must match at least *n* times.

  A sub-regex followed by `{n,m}` must match at least *n* and at most *m* times.

  In any other context, `{` and `}` are treated as normal characters.

  See “Example 7” on page 1-1995.
- Following a sub-regex, a question mark (`?`) works as `{0,1}`, making the sub-regex optional.

  A plus sign (`+`) in this context works as `{1,}` and allows an arbitrary positive number of repetitions.

  An asterisk after an expression is equivalent to `{0,}` and allows an arbitrary number of repetitions, including zero.

  See “Example 6” on page 1-1994.

- By default, `{n,}` and its three shorthand forms match as many characters as possible. By following them with another question mark (for example, `"a(b[cd]){2,}?bd"`, `"(0|1)*?12"`), you can specify that `strmatch` must return the lowest number of characters consistent with the remainder of the pattern.
- While a backslash (which must be typed as `"\\"`) escapes any special character (including itself), it makes some characters following it special. See “Example 10” on page 1-1998.
  - `"\\w"` matches a “word” character (alphanumeric or underline).
  - `"\\W"` matches a character not matched by `"\\w"`.
  - `"\\s"` matches a white-space character (space, or tabulator, or, if the `s` modifier is active, also an end-of-line character).
  - `"\\S"` matches a character not matched by `"\\s"`.
  - `"\\d"` matches a digit.
  - `"\\D"` matches a nondigit.
  - `"\\b"` matches the place between a word character and a nonword character, for example, the place where a word starts or ends.
  - `"\\B"` is also zero-width, but matches those places where `"\\b"` does not.
  - `"\\A"` and `"\\Z"` match at the beginning and end of the string, respectively. `"\\Z"` ignores a `"\n"` at the end of the string; `"\\z"` behaves like `"\\Z"`, but does not ignore a trailing `"\n"`.
  - `'\\X'` matches a grapheme cluster. For example, the letter `a̅` is a grapheme cluster: it consists of `a` and `̅`. `'\\X'` lets you access `a̅` as one entity.
- Characters enclosed between `[` and `]` form a character class. See “Example 9” on page 1-1996.

  A character class starting with `^` is negated, and matches all the characters not listed. The symbol `^` at any other place in the character class has no special meaning.

  Inside a character class, the special characters, except for a hyphen, do not have any special meaning. If a hyphen (`-`) is not the first character, then it creates a range of characters. The language settings of your operating system (technically speaking, the current locale) affect how `strmatch` interprets this range. Likely, in every language setting `"[0-9]"` represents any digit.

  To specify character classes independent of language settings, use named access to POSIX character classes:

- `"[[:digit:]]"` for any digit.
- `"[[:alpha:]]"` for characters (the language settings define what makes a character).
- `"[[:alnum:]]"` for alphanumerical characters.
- `"[[:word:]]"` for alphanumerical characters plus the underline (_).
- `"[[:punct:]]"` for punctuation characters, such as a dot or a comma.
- `"[[:ascii:]]"` for characters in the ASCII range (decimal codes 32 through 127).
- `"[[:blank:]]"` for horizontal spaces, such as`[ \t]`.
- `"[[:space:]]"` for spaces, including end-of-line.
- `"[[:cntrl:]]"` for control characters, such as newlines. Note that you cannot type most control characters in MuPAD, but they can occur in strings read from files.
- `"[[:graph:]]"` for the class of alphanumeric or punctuation characters, that is, characters with visual graphical representation.
- `"[[:print:]]"` is equivalent to `"[ [:graph:]]"`. It adds the space character to the `graph` class.
- `"[[:lower:]]"` and `"[[:upper:]]"` for the characters that your language settings consider lowercase and uppercase letters. For example, a German system is more likely to know about ä being a lowercase letter than a U.S. system.
- `"[[:xdigit:]]"` matches hexadecimal digits. It is equivalent to `[0123456789aAbBcCdDeEfF]`.

You combine these classes with one another or add characters from one class to another class. For example, you can match septendecimal digits with `"[[:xdigit:]gG]"`.

You can negate posix character classes using a caret. For example, `"[[:^digit:]]"` matches nondigits. This is equivalent to `"[^[:digit:]]"`, but `"[0[:^digit:]]"` to allow any nondigit or zero is more difficult to express otherwise.

- Groups starting with `(?` have special meanings:
  - Groups starting with `(?:` behave like other groups, but do not create output matches for the `ReturnMatches` option.
  - `"(?#text)"` is a comment and effectively ignored.

- Groups starting with `(?X:`, where `X` is one of `i`, `m`, `s`, `x`, locally apply modifiers:
  - `i` causes all pattern matching to be case-insensitive (as defined by the system's locale).
  - `m` causes a “multiline” match, where `^` and `$` match after/before `"\n"` characters in the string.
  - `s` makes the dot match newlines.
  - `x` allows perl-style comments in the pattern. In this case, `strmatch` ignores spaces in most contexts. The # characters start comments that extend to the end of the line.

  When using these options in an outer group, you can disable them by preceding them with a minus sign, as in `"(?-i:aB)"`.
- The string `"(?X)"`, where `X` is one of the characters listed above, switches the corresponding setting on up to the end of the enclosing group.
- `(?=` starts a positive zero-width lookahead assertion. This is a zero-width item (and therefore does not add something to the output) that matches if its contents match at the current position. See “Example 12” on page 1-1999.
- `(?!` starts a zero-width negative look-ahead assertion. It behaves almost identical to `(?=` except it matches if and only if `(?=` does not.
- `(?<=` starts a positive zero-width look-behind assertion, which is like `(?=`, but looking in the other direction. Look-behind assertions must have a fixed width. See “Example 13” on page 1-2000.
- `(?<!` starts a negative zero-width look-behind assertion, which matches if and only if a `(?<=` at the same place does not match.

See Also

MuPAD Functions

`_concat` | `length` | `stringlib::contains` | `stringlib::maskMeta` | `stringlib::pos` | `substring`

strprint

Print into string

Syntax

```
strprint(<All>, <Unquoted>, <NoNL>, <KeepOrder>, object1, object2, …)
```

Description

`strprint(objects)` returns the string `print(objects)` would display on the screen.

`strprint` returns a string that contains the output `print` would have sent to the screen for the same arguments. This string contains `` `\n' `` characters if the output would have consisted of multiple lines.

On Windows systems, each `\n` is preceded by `\r`, because that is the traditional end-of-line combination since that is the end-of-line combination inherited from CP/M. The examples in this documentation assume a system like UNIX. Also note that with Typesetting activated, `\r`, `\n`, `\t`, and `\b` will not be displayed in a string.

All options and dependencies on variables are interpreted as described in the documentation of `print`. Especially, this means `PRETTYPRINT` affects the output of `strprint`. Overloading of `print` is taken into account. See `?print` for details.

Environment Interactions

`strprint` is sensitive to the environment variables `DIGITS`, `PRETTYPRINT`, and `TEXTWIDTH`, and to the output preferences `Pref::floatFormat`, `Pref::keepOrder`, and `Pref::trailingZeroes`.

Examples

Example 1

The string returned by `strprint`, when printed with option `Unquoted`, yields the same output as the operands would have in the first place:

```
s := strprint(a*x^2-7):
print(Unquoted, s)
```

```
   2
a x  - 7
```

This can be used to combine multiple outputs:

```
s1 := strprint(a*x^2-7):
s2 := _concat("-" $ TEXTWIDTH)."\n":
s3 := strprint(sin(1/x)):
print(Unquoted, s1.s2.s3)
```

```
   2
a x  - 7
--------------------------------------------------------------------------------
   / 1 \
sin| - |
   \ x /
```

In the example above, you can see that the output of `strprint` does not contain the spaces usually used for centering. The output in the first example was centered, because it used only a fraction of the text width, while the string `s1.s2.s3` in the second example spans the whole width of the line and is therefore printed flush left.

Example 2

For demonstrative purposes, let us write a domain that puts an expression into a box. We make use of the fact that `strprint` returns strings starting with a newline, of `output::fence` and of indexed assignment to strings:

```
domain box
  print := proc(e)
```

```
            local ex, str, w;
            save TEXTWIDTH;
          begin
            if TEXTWIDTH > 15 then
              TEXTWIDTH := TEXTWIDTH - 4;
            end_if;
            ex := extop(e, 1);
            str := strprint(All, ex);
            w := str[5]+4;
            str := output::fence("| ", " |",
                                 "\n".str[1]."\n",
                                 str[5], str[6]+1);
            str[1..w]     := "+"._concat("-"$w-2)."+";
            str[-w-1..-2] := "+"._concat("-"$w-2)."+";
            str;
          end_proc;

   new := x -> new(dom, x);
end_domain

print(Plain, box(a), box(sin(1/x)))
```

```
       +----------+
+---+  |    / 1 \ |
| a |, | sin| - | |
+---+  |    \ x / |
       +----------+
```

```
print(Plain, box(box(hold(E=m*c^2))))
```

```
+--------------+
| +----------+ |
| |        2 | |
| | E = m c  | |
| +----------+ |
+--------------+
```

Example 3

As a last example, we implement a print-method for matrices over $\mathbb{Z}_5$:

```
M5 := Dom::Matrix(Dom::IntegerMod(5))
```

```
Dom::Matrix(Dom::IntegerMod(5))
```

The standard output function simply puts “mod 5” behind every entry, inherited from the output method of `Dom::IntegerMod(5)`:

```
A := M5([[1,2,3,4],[5,6,7,8],[-2,-3,0,1]])
```

$$\begin{pmatrix} 1 \bmod 5 & 2 \bmod 5 & 3 \bmod 5 & 4 \bmod 5 \\ 0 \bmod 5 & 1 \bmod 5 & 2 \bmod 5 & 3 \bmod 5 \\ 3 \bmod 5 & 2 \bmod 5 & 0 \bmod 5 & 1 \bmod 5 \end{pmatrix}$$

We now replace this method:

```
M5::print := proc(A)
               local str, h1, w1, h, w, b;
             begin
               [str, h1, w1, h, w, b] := strprint(All, expr(A));
               _concat(str, " " $ w, "[mod 5]");
             end_proc:

print(A):


/ 1, 2, 3, 4 \
|            |
| 0, 1, 2, 3 |
|            |
\ 3, 2, 0, 1 /
              [mod 5]
```

Alternatively, we can set the `[mod 5]` right beneath the brackets:

```
M5::print := proc(A)
               local str;
             begin
               str := strprint(expr(A));
               str[-1..-1] := " [mod 5]";
               str
             end_proc:

print(A):
```

```
/ 1, 2, 3, 4 \
|            |
| 0, 1, 2, 3 |
|            |
\ 3, 2, 0, 1 / [mod 5]
```

Parameters

$object_1$, $object_2$, …

Any MuPAD objects

Options

All

When the option `All` is given, `strprint` returns additional information on the string generated by printing. More specifically, it returns a list consisting of

1. the formatted string,
2. the height (in characters) of the first line,
3. the width of the first line,
4. the height of the complete string,
5. the width of the complete string,
6. the baseline, counted from top to bottom.

"Example 2" on page 1-2007 contains sample code that makes use of this information.

Unquoted

Display character strings without quotation marks and with expanded control characters `'\n'`, `'\t'`, and `'\b'`.

NoNL

Like `Unquoted`, but no newline is put at the end. `PRETTYPRINT` is implicitly set to `FALSE`.

KeepOrder

Display operands of sums (of type `"_plus"`) always in the internal order.

Return Values

`DOM_STRING` or a list of a `DOM_STRING` and five integers.

Overloaded By

See Also

MuPAD Functions

`doprint` | `expose` | `expr2text` | `fprint` | `funcenv` | `output::fence` | `print`

subs

Substitute into an object

Syntax

```
subs(f, old = new, <Unsimplified>)
subs(f, old1 = new1, old2 = new2, …, options)
subs(f, [old1 = new1, old2 = new2, …], options)
subs(f, {old1 = new1, old2 = new2, …}, options)
subs(f, table(old1 = new1, old2 = new2, …), options)
subs(f, s1, s2, …, options)
```

Description

`subs(f, old = new)` searches `f` for operands matching `old`, and replaces `old` with `new`. See “Example 1” on page 1-2013.

The `subs` function returns a modified copy of the object. The function does not change the object itself.

By default, the `subs` function does not evaluate the result of a substitution. To enforce evaluation of all modified subexpressions, use the `EvalChanges` option. Also, you can reevaluate the whole returned result by using the `eval` function. Evaluation of the returned result is slower and less efficient than evaluation of the modified subexpressions. See “Example 3” on page 1-2014 and “Example 4” on page 1-2015.

The call `subs(f, old1 = new1, old2 = new2, ...)` applies the specified substitutions in a sequence from left to right (sequential substituton). This call applies each substitution (except for the first substitution) to the result of the previous substitution. See “Example 5” on page 1-2015.

The call `subs(f, [old1 = new1, old2 = new2, ...])` applies all specified substitutions to the operands of the original input object `f` (parallel substitution). This call does not use the results of any previous substitutions. If you specify multiple

substitutions of the same operand, this call computes only the first substitution. Specifying substitutions by lists, sets, or tables invokes parallel substitution. See “Example 6” on page 1-2016.

The call `subs(f, s1, s2, ...)` is a general form of substitution that can combine sequential and parallel substitutions. This call is equivalent to `subs(... subs(subs(f, s1), s2), ...)`. MuPAD treats each substitution step as a sequential or a parallel substitution depending on the form of the parameters `s1`, `s2`, ... See “Example 7” on page 1-2017.

`subs` replaces only those operands that you can access via the function `op` (“syntactical substitution”). To apply a more “semantical” substitution, use the `subsex` function. The `subsex` function also identifies and replaces partial sums and products. See “Example 8” on page 1-2017.

You can use `subs` to replace operands of expression sequences. The `subs` function does not flatten such objects. See “Example 9” on page 1-2018.

If you do not specify substitutions, `subs` returns the original expression without modifications. For example, `subs(f)` returns `f`.

Examples

Example 1

Use the `subs` function to substitute the operands in the following expressions:

```
subs(a + b*a, a = 4)
```

$$4\,b + 4$$

```
subs([a * (b + c), sin(b +c)], b + c = a)
```

$$[a^2, \sin(a)]$$

Example 2

When replacing the sine function in an expression, use the `hold` command to prevent the evaluation of the identifier `sin`:

```
subs(sin(x), hold(sin) = cos);
domtype(hold(sin))
```

$$\cos(x)$$

DOM_IDENT

Otherwise, MuPAD replaces `sin` by its value. The function environment (see `funcenv`) defines the value of `sin`:

```
subs(sin(x), sin = cos);
domtype(sin)
```

$$\sin(x)$$

DOM_FUNC_ENV

Inside the expression `sin(x)`, the 0-th operand `sin` is the identifier, not the function environment:

```
domtype(op(sin(x), 0))
```

DOM_IDENT

Example 3

The `subs` function evaluates the original expression, performs a substitution, but does not evaluate the modified expression:

```
subs(y^2 + sin(x), x = PI)
```

$$y^2 + \sin(\pi)$$

To evaluate the modified subexpression, use the `EvalChanges` option:

```
subs(y^2 + sin(x), x = PI, EvalChanges)
```

$$y^2$$

Alternatively, use the `eval` function to evaluate the result returned by `subs`:

```
S := subs(y^2 + sin(x), x = PI):
eval(S)
```

$$y^2$$

Example 4

The `subs` function with the `EvalChanges` option returns the same results as the evaluation of the whole expression:

```
eval(subs(sin(x + 3 - PI)*numeric::int(_plus(sin(k/y) $ k = 1..5),
                                       y = 0..1), x=-3));
subs(sin(x + 3 - PI)*numeric::int(_plus(sin(k/y) $ k = 1..5),
                                  y = 0..1), x = -3, EvalChanges)
```

$$0$$

$$0$$

The evaluation of the returned result is slower and less efficient than the evaluation of the modified subexpressions:

```
time(eval(subs(sin(x + 3 - PI)*numeric::int(_plus(sin(k/y) $ k = 1..5),
                                            y = 0..1), x = -3)));
time(subs(sin(x + 3 - PI)*numeric::int(_plus(sin(k/y) $ k = 1..5),
                                       y = 0..1), x = -3, EvalChanges))
```

$$12218.75$$

$$6265.625$$

Example 5

The following call results in the sequential substitution $x \to y \to z$:

```
subs(x^3 + y*z, x = y, y = z)
```

$$z^3 + z^2$$

Example 6

The `subs` function lets you use sequential and parallel substitutions. For example, substitute the operand in the following expressions sequentiallly:

```
subs(a^2 + b^3, a = b, b = a)
```

$$a^3 + a^2$$

```
subs(a^2 + b^3, b = a, a = b)
```

$$b^3 + b^2$$

For the same expression, parallel substitution swaps the identifiers:

```
subs(a^2 + b^3, [a = b, b = a])
```

$$a^3 + b^2$$

In the following call, the substitution of `y + x` for `a` yields the intermediate result `y + 2*x`. From there, the substitution of `z` for `x` results in `y + 2 z`:

```
subs(a + x, a = x + y, x = z)
```

$$y + 2\,z$$

Parallel substitution produces a different result. The following call substitues `a` with `x + y`. Simultaneously, this call substitutes the operand `x` *of the original expression* `a + x` with `z`:

```
subs(a + x, [a = x + y, x = z])
```

$$x + y + z$$

If you specify the substitutions using a set of a table of equations, the `subs` function also performs a parallel substitution:

```
subs(a + x, {a = x + y, x = z})
```

$$x+y+z$$

```
T := table(): T[a] := x + y: T[x] := z: T
```

a	$x+y$
x	z

```
subs(a + x, T)
```

$$x+y+z$$

```
delete T:
```

Example 7

You can combine sequential and parallel substitutions:

```
subs(a + x, {a = x + y, x = z}, x = y)
```

$$2\,y+z$$

Example 8

The `subs` function replaces only those operands that the `op` function can return. The following expression contains the subexpression `x + y` as the operand `op(f, [1, 2])`:

```
f := sin(z*(x + y)): op(f, [1, 2]);
```

$$x+y$$

Consequently, the `subs` function replaces this subexpression:

```
subs(f, x + y = z)
```

$$\sin(z^2)$$

Syntactically, the following sum does not contain the subexpression `x + y`. Therefore, the `subs` function does not replace it:

```
subs(x + y + z, x + y = z)
```

$$x + y + z$$

In contrast to `subs`, the `subsex` function finds and replaces partial sums and products:

```
subsex(x + y + z, x + y = z)
```

$$2\,z$$

```
subs(a*b*c, a*c = 5), subsex(a*b*c, a*c = 5)
```

$$a\,b\,c,\ 5\,b$$

```
delete f:
```

Example 9

You can substitute operands of expression sequences. Enclose sequences in parentheses:

```
subs((a, b, a*b), a = x)
```

$$x,\ b,\ b\,x$$

Example 10

The `Unsimplified` option suppresses simplification:

```
subs(a + b + 2, a = 1, b = 0, Unsimplified)
```

$$1 + 0 + 2$$

Example 11

If you try to substitute something in a domain, MuPAD ignores the substitution. For example, define a new domain with the methods `"foo"` and `"bar"`:

```
mydomain := newDomain("Test"):
mydomain::foo := x -> 4*x:
mydomain::bar := x -> 4*x^2:
```

Now try to replace every number 4 inside the domain with the number 3:

```
mydomain := subs(mydomain, 4 = 3):
```

That substitution does not have any effect:

```
mydomain::foo(x), mydomain::bar(x)
```

$$4\,x,\ 4\,x^2$$

To substitute objects in a domain method, you must substitute in the individual methods:

```
mydomain::foo := subs(mydomain::foo, 4 = 3):
mydomain::bar := subs(mydomain::bar, 4 = 3):
mydomain::foo(x), mydomain::bar(x)
```

$$3\,x,\ 3\,x^2$$

```
delete mydomain:
```

Parameters

`f`

An arbitrary MuPAD object

`old`, $\texttt{old}_1$, $\texttt{old}_2$, …

Arbitrary MuPAD objects

`new`, $\texttt{new}_1$, $\texttt{new}_2$, …

Arbitrary MuPAD objects

s_1, s_2, …

Either equations `old = new`, or lists or sets of such equations, or tables whose entries are interpreted as such equations.

Options

`EvalChanges`

After substitution, evaluate all modified subexpressions.

By default, the `subs` function does not evaluate the modified object. The `EvalChanges` option enforces the evaluation of all modified subexpressions. See “Example 3” on page 1-2014 and “Example 4” on page 1-2015.

`Unsimplified`

Do not simplify the result of a substitution.

As the last step of a substitution, MuPAD automatically simplifies (but does not evaluate) the modified object. The `Unsimplified` option suppresses the final simplification. See “Example 10” on page 1-2018.

Return Values

Copy of the input object with replaced operands.

Overloaded By

f

See Also

MuPAD Functions

`evalAt` | `extnops` | `extop` | `extsubsop` | `has` | `map` | `match` | `op` | `subsex` | `subsop`

subset, _subset, _notsubset

Relation “is a subset of”

Syntax

```
A subset B
```

```
_subset(A, B)
```

```
not A subset B
```

```
_notsubset(A, B)
```

Description

`A subset B` represents the expression $A \subset B$.

A is a subset of B if $x \in A \Rightarrow x \in B$.

The function `_notsubset` exists for typesetting purposes. It is returned as the result of negating a `subset` expression. See “Example 4” on page 1-2022.

If called with symbolic arguments (anything but sets), these functions return a symbolic expression of type _in or the unevaluated input.

Examples

Example 1

When called with two sets, these functions return a Boolean value:

```
{1} subset {1,2,3},
{} subset {1},
{1} subset {1},
{1} subset {}
```

```
TRUE, TRUE, TRUE, FALSE
```

Example 2

Note: identifiers in sets are not assumed to be place-holders. See `?=` for details on syntactic equality.

```
{x} subset {1,2}
```

FALSE

Example 3

If one of the arguments is not a set, these functions return an equivalent symbolic expression:

```
{1} subset A, A subset {1}
```

$1 \in A, A \subset \{1\}$

Example 4

For “pretty typesetting”, the negation of `subset`is implemented in a special function environment:

```
not A subset B
```

$A \not\subset B$

```
type(%)
```

"_notsubset"

Parameters

A, B

MuPAD expressions

Return Values

TRUE, FALSE, or an expression.

Overloaded By

A, B

See Also

MuPAD Functions
`in` | `intersect` | `minus` | `union`

subsex

Extended substitution

Syntax

```
subsex(f, old = new, <Unsimplified>)
subsex(f, old1 = new1, old2 = new2, …, <Unsimplified>)
subsex(f, [old1 = new1, old2 = new2, …], <Unsimplified>)
subsex(f, {old1 = new1, old2 = new2, …}, <Unsimplified>)
subsex(f, table(old1 = new1, old2 = new2, …), <Unsimplified>)
subsex(f, s1, s2, …, <Unsimplified>)
```

Description

`subsex(f, old = new)` returns a copy of the object `f` in which all expressions matching `old` are replaced by the value `new`. In contrast to the function `subs`, `subsex` also replaces "incomplete" subexpressions.

`subsex` returns a modified copy of the object, but does not change the object itself.

`subsex(f, old = new)` searches `f` for subexpressions matching `old`. Each such subexpression is replaced by `new`.

In most cases, `subsex` leads to the same result as `subs`. However, in contrast to `subs`, `subsex` allows to replace "incomplete" subexpressions such as `a + b` in a sum `a + b + c`. In general, combinations of the operands of the n-ary "operators" `+`, `*`, `and`, `_exprseq`, `intersect`, `or`, `_lazy_and`, `_lazy_or`, and `union` can be replaced. In particular, partial sums and partial products can be replaced. Note that these operations are assumed to be commutative, e.g., `subsex(a*b*c, a*c = new)` does replace the partial product `a*c` by `new`. See "Example 1" on page 1-2025 and "Example 2" on page 1-2026.

`subsex` additionally replaces powers with the same base, if the exponent of the expression is an integer multiple of the replacement power, e.g. like in `subsex(a^4,`

`a^2 = new)`. This rule also matches inverted expressions like in `subsex(1/sqrt(x), sqrt(x)=new)`, which is internally equivalent to `subsex(x^(-1/2), x^(1/2)=new)`. Cf. “Example 3” on page 1-2026.

`subsex` is much slower than `subs`! If `subs` can do the substitution, use `subs` rather than `subsex`.

The call `subsex(f, old1 = new1, old2 = new2, ...)` invokes a “sequential substitution”. See the `subs` help page for details.

The call `subsex(f, [old1 = new1, old2 = new2, ...])` invokes a “parallel substitution”. See the `subs` help page for details.

The call `subsex(f, s1, s2, ...)` describes the most general form of substitution which may combine sequential and parallel substitutions. This call is equivalent to `subsex(... subsex(subsex(f, s1), s2), ...)`. Depending on the form of `s1`, `s2`, ..., sequential or parallel substitutions are carried out in each step. An example can be found on the `subs` help page.

After substitution, the result is not evaluated. Use the function `eval` to enforce evaluation. Cf. “Example 5” on page 1-2028.

Operands of expression sequences can be replaced by `subsex`. Such objects are not flattened. Cf. “Example 6” on page 1-2028.

The call `subsex(f)` is allowed; it returns `f` without modifications.

Examples

Example 1

We demonstrate some simple substitutions; `subsex` finds and replaces partial sums and products:

```
subsex(a + b + c, a + c = x)
```

$$b + x$$

```
subsex(a*b*c, a*c = x)
```

$$b\,x$$

```
subsex(a * (b + c) + b + c, b + c = a)
```

$$a^2 + a$$

```
subsex(a + b*c*d + b*d, b*d = c);
```

$$c^2 + c + a$$

Example 2

We replace subexpressions inside an expression sequence and a symbolic union of sets:

```
subsex((a, b, c, d), (b, d) = w)
```

$$a, c, w$$

```
subsex(a union b union c, a union b = w)
```

$$c \cup w$$

The same can be achieved by using the functional equivalent `union` of the operator `union`:

```
subsex(_union(a, b, c), _union(a, b) = w)
```

$$c \cup w$$

Example 3

`subsex` replaces powers with with the same base, if the exponent of the expression is an integer multiple of the replacement power:

```
subsex(1/a^4, a^2 = X)
```

$$\frac{1}{X^2}$$

This holds even for exponents which are expressions:

```
subsex(1/a^(6*x), a^(2*x) = X)
```

$$\frac{1}{X^3}$$

`1/sqrt(x)` is internally `x^(-1/2)`, so the replacement of `sqrt(x)` which is internally `x^(1/2)` works, too:

```
subsex(1/sqrt(n), sqrt(n) = X)
```

$$\frac{1}{X}$$

Example 4

`subsex` (and also `subs`) are often useful to convert the output of one command into a form required by the next one. As an example, we compute the Laplace transform of the two-dimensional ODE

$$x'(t) = x(t) + 2\,y(t),\ x(0) = 1$$

,

$$y'(t) = 5\,x(t) + 2\,y(t),\ y(0) = -2$$

,

and transform the result into a form suitable for calling `solve` by replacing the unknown Laplace transforms by symbolic names:

```
xfrm1 := laplace(x'(t) = x(t) + 2*y(t), t, s);
xfrm2 := laplace(y'(t) = 5*x(t) + 2*y(t), t, s)
```

$$s\,\mathrm{laplace}(x(t), t, s) - x(0) = \mathrm{laplace}(x(t), t, s) + 2\,\mathrm{laplace}(y(t), t, s)$$

$$s\,\mathrm{laplace}(y(t), t, s) - y(0) = 5\,\mathrm{laplace}(x(t), t, s) + 2\,\mathrm{laplace}(y(t), t, s)$$

For readability, we give names to both substitutions:

```
sub_x := laplace(x(t),t,s) = X:
sub_y := laplace(y(t),t,s) = Y:
```

```
Leqn1 := subs(xfrm1, sub_x, sub_y, x(0) = 1);
Leqn2 := subs(xfrm2, sub_x, sub_y, y(0) =-2)
```

$$X\,s - 1 = X + 2\,Y$$

$$Y\,s + 2 = 5\,X + 2\,Y$$

```
solve({Leqn1, Leqn2}, {X, Y})
```

$$\begin{cases} \varnothing & \text{if } s^2 = 3\,s + 8 \\ \left\{\left[X = -\frac{s-6}{-s^2+3\,s+8},\ Y = \frac{2\,s-7}{-s^2+3\,s+8}\right]\right\} & \text{if } s^2 \neq 3\,s + 8 \end{cases}$$

(This example was suggested by MuPAD user Brad Cooper.)

Example 5

The result of `subsex` is not evaluated. In the following call, the identifier `sin` is not replaced by its value, i.e., by the procedure defining the behavior of the system's sine function. Consequently, `sin(2*PI)` is not simplified to 0 by this procedure:

```
subsex(sin(2*x*y), x*y = PI)
```

$$\sin(2\,\pi)$$

The function `eval` enforces evaluation:

```
eval(subsex(sin(2*x*y), x*y = PI))
```

$$0$$

Example 6

Operands of expression sequences can be subtituted. Note that sequences need to be enclosed in brackets:

```
subsex((a, b, a*b*c), a*b = x)
```

$$a, b, c\,x$$

Example 7

The option `Unsimplified` suppresses simplification:

```
subsex(2 + a + b, a + b = 0, Unsimplified)
```

$$2 + 0$$

Parameters

`f`

An arbitrary MuPAD object

`old`, $\texttt{old}_1$, $\texttt{old}_2$, …

Arbitrary MuPAD objects

`new`, $\texttt{new}_1$, $\texttt{new}_2$, …

Arbitrary MuPAD objects

$\texttt{s}_1$, $\texttt{s}_2$, …

Either equations `old = new`, or lists or sets of such equations, or tables whose entries are interpreted as such equations.

Options

`Unsimplified`

Prevents simplification of the returned object after substitution

As the last step of a substitution, the modified object is simplified (however, not evaluated). This option suppresses this final simplification. An example can be found on the `subs` help page.

Return Values

Copy of the input object with replaced operands.

Overloaded By

f

See Also

MuPAD Functions

`evalAt` | `extnops` | `extop` | `extsubsop` | `has` | `map` | `match` | `op` | `subs` | `subsop`

subsop

Replace operands

Syntax

```
subsop(object, i1 = new1, i2 = new2, …, <Unsimplified>)
```

Description

`subsop` returns a modified copy of the object, but does not change the object itself.

`subsop(object, i = new)` replaces the operand `op(object, i)` by `new`. Operands are specified in the same way as with the function `op`: `i` may be an integer or a list of integers. E.g., `subsop(object, [j, k] = new)` replaces the suboperand `op(op(object, j), k)`. Cf. “Example 2” on page 1-2032. In contrast to `op`, ranges cannot be used in `subsop` to specify more than one operand to replace. Several substitution equations have to be specified instead.

If several operands are to be replaced, the specified substitutions are processed in sequence from left to right. Each substitution is carried out and the result is processed further with the next substitution. The intermediate objects are not simplified.

The result of `subsop` is not evaluated further. It can be evaluated via the function `eval`. Cf. “Example 3” on page 1-2033.

Operands of expression sequences can be replaced by `subsop`. Such objects are not flattened.

Note that the order of the operands may change by replacing operands and evaluating the result. Cf. “Example 4” on page 1-2033.

`FAIL` is returned if an operand cannot be accessed.

Substitution via `subsop` is faster than via `subs` or `subsex`.

The call `subsop(object)` is allowed; it returns the object without modifications.

Examples

Example 1

We demonstrate how to replace one or more operands of an expression:

```
x := a + b: subsop(x, 2 = c)
```

$$a + c$$

```
subsop(x, 1 = 2, 2 = c)
```

$$c + 2$$

Also the 0-th operand of an expression (the “operator”) can be replaced:

```
subsop(x, 0 = _mult)
```

$$a\,b$$

The variable `x` itself was not affected by the substitutions:

```
x
```

$$a + b$$

```
delete x:
```

Example 2

The following call specifies the suboperand `c` by a list of integers:

```
subsop([a, b, f(c)], [3, 1] = x)
```

$$[a, b, f(x)]$$

Example 3

This example demonstrates the effect of simplification. The following substitution replaces the first operand `a` by 2. The result simplifies to 3:

```
subsop(a + 1, 1 = 2)
```

$$3$$

The option `Unsimplified` suppresses the simplification:

```
subsop(a + 1, 1 = 2, Unsimplified)
```

$$2 + 1$$

The next call demonstrates the difference between *simplification* and *evaluation*. After substitution of `PI` for `x`, the identifier `sin` is not evaluated, i.e., the body of the system function `sin` is not executed:

```
subsop(sin(x),  1 = PI)
```

$$\sin(\pi)$$

Evaluation of `sin` simplifies the result:

```
eval(%)
```

$$0$$

Example 4

The order of operands may change by substitutions. Substituting `z` for the identifier `b` changes the internal order of the terms in `x`:

```
x := a + b + c: op(x)
```

$$a,\ b,\ c$$

```
x := subsop(x, 2 = z): op(x)
```

$$a, c, z$$

```
delete x:
```

Parameters

`object`

Any MuPAD object

i_1, i_2, …

Integers or lists of integers

new_1, new_2, …

Arbitrary MuPAD objects

Options

`Unsimplified`

As the last step of a substitution, the modified object is simplified (however, not evaluated). This option suppresses this final simplification. Cf. “Example 3” on page 1-2033.

Return Values

Input object with replaced operands or `FAIL`.

Overloaded By

object

Algorithms

For overloading `subsop`, it is sufficient to handle the cases `subsop(object)` and `subsop(object, i = new)`.

The case where the position of the operand to be replaced is given by a list is always handled recursively: First, `op` is called with the list bar the last element to find the object to substitute in (using the overloading of `op` if present, storing all the intermediate results), then the substitution is performed on that sub-object (using the overloading of `subsop` of the form `subsop(subobj, i = new)`). The result is substituted into the last-but-one result of the recursive `op` call, again respecting any overloading of `subsop`, and so on up to the front of the list.

See Also

MuPAD Functions

`extnops` | `extop` | `extsubsop` | `map` | `match` | `op` | `subs` | `subsex`

substring

Extract a substring from a string

Syntax

```
substring(string, i)

substring(string, i, l)

substring(string, i .. j)
```

Description

`substring(string, i)` returns the *i*-th character of a string.

`substring(string, i, l)` returns the substring of length `l` starting with the *i*-th character of the string.

`substring(string, i..j)` returns the substring consisting of the characters *i* through *j*, inclusive.

The empty string `""` is returned if the length `l = 0` is specified.

`substring` is considered obsolete. You should use index access to strings instead.

Examples

Example 1

We extract individual characters from a string:

```
substring("123456789", i) $ i = 1..9
```

```
"1", "2", "3", "4", "5", "6", "7", "8", "9"
```

Substrings of various lengths are extracted:

```
substring("123456789", 1, 2), substring("123456789", 4, 4)
```

```
"12", "4567"
```

Substrings of length 0 are empty strings:

```
substring("123456789", 4, 0)
```

```
""
```

Ranges may be used to specify the substrings:

```
substring("123456789", 1..9)
```

```
"123456789"
```

Example 2

The following `while` loop removes all trailing blank characters from a string:

```
string := "MuPAD       ":
while substring(string, length(string)) = " " do
  string := substring(string, 1..length(string) - 1)
end_while
```

```
"MuPAD"
```

Parameters

`string`

A nonempty character string

`i`

An integer between 1 and `length(string)`

`l`

An integer between `1` and `length(string)`

`j`

An integer between `i` and `length(string)`

Return Values

Character string

See Also

MuPAD Functions
`length` | `stringlib::subs` | `strmatch`

_subtract

Subtract expressions

Syntax

```
_subtract(x, y)
```

Description

`_subtract(x, y)` subtracts `y` from `x`.

The difference operator `-` does not call `_subtract(x, y)`. The difference `x - y` is equivalent to `x + (-y) = _plus (x, _negate(y))`.

To implement the slot `d::_subtract` for your domain `d`, use the following convention:

- If both `x` and `y` are elements of `d`, the slot must return an appropriate difference of type `d`.
- If `x` or `y` is not an element of `d` and cannot be converted to an element of `d`, the slot must return `FAIL`.
- If `x` or `y` is not an element of `d`, but can be converted to type `d`, use the following approach. This object must be converted to an element of `d` only if the mathematical semantics is obvious to all users of `d`, including the users who treat this domain as a "black box". For example, you can regard integers as rational numbers because of the natural mathematical embedding, but you must make sure that all users are aware of this approach. Otherwise, the `"_subtract"` method must return `FAIL` instead of using implicit conversions. If you use implicit conversions for the elements of your domain, document these conversions.

In the MuPAD standard installation, most of the library domains comply with this convention.

`_subtract` can subtract polynomials of the `DOM_POLY` type from a polynomial of the same type. The polynomials must have the same indeterminates and the same coefficient ring.

`_subtract` can subtract finite sets from a finite set. For finite sets `X` and `Y`, the difference is the set $\{x - y \mid x \in X, y \in Y\}$.

Examples

Example 1

Compute the difference of the following arithmetical expressions by using the `_subtract` method. Then, compute the difference of the same expressions by using the difference operator:

```
_subtract(x, y), x - y
```

$$x - y, x - y$$

Although both `_subtract` and the difference operator return the same result for these expressions, the `_subtract` call is not equivalent to `x - y`:

```
type(hold(x - y)), type(hold(_subtract(x, y)))
```

"_plus", "_subtract"

Example 2

Use the `_subtract` function when combining the following lists:

```
zip([a, b, c, d], [1, 2, 3, 4], _subtract)
```

$$[a - 1, b - 2, c - 3, d - 4]$$

Parameters

x, y

arithmetical expressions, polynomials of type `DOM_POLY`, or sets

Return Values

arithmetical expression, a polynomial, or a set.

Overloaded By

x, y

See Also

MuPAD Functions

`*` | `+` | `-` | `/` | `^` | `_invert` | `poly` | `Pref::keepOrder`

sum

Definite and indefinite summation

Syntax

```
sum(f, i)

sum(f, i = a .. b)

sum(f, i = RootOf(p, x))
```

Description

`sum(f, i)` computes a symbolic antidifference of $f(i)$ with respect to i.

`sum(f, i = a..b)` tries to find a closed form representation of the sum $\sum_{i=a}^{b} f(i)$.

`sum` serves for simplifying *symbolic* sums (the discrete analog of integration). It should *not* be used for simply adding a finite number of terms: if `a` and `b` are integers of type `DOM_INT`, the call `_plus(f $ i = a..b)` gives the desired result, while `sum(f, i = a..b)` may return unevaluated. `expand` may be used to sum such an unevaluated finite sum. See "Example 3" on page 1-2044.

`sum(f, i)` computes the indefinite sum of `f` with respect to `i`. This is an expression `g` such that $f(i) = g(i + 1) - g(i)$.

It is implicitly assumed that `i` runs through integers only.

`sum(f, i = a..b)` computes the definite sum with `i` running from `a` to `b`.

If `a` and `b` are numbers, then they must be integers.

If `b - a` is a nonnegative integer, then the explicit sum $f(a) + f(a + 1) + \ldots + f(b)$ is returned, provided that this sum has no more than 1000 terms.

`sum(f, i = RootOf(p, x))` computes the sum with `i` extending over all roots of the polynomial `p` with respect to `x`.

If `f` is a rational function of `i`, a closed form of the sum will be found.

See “Example 2” on page 1-2044.

The system returns a symbolic call of `sum` if it cannot compute a closed form representation of the sum.

Infinite symbolic sums without symbolic parameters can be evaluated numerically via `float` or `numeric::sum`. Cf. “Example 4” on page 1-2046.

Examples

Example 1

We compute some indefinite sums:

```
sum(1/(i^2 - 1), i)
```

$$-\frac{1}{2\,(i-1)}-\frac{1}{2\,i}$$

```
sum(1/i/(i + 2)^2, i)
```

$$\frac{\psi'(i+2)}{2}-\frac{1}{4\,i+4}-\frac{1}{4\,i}$$

```
sum(binomial(n + i, i), i)
```

$$\begin{cases} 0 & \text{if } n=-1 \\ \dfrac{i\binom{i+n}{i}}{n+1} & \text{if } n\neq -1 \end{cases}$$

We compute some definite sums. Note that $\pm\infty$ are valid boundaries:

```
sum(1/(i^2 + 21*i), i = 1..infinity)
```

$$\frac{18858053}{108636528}$$

```
sum(1/i, i = a .. a + 3)
```

$$\psi(a+4) - \psi(a)$$

```
expand(%)
```

$$\frac{1}{a+1} + \frac{1}{a+2} + \frac{1}{a+3} + \frac{1}{a}$$

Example 2

We compute some sums over all roots of a polynomial:

```
sum(i^2, i = RootOf(x^3 + a*x^2 + b*x + c, x))
```

$$a^2 - 2\,b$$

```
sum(1/(z + i), i = RootOf(x^4 - y*x + 1, x))
```

$$\frac{4\,z^3 + y}{z^4 + y\,z + 1}$$

Example 3

`sum` can compute finite sums if indefinite summation succeeds:

```
sum(1/(i^2 + i), i = 1..100)
```

$$\frac{100}{101}$$

`_plus` yields the same result more quickly if the number of summands is small:

```
_plus(1/(i^2 + i) $ i = 1..100)
```

$$\frac{100}{101}$$

In such cases, `sum` is much more efficient than `_plus` if the number of summands is large:

```
sum(1/(i^2 + i), i = 1..10^30)
```

$$\frac{1000000000000000000000000000000}{1000000000000000000000000000001}$$

Finite sums for which no indefinite summation is possible are expanded if they have no more than 1000 terms:

```
sum(binomial(n, i), i = 0..4)
```

$$n + \binom{n}{2} + \binom{n}{3} + \binom{n}{4} + 1$$

An application of `expand` is necessary to expand the binomials:

```
expand(%)
```

$$\frac{n^4}{24} - \frac{n^3}{12} + \frac{11\,n^2}{24} + \frac{7\,n}{12} + 1$$

Finite sums with more than 1000 terms are not expanded:

```
sum(binomial(n, i), i = 0..1000)
```

$$\sum_{i=0}^{1000} \binom{n}{i}$$

You might use `expand` here to expand the sum and obtain a huge expression. If you really want to do that, we recommend using `_plus` directly.

However, if one of the boundaries is symbolic, then `_plus` cannot be used:

```
_plus(1/(i^2 + i) $ i = 1..n)
```

$$\_plus\left(\frac{1}{i^2+i}\ \$\ i = 1..n\right)$$

```
_plus(binomial(n, i) $ i = 0..n)
```

$$\_plus\left(\binom{n}{i}\ \$\ i = 0..n\right)$$

```
sum(1/(i^2 + i), i = 1..n), sum(binomial(n, i), i = 0..n)
```

$$\frac{n}{n+1},\ 2^n$$

Example 4

The following infinite sum cannot be computed symbolically:

```
sum(ln(i)/i^5, i = 1..infinity)
```

$$\sum_{i=1}^{\infty} \frac{\ln(i)}{i^5}$$

We obtain a floating-point approximation via `float`:

```
float(%)
```

$$0.02857378051$$

Alternatively, the function `numeric::sum` can be used directly. This is usually much faster than applying `float`, since it avoids the overhead of `sum` attempting to compute a symbolic representation:

```
numeric::sum(ln(i)/i^5, i = 1..infinity)
```

$$0.02857378051$$

Parameters

f

An arithmetical expression depending on `i`

i

The summation index: an identifier or indexed identifier

a, b

The boundaries: arithmetical expressions

p

A polynomial of type `DOM_POLY` or a polynomial expression

x

An indeterminate of `p`

Return Values

arithmetical expression.

Algorithms

The function `sum` implements Abramov's algorithm for rational expressions, Gosper's algorithm for hypergeometric expressions, and Zeilberger's algorithm for the definite summation of holonomic expressions.

See Also

MuPAD Functions

`+` | `_plus` | `int` | `numeric::sum` | `product` | `rec` | `sum::addpattern`

sum::addpattern

Add patterns for definite and indefinite summation

Syntax

```
sum::addpattern(pat, k, res, <[var, …], <[cond, …]>>)
```

```
sum::addpattern(pat, k = a .. b, res, <[var, …], <[cond, …]>>)
```

Description

`sum::addpattern(pat, k, res)` teaches `sum` to make use of $\sum_k \text{pat} = \text{res}$.

`sum::addpattern(pat, k=a..b, res)` teaches `sum` that $\sum_{x=a}^{b} \text{pat} = \text{res}$.

A part of a computer algebra system's summation abilities stems from mathematical pattern matching. The MuPAD pattern matcher can be extended at runtime with `sum::addpattern`.

For definite summation, each bound is either an arithmetical expression which may contain pattern variables, or an identifier which can be used as a variable in the result and condition terms.

Users can include pattern variables and conditions on these by giving additional arguments. These conditions, as well as the result, are protected from premature evaluation, i.e., it is not necessary to write `hold( _not @ iszero )(a^2-b)`, a simple `not iszero(a^2-b)` suffices.

The difference between `not iszero(a^2-b)` and `a^2-b <> 0` when given as a condition is that the latter takes into account assumptions on the identifiers encountered, while the first does not. Cf. “Example 4” on page 1-2050.

Patterns introduced by `sum::addpattern` are also used in recursive calls of `sum` and are automatically extended to include simple applications of summation by change of variables. Cf. “Example 1” on page 1-2049.

Environment Interactions

Calling `sum::addpattern` changes the expressions returned by future calls to `sum`.

Examples

Example 1

Not surprisingly, MuPAD does not know how to do an indefinite summation with the function *foo*:

```
sum(foo(n), n)
```

$$\sum_n \text{foo}(n)$$

We add a pattern for this function:

```
sum::addpattern(foo(k), k, bar(k))
```

```
sum(foo(n), n)
```

$$\text{bar}(n)$$

Note that this pattern is also used indirectly:

```
sum(foo(k+3),k)
```

$$\text{bar}(k+3)$$

Example 2

Definite sums can be added similarly:

```
sum::addpattern(foo(k), k=1..infinity, bar(k))
```

```
sum(foo(k), k=1..infinity)
```

$$\text{bar}(k)$$

The above pattern will also match this definite sum with different bounds:

```
sum(foo(k), k=3..infinity)
```

$$\mathrm{bar}(k) - \mathrm{foo}(2) - \mathrm{foo}(1)$$

Note that this pattern is also used indirectly:

```
sum(foo(k)+1/k^3, k=1..infinity)
```

$$\zeta(3) + \mathrm{bar}(k)$$

The bounds may also be variables occurring in the pattern or the result:

```
sum::addpattern(foo(k,a), k=0..a, bar(a), [a])
```

```
sum(foo(k,7), k=0..7)
```

$$\mathrm{bar}(7)$$

Example 3

The name of the summation variable used in the call to `sum::addpattern` does not restrict later calls to `sum`:

```
sum::addpattern(x^(2*i+1)/(2*i+1), i=0..infinity,
                piecewise([abs(x) < 1,
                           arccoth(x) + PI/2*sqrt(-1/x^2)*x]),
                [x])
```

```
sum(x^(2*n+1)/(2*n+1),n=0..infinity)
```

$$\begin{cases} \mathrm{arccoth}(x) + \dfrac{\pi x \sqrt{-\frac{1}{x^2}}}{2} & \text{if } |x| < 1 \end{cases}$$

Example 4

Conditions are checked using `is` and therefore react to assumptions:

```
sum::addpattern(binomial(-1/2, k)*x^(2*k^2 + 1)/(2*k + 1),
                                k = 0..infinity, arcsinh(x),
                                       [x], [abs(x) < 1])
```

```
sum(binomial(-1/2, k)*x^(2*k^2 + 1)/(2*k + 1),
    k = 0..infinity) assuming -1 < x < 1
```

$$\operatorname{arcsinh}(x)$$

```
sum(binomial(-1/2, k)*x^(2*k^2 + 1)/(2*k + 1),
    k = 0..infinity) assuming x > 1
```

$$\sum_{k=0}^{\infty} \frac{x^{2k^2+1}\binom{-\frac{1}{2}}{k}}{2k+1}$$

If MuPAD cannot decide whether the conditions are satisfied, a piecewise defined object is returned:

```
sum(binomial(-1/2, k) * x^(2*k^2+1)/(2*k+1),
    k = 0..infinity)
```

$$\begin{cases} \operatorname{arcsinh}(x) & \text{if } |x| < 1 \end{cases}$$

If either the conditions are not satisfied or substituting the values into the result yields an error, the pattern is ignored. There is no need to include a condition to guard against an error, MuPAD simply computes the sum as usual:

```
sum::addpattern(c^k, k=0..n, (c^n-1)/(c-1), [c]);
sum(1^k, k=0..n)
```

$$n+1$$

Parameters

pat

The pattern to match: an arithmetical expression in `k`.

k

The summation index: an identifier.

a .. b

The boundaries for a definite summation: arithmetical expressions or identifiers.

res

The pattern for the result of the summation: an arithmetical expression

var, …

"pattern variables": placeholders in `pat` and `ret`, i.e., identifiers or indexed identifiers. They do not represent themselves but almost arbitrary MuPAD expressions not containing `k`. You may restrict them by the conditions in the 5th parameter.

cond, …

Conditions on the pattern variables

Return Values

Object of type `DOM_NULL`

See Also

MuPAD Functions

`sum`

surd

N -th root

Syntax

```
surd(x, n)
```

Description

For a complex number `x` and integer `n`, `surd(x, n)` returns the n-th root of x whose (complex) argument is closest to that of `x`.

If `x` is a positive real number, `surd(x, n)` coincides with `x^(1/n)`. If `x` is a negative real number and `n` is odd, then `surd(x, n)` coincides with `-|x|^(1/n)`.

`surd(x, n)` returns that complex solution y of $y^n = x$ with polar angle closest to that of x; among two equally distant y's, the one with smaller argument is chosen. In contrast, `x^(1/n)` represents the solution with the smallest absolute value of the polar angle in the range $(-\pi, \pi]$.

If `n` is a numerical value, it must be a non-zero integer. If it is symbolic, it is understood to represent a non-zero integer.

`surd(x, 2)` is mathematically equivalent to `sqrt(x)`. Unlike `sqrt`, however, `surd` may return an unevaluated symbolic call.

Environment Interactions

When called with a floating-point argument, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

If n is odd and x is real, then `surd(x, n)` is real, too. On the other hand, `x^(1/n)` is not real if x is negative:

```
surd(-27, 3), surd(-27.0, 3), (-27)^(1/3), (-27.0)^(1/3)
```

$$-3,\ -3.0,\ 3\,(-1)^{1/3},\ 1.5 + 2.598076211\,i$$

Example 2

`surd` may be called with symbolic arguments:

```
surd(3, n)
```

$$3^{1/n}$$

Sometimes, `surd` returns an unevaluated function call:

```
surd(x, 3), surd(x, n^2 + n)
```

$$\sqrt[3]{x},\ \sqrt[n^2+n]{x}$$

Parameters

x

An arithmetical expression

n

An arithmetical expression

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`_power` | `sqrt`

sysname

Name of the operating system

Syntax

```
sysname(<Arch>)
```

Description

`sysname()` returns information on the operating system on which MuPAD is currently executed. It can return one of the following strings:

- `"UNIX"` for UNIX operating systems including Mac OS X and Linux,
- `"MSDOS"` for MS-DOS® operating systems including Microsoft Windows,

`sysname(Arch)` returns a more specific name of the operating system as a character string.

Examples

Example 1

On a 64-bit Microsoft Windows operating system, `sysname` returns the following values:

```
sysname(), sysname(Arch)
```

```
"MSDOS", "win64"
```

On a 32-bit Microsoft Windows operating system, `sysname(Arch)` returns:

```
sysname(Arch)
```

```
"win32"
```

Example 2

On a 64-bit Linux operating system, `sysname` returns the following values:

```
sysname(), sysname(Arch)
```

```
"UNIX", "glnxa64"
```

Example 3

On a 64-bit Apple Macintosh operating system, `sysname` returns the following values:

```
sysname(), sysname(Arch)
```

```
"UNIX", "maci64"
```

Options

`Arch`

Makes `sysname` return more specific information on the architecture

Return Values

character string.

See Also

MuPAD Functions

`system`

sysorder

Compare objects according to the internal order

Syntax

```
sysorder(object1, object2)
```

Description

`sysorder(object1, object2)` returns `TRUE` if the MuPAD internal order of `object1` is less than or equal to the order of `object2`. Otherwise, `FALSE` is returned.

Note: The exceptions are domains.

One should not try and use the internal order to sort objects according to specific criteria. E.g., its does not necessarily reflect the natural ordering of numbers or strings. Further, the internal order may differ between different MuPAD versions.

The only feature one may rely upon is its uniqueness. Cf. “Example 2” on page 1-2059.

Examples

Example 1

We give some examples how `sysorder` behaves in the current MuPAD version. For numbers, the internal order is equal to the natural order:

```
sysorder(3, 4) = bool(3 <= 4),
sysorder(45, 33) = bool(45 <= 33),
sysorder(0, 4) = bool(0 <= 4)
```

$$\text{TRUE} = \text{TRUE},\ \text{FALSE} = \text{FALSE},\ \text{TRUE} = \text{TRUE}$$

```
sysorder(1/3, 1/4) = bool(1/3 <= 1/4),
sysorder(-4, 2) = bool(-4 <= 2),
sysorder(-4, -2) =  bool(-4 <= -2)
```

$$\text{FALSE} = \text{FALSE},\ \text{TRUE} = \text{TRUE},\ \text{TRUE} = \text{TRUE}$$

Example 2

We give a simple application of `sysorder`. Suppose, we want to implement a function `f`, say, whose only known property is its skewness `f(-x) = -f(x)`. Expressions involving `f` should be simplified automatically, e.g., `f(x) + f(-x)` should yield zero for any argument `x`. To achieve this, we use `sysorder` to decide, whether a call `f(x)` should return `f(x)` or `-f(-x)`:

```
f := proc(x) begin
       if sysorder(x, -x) then
            return(-procname(-x))
       else return(procname(x))
       end_if;
     end_proc:
```

For numerical arguments, `f` prefers to rewrite itself with positive arguments:

```
f(-3), f(3), f(-4.5), f(4.5), f(-2/3), f(2/3)
```

$$-f(3),\ f(3),\ -f(4.5),\ f(4.5),\ -f\left(\frac{2}{3}\right),\ f\left(\frac{2}{3}\right)$$

For other arguments, the result is difficult to predict:

```
f(x), f(-x), f(sqrt(2) + 1), f(-sqrt(2) - 1)
```

$$-f(-x),\ f(-x),\ f(\sqrt{2}+1),\ -f(\sqrt{2}+1)$$

With this implementation, expressions involving `f` simplify automatically:

```
f(x) + f(-x) - f(3)*f(x) + f(-3)*f(-x) + sin(f(7)) + sin(f(-7))
```

$$0$$

```
delete f:
```

Parameters

object1, object2

Arbitrary MuPAD objects

Return Values

TRUE or FALSE.

See Also

MuPAD Functions

`_less` | `listlib::removeDupSorted` | `sort`

system

Execute a command of the operating system

Syntax

```
! command
```

```
system(command)
```

Description

`system("command")` executes a command of the operating system or a program, respectively.

`!command` is equivalent to `system("command"):`; note that `!command` will suppress output of its return value.

The syntax `!command` is allowed during interactive input only, not when reading MuPAD input from a file. "`!`" must be the first character on the input line.

`system` is not available in all MuPAD versions. If not available, a call to `system` results in the following error message:

```
Error: Function not available for this client [system].
```

`system("command")` sends the command to the operating system. E.g., this command may start another application program on the computer. The return value 0 indicates that the command was executed successfully. Otherwise, an integer error code is returned which depends on the operating system and the command.

If the called command writes output to `stderr` on UNIX systems, the output will go to the MuPAD `stderr`. Outputs on the standard output channel will be inserted in the command's output, but are not accessible programmatically.

Examples

Example 1

On a UNIX system, the `date` command is executed. The command output is printed to the screen, the error code 0 for successful execution is returned to the MuPAD session:

```
errorcode := system("date"):
```

```
Fri Sep 29 14:42:13 MEST 2000
```

```
errorcode
```

```
    0
```

Now the `date` command is called with the command line option `'+%m'` in order to display the current month only:

```
errorcode := system("date '+%m'"):
```

```
09
```

Missing the prefix `'+'` in the command line option of `date`, `date` and therefore `system` returns an error code. Note that the error output goes to `stderr`:

```
system("date '%m'")
```

```
    1
```

```
delete errorcode:
```

Example 2

The output of a program started with the `system` command cannot be accessed in MuPAD directly, but it can be redirected into a file and then be read using the `read` or `ftextinput` command:

```
system("echo communication example > comm_file"):
ftextinput("comm_file")
```

```
"communication example"
```

```
system("rm -f comm_file"):
```

Parameters

`command`

A command of the operating system or a name of a program as a MuPAD character string

Return Values

“error code”: an integer.

See Also

MuPAD Functions

`sysname`

table

Create a table

Syntax

```
table()
```

```
table(index1 = entry1, index2 = entry2, …, <default>)
```

```
table(<list>, <set>, <tab>, …, <default>)
```

Description

`table()` creates a new empty table.

`table(index1 = entry1, index2 = entry2, ...)` creates a new table with the given indices and entries.

In MuPAD, tables are the most flexible objects for storing data. In contrast to arrays or lists, arbitrary MuPAD objects can be used as indices. Indexed access to table entries is fast and nearly independent of the size of the table. Thus, tables are suitable containers for large data.

For a table`T`, say, an indexed call `T[index]` returns the corresponding entry. If no such entry exists, the default value of the table is returned, if the table has one. If no default value has been set and, the indexed expression `T[index]` is returned symbolically.

An indexed assignment of the form `T[index] := entry` adds a new entry to an existing table `T` or overwrites an existing entry associated with the index.

`table` can be used to create tables from other tables, lists or sets of equations. Cf. “Example 2” on page 1-2066.

`table` is used for the explicit creation of a table. There also is the following mechanism for creating a table implicitly.

If the value of an identifier `T`, say, is neither a table nor an array nor an hfarray nor a list, then an indexed assignment `T[index] := entry` is equivalent to `T :=`

`table(index = entry)`. I.e., implicitly, a new table with one entry is created. Cf. “Example 3” on page 1-2067.

If the value of `T` was either a table or an array or an hfarray or a list, then the indexed assignment only inserts a new entry without changing the type of `T` implicitly.

Table entries can be deleted with the function `delete`. Cf. “Example 4” on page 1-2067.

Examples

Example 1

The following call creates a table with two entries:

```
T := table(a = 13, c = 42)
```

a	13
c	42

The data may be accessed via indexed calls. Note the symbolic result for the index `b` which does not have a corresponding entry in the table:

```
T[a], T[b], T[c]
```

$$13,\ T_b,\ 42$$

Entries of a table may be changed via indexed assignments:

```
T[a] := T[a] + 10: T
```

a	23
c	42

Expression sequences may be used as indices or entries, respectively. Note, however, that they have to be enclosed in brackets when using them as input parameters for `table`:

```
T := table((a, b) = "hello", a + b = (50, 70))
```

$a + b$	50, 70
a, b	"hello"

```
T[a + b]
```

50, 70

Indexed access does not require additional brackets:

```
T[a, b] := T[a, b]." world": T
```

$a + b$	50, 70
a, b	"hello world"

```
delete T:
```

Example 2

A `table` can be created from other tables, lists or sets:

```
table(table(a = 1, b = 2),
      {a = 3, c = 4},
      [b = 5, e = 6])
```

a	3
b	5
c	4
e	6

Please note that a set has no order of operands. When a set contains several values under the same index, the table entry is chosen “randomly”:

```
table({a = 3, a = 4});
table({a = 4, a = 3})
```

a	4

Example 3

Below, a new table is created implicitly by an indexed assignment using an identifier `T` without a value:

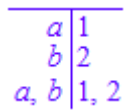
```
delete T: T[4] := 7: T
```


```
delete T:
```

Example 4

Use `delete` to delete entries:

```
T := table(a = 1, b = 2, (a, b) = (1, 2))
```




```
delete T[b], T[a, b]: T
```


```
delete T:
```

Example 5

One of the uses of tables is to count the number of occurrences of some objects. In this situation, an implementation not using default values would have to look like this:

```
T := table():
L := [1,2,3,a,b,c,a,b,a]:
for i in L do
```

```
  if contains(T, i) then
    T[i] := T[i] + 1;
  else
    T[i] := 1;
  end_if;
end_for:
T
```

1	1
2	1
3	1
a	3
b	2
c	1

Note the test whether `T[i]` has already been set. If it has not, we cannot use its previous value, because that would remain symbolic:

```
T := table():
T[a] := T[a] + 1:
T
```

a	$T_a + 1$

By creating `T` as `table(0)` instead of `table()`, we can tell MuPAD to regard `T[i]` as `0` if it has not been told anything else and the code from above becomes substantially shorter and, much more important, much easier to read:

```
T := table(0):
L := [1,2,3,a,b,c,a,b,a]:
for i in L do
  T[i] := T[i] + 1;
end_for:
T
```

1	1
2	1
3	1
a	3
b	2
c	1

A slightly more complicated version counting all identifiers in an expression:

```
ex := sin(a*x+b)-cos(c+x):
cnt := table(0):
misc::maprec(ex,
  {DOM_IDENT} = (x -> (cnt[x] := cnt[x]+1; x))):
cnt
```

_mult	2
_plus	3
a	1
b	1
c	1
cos	1
sin	1
x	2

Parameters

$\mathtt{index}_1$, $\mathtt{index}_2$, …

The indices: arbitrary MuPAD objects

$\mathtt{entry}_1$, $\mathtt{entry}_2$, …

The corresponding entries: arbitrary MuPAD objects

`list`

A list of equations

`set`

A set of equations

`tab`

A table

`default`

The default value: A MuPAD object which is not an equation, a list, a set, nor a table

Return Values

Object of type DOM_TABLE.

See Also

MuPAD Domains
DOM_ARRAY | DOM_HFARRAY | DOM_LIST | DOM_TABLE

MuPAD Functions
_assign | _index | array | assignElements | delete | hfarray | indexval

taylor

Compute a Taylor series expansion

Syntax

```
taylor(f, x, <order>, <mode>)

taylor(f, x = x0, <order>, <mode>)

taylor(f, x, <AbsoluteOrder = order>)

taylor(f, x = x0, <AbsoluteOrder = order>)

taylor(f, x, <RelativeOrder = order>)

taylor(f, x = x0, <RelativeOrder = order>)
```

Description

`taylor(f, x = x0)` computes the first terms of the Taylor series of `f` with respect to the variable `x` around the point `x0`.

If `taylor` finds the corresponding Taylor series, the result is a series expansion of domain type `Series::Puiseux`. Use `expr` to convert it to an arithmetical expression of domain type `DOM_EXPR`. See “Example 1” on page 1-2072.

If a Taylor series does not exist or if `taylor` cannot find it, then `taylor` throws an error. See “Example 2” on page 1-2073 and “Example 3” on page 1-2074.

Mathematically, the expansion computed by `taylor` is valid in some open disc around the expansion point in the complex plane.

If `x0` is `complexInfinity`, then an expansion around the complex infinity, i.e., the north pole of the Riemann sphere, is computed. If `x0` is `infinity` or `-infinity`, a directed series expansion valid along the real axis is computed.

Such an expansion is computed as follows: The series variable `x` in `f` is replaced by $x = \pm\frac{1}{u}$. Then a directed series expansion at $u = 0$ from the right is computed. If `x0 =`

`complexInfinity`, then an undirected expansion around $u = 0$ is computed. Finally, $u = \pm\frac{1}{x}$ is substituted in the result.

Mathematically, the result of an expansion around `complexInfinity` or ±`infinity` is a power series in $\frac{1}{x}$. See “Example 4” on page 1-2074.

With the default mode `RelativeOrder`, the number of requested terms for the expansion is `order` if specified. If no `order` is specified, the value of the environment variable `ORDER` used. You can change the default value 6 by assigning a new value to `ORDER`.

The number of terms is counted from the lowest degree term on for finite expansion points, and from the highest degree term on for expansions around infinity, i.e., “`order`” has to be regarded as a “relative truncation order”.

If `AbsoluteOrder` is specified, `order` represents the truncation order of the series (i.e., the `x` power in the Big-Oh term).

`taylor` uses the more general series function `series` to compute the Taylor expansion. See the corresponding help page for `series` for details about the parameters and the data structure of a Taylor series expansion.

Environment Interactions

The function is sensitive to the environment variable `ORDER`, which determines the default number of terms in series computations.

Examples

Example 1

Compute a Taylor series around the default point 0:

```
s := taylor(exp(x^2), x)
```

$$1 + x^2 + \frac{x^4}{2} + O(x^6)$$

The result of `taylor` is of the following domain type:

```
domtype(s)
```

```
Series::Puiseux
```

If you apply the function `expr` to a series, the result is an arithmetical expression without the order term:

```
expr(s)
```

$$\frac{x^4}{2} + x^2 + 1$$

```
domtype(%)
```

```
DOM_EXPR
```

```
delete s:
```

Example 2

A Taylor series expansion of $f(x) = \frac{1}{x^2-1}$ around $x = 1$ does not exist. Therefore, `taylor` throws an error:

```
taylor(1/(x^2 - 1), x = 1)
```

```
Error: Cannot compute a Taylor expansion of '1/(x^2 - 1)'. Try 'series' for a more gene
```

Call `series` to compute a more general series expansion. A Laurent expansion does exist:

```
series(1/(x^2 - 1), x = 1)
```

$$\frac{1}{2\,(x-1)} - \frac{1}{4} + \frac{x-1}{8} - \frac{(x-1)^2}{16} + \frac{(x-1)^3}{32} - \frac{(x-1)^4}{64} + O((x-1)^5)$$

Example 3

If `taylor` cannot find a Taylor series expansion, it also throws an error.

```
taylor(psi(1/x), x = 0)
```

```
Error: Cannot compute a Taylor expansion of 'psi(1/x)'. Try 'series' with the 'Left',
```

Call `series` with the optional argument. In this case, `series` returns a more general type of expansion. In cases where `series` cannot find a series expansion, it returns the symbolic function call.

```
series(psi(1/x), x = 0, Right)
```

$$-\ln(x) - \frac{x}{2} - \frac{x^2}{12} + \frac{x^4}{120} + O(x^6)$$

Example 4

This is an example of a directed Taylor expansion along the real axis around `infinity`:

```
taylor(exp(1/x), x = infinity)
```

$$1 + \frac{1}{x} + \frac{1}{2\,x^2} + \frac{1}{6\,x^3} + \frac{1}{24\,x^4} + \frac{1}{120\,x^5} + O\left(\frac{1}{x^6}\right)$$

In fact, this is even an undirected expansion:

```
taylor(exp(1/x), x = complexInfinity)
```

$$1 + \frac{1}{x} + \frac{1}{2\,x^2} + \frac{1}{6\,x^3} + \frac{1}{24\,x^4} + \frac{1}{120\,x^5} + O\left(\frac{1}{x^6}\right)$$

Parameters

`f`

An arithmetical expression representing a function in `x`

`x`

An identifier or an indexed identifier

`x0`

The expansion point: an arithmetical expression. Also expressions involving `infinity` or `complexInfinity` are accepted.

If not specified, the default expansion point 0 is used.

`order`

The truncation order (in conjunction with `AbsoluteOrder`) or, in conjunction with `RelativeOrder`, the number of terms to be computed, respectively. A nonnegative integer; the default order is given by the environment variable `ORDER` (default value 6).

`mode`

One of the flags `AbsoluteOrder` or `RelativeOrder`. The default is `RelativeOrder`.

Options

`AbsoluteOrder`

With this flag, the integer value `order` is the truncation order of the computed series (i.e., the exponent of `x` in the Big-Oh term).

`RelativeOrder`

With this flag, the exponents of `x` in the computed series range from some leading order `v` to the highest exponent `v + order - 1` (i.e., the exponent of `x` in the Big-Oh term is `v + order`). In this case, `order` essentially is the "number of `x` powers" in the computed series if the series involves all integer powers of `x`.

Return Values

Object of domain type `Series::Puiseux` or a symbolic expression of type `"taylor"`.

Overloaded By

f

See Also

MuPAD Functions
asympt | diff | limit | mtaylor | O | series | Series::Puiseux | Type::Series

tbl2text

Concatenate the strings in a table

Syntax

```
tbl2text(strtab)
```

Description

`tbl2text` concatenates all entries of a table of character strings.

The table must be indexed by 1, 2, 3 etc. All entries must be character strings. They are concatenated in the order of their indices.

`tbl2text` restores strings split by `text2tbl`.

Examples

Example 1

A character string can be created from an arbitrary number of table entries:

```
tbl2text(table(1 = "Hell", 2 = "o", 3 = " ", 4 = "world."))
```

```
"Hello world."
```

Parameters

`strtab`

A table of character strings

Return Values

Character string.

See Also

MuPAD Functions

`_concat` | `coerce` | `expr2text` | `int2text` | `text2expr` | `text2list` | `text2tbl`

tcoeff

Trailing coefficient of a polynomial

Syntax

```
tcoeff(p, <order>)
```

```
tcoeff(f, <vars>, <order>)
```

Description

`tcoeff(p)` returns the trailing coefficient of the polynomial `p`.

The returned coefficient is "trailing" with respect to the lexicographical ordering, unless a different ordering is specified via the argument `order`. Cf. "Example 1" on page 1-2079.

A polynomial expression `f` is first converted to a polynomial with the variables given by `vars`. If no variables are given, they are searched for in `f`. See `poly` about details of the conversion. The result is returned as polynomial expression. `FAIL` is returned if `f` cannot be converted to a polynomial. Cf. "Example 3" on page 1-2080.

The result of `tcoeff` is not fully evaluated. Evaluation can be enforced by the function `eval`. Cf. "Example 2" on page 1-2080.

Examples

Example 1

We demonstrate how various orderings influence the result:

```
p := poly(5*x^2*y^3 + 4*x^3*y*z + 3*x*y^4*z, [x, y, z]):
tcoeff(p), tcoeff(p, DegreeOrder), tcoeff(p, DegInvLexOrder)
```

```
3, 5, 4
```

The following call uses the reverse lexicographical order on 3 indeterminates:

```
tcoeff(p, Dom::MonomOrdering(RevLex(3)))
```

$$4$$

```
delete p:
```

Example 2

The result of `tcoeff` is not fully evaluated:

```
p := poly(27*x^2 + a*x, [x]): a := 5:
tcoeff(p), eval(tcoeff(p))
```

$$a, 5$$

```
delete p, a:
```

Example 3

The expression `1/x` may not be regarded as polynomial:

```
lterm(1/x)
```

$$\text{FAIL}$$

Parameters

`p`

A polynomial of type `DOM_POLY`

`f`

A polynomial expression

`vars`

A list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

`order`

The term ordering: either `LexOrder`, or `DegreeOrder`, or `DegInvLexOrder`, or a user-defined term ordering of type `Dom::MonomOrdering`. The default is the lexicographical ordering `LexOrder`.

Return Values

Element of the coefficient domain of the polynomial or `FAIL`.

Overloaded By

p

See Also

MuPAD Functions

`coeff` | `collect` | `degree` | `degreevec` | `ground` | `lcoeff` | `ldegree` | `lmonomial` | `lterm` | `monomials` | `nterms` | `nthcoeff` | `nthmonomial` | `nthterm` | `poly` | `poly2list`

testargs

Decide whether procedure arguments should be tested

Syntax

```
testargs()
```

```
testargs(b)
```

Description

Inside a procedure, `testargs` indicates whether the procedure should check its arguments.

Checking the input parameters of a procedure may be costly. For this reason, most functions of the MuPAD libraries are implemented according to the following philosophy:

If a procedure is called on the interactive level, i.e., if its parameters are supplied interactively by the user, then the parameters should be checked. If the input parameters do not comply with the documented specification of the procedure, then appropriate error messages should be returned to notify the user of wrong usage.

If the procedure is called by another procedure, then no check of the parameters should be performed to improve efficiency. The calling procedure is supposed to make sure that appropriate parameters are passed.

`testargs` is the tool to check whether the arguments should be tested: called inside the body of a procedure, `testargs()` returns `TRUE` if the procedure was called on the interactive level. Otherwise, it returns `FALSE`.

`testargs` has two modes. In the “standard mode”, its functionality is as described above. In the “argument checking mode”, the call `testargs()` always returns `TRUE`. This supports the debugging of procedures: any function using `testargs` checks its parameters and returns useful error messages if called in an inappropriate way.

The call `testargs(TRUE)` switches to the “argument checking mode”, i.e., parameter testing is switched on globally.

The call `testargs(FALSE)` switches to the “standard mode”, i.e., parameter testing is used only on the interactive level.

The call `testargs(b)` returns the previously set value.

`testargs` should not be used to change the behavior of a function other than performing type-checks, since the user may have switched to “argument checking mode”.

Checking the input parameters of a procedure can also be controlled with the function `Pref::typeCheck`.

Examples

Example 1

The following example demonstrates how `testargs` should be used inside a procedure. The function `p` is to generate a sequence of `n` zeroes; its argument should be a positive integer:

```
p := proc(n)
begin
   if testargs() then
       if not testtype(n, Type::PosInt) then
          error("expecting a positive integer");
       end_if;
   end_if;
   return(0 $ n)
end_proc:
```

Its argument is checked when `p` is called on the interactive level:

```
p(13/2)
```

```
Error: expecting a positive integer [p]
```

Calling `p` from within a procedure with an inappropriate parameter does not invoke the argument testing. The following strange output is caused by the attempt to evaluate `0 $ n`:

```
f := proc(n) begin p(n) end_proc:  f(13/2)
```

$$0\ \$\ \frac{13}{2}$$

We switch on the “argument checking mode” of `testargs`:

```
testargs(TRUE):
```

Now also a non-interactive call to `p` produces an informative error message:

```
f(13/2)
```

```
Error: expecting a positive integer [p]
```

We clean up, restoring the “standard mode” of `testargs`:

```
testargs(FALSE): delete f, g:
```

Parameters

b

TRUE or FALSE

Return Values

TRUE or FALSE.

See Also

MuPAD Functions

`Pref::typeCheck` | `proc` | `testtype`

testeq

Check the mathematical equivalence of expressions

Syntax

```
testeq(ex1, options)

testeq(ex1, ex2, options)
```

Description

`testeq(ex1, ex2)` checks whether the expressions `ex1` and `ex2` are mathematically equivalent.

`testeq(ex1, ex2)` returns `TRUE` if the difference `ex1 - ex2` can be simplified to zero.

`testeq` returns `FALSE` if `ex1` and `ex2` attain different values for at least one choice of variables contained in them.

By default, `testeq` performs five random tests. If randomly chosen values of the variables are inconsistent with the assumptions on these variables or the test returns the value `undefined`, the `testeq` function performs an additional test. The number of additional tests cannot exceed the number of initial tests. By default, the maximal total number of tests is 10. See “Example 4” on page 1-2087.

If the equivalence of `ex1` and `ex2` cannot be decided, `testeq` returns `UNKNOWN`.

If only one expression is passed to `testeq`, it is checked whether this expression is equivalent to zero.

`testeq` uses `Simplify(ex1 - ex2)` and `is(ex1 - ex2 = 0)` to determine its result. The result `UNKNOWN` can be caused by weaknesses of `Simplify` and `is`.

Using the options, the simplification process can be made stronger at the cost of increased run time.

Examples

Example 1

Check the mathematical equivalence of expressions:

```
testeq(sin(x)^2, 1 - cos(x)^2)
```

```
    TRUE
```

```
testeq(sin(2*x), 2*sin(x)*cos(x))
```

```
    TRUE
```

```
testeq((cos(a) + sin(a))^2, 2*(cos(PI/4 - a)^2))
```

```
    TRUE
```

In order to be equivalent, two expressions must be equivalent for all values their variables can attain. For certain values of the parameter `a` the following two expressions are equivalent, but for other values they are not; therefore, they are not equivalent:

```
testeq((cos(a) + sin(a))^2, 3*(cos(PI/4 - a)^2))
```

```
    FALSE
```

Example 2

Applying `expand` and `rewrite` to an expression always produces an equivalent expression. However, with the default setting of `100` steps for the internal simplification procedure, the equivalence is not recognized in the following example:

```
f:= exp(arcsin(I*sin(x))):
g:= rewrite(expand(f), ln):
testeq(f, g)
```

```
    UNKNOWN
```

After 1000 steps, however, the expressions are recognized as being equivalent:

```
testeq(f, g, Steps = 1000);
delete f, g:
```

```
    TRUE
```

Example 3

When trying to prove the equivalence of two expressions, the `testeq` command runs random tests before applying `IgnoreAnalyticConstraints`. If tests for random values of identifiers show that expressions are not equivalent, `testeq` disregards the `IgnoreAnalyticConstraints` option and returns FALSE:

```
testeq(x^(ln(a))*x^(ln(b)) = x^(ln(a*b)),
               IgnoreAnalyticConstraints)
```

```
    FALSE
```

If, for a given number of attempts, random tests do not find the inequality between expressions, `testeq` applies the `IgnoreAnalyticConstraints` option:

```
testeq(ln(a) + ln(b) = ln(a*b), IgnoreAnalyticConstraints)
```

```
    TRUE
```

By default, random tests check the equality of expressions for five random sets of values of identifiers. Increasing the number of attempts can prove inequality:

```
testeq(ln(a) + ln(b) = ln(a*b), NumberOfRandomTests = 10,
                                IgnoreAnalyticConstraints)
```

```
    FALSE
```

Example 4

When `testeq` performs tests, it takes into account the assumptions on variables that you specify:

```
testeq(x, abs(x)) assuming x > 0
```

```
TRUE
```

If `testeq` chooses values of the variables that are inconsistent with the assumptions on these variables, it performs an additional test. The number of tests cannot exceed `2n`, where `n` is the original number of tests defined by the `NumberOfRandomTests` option. If `testeq` performs `2n` tests and all values of the variables are inconsistent with the assumptions on the variables, `testeq` returns `UNKNOWN`:

```
testeq(x, abs(x)) assuming x^2 + x + 7 = x^13 + 11
```

```
UNKNOWN
```

For this particular assumption, MuPAD cannot find a closed-form expression to substitute for `x`:

```
solve(x^2 + x + 7 = x^13 + 11, x)
```

$$\mathrm{RootOf}(z^{13} - z^2 - z + 4,\ z)$$

Therefore, increasing the number of tests does not help `testeq` decide if the expressions are equivalent:

```
testeq(x, abs(x), NumberOfRandomTests = 100)
            assuming x^2 + x + 7 = x^13 + 11
```

```
UNKNOWN
```

Parameters

`ex1, ex2`

Any MuPAD expressions

Options

`Steps`

Option, specified as `Steps = n`

This option is directly passed to `Simplify` and determines the maximum number of steps allowed for the internal simplification process. The default value of `n` is `100`. Increasing the number of steps can give you a simpler result, often at the costs of increased runtime. For details, see the `Simplify` help page.

`Seconds`

Option, specified as `Seconds = t`

This option is directly passed to `Simplify` and sets a time limit `t` in seconds for the internal simplification process. The default setting is `infinity`, i.e., the simplification process will not terminate due to a time limitation, but due to other internal stopping criteria. See the documentation of `Simplify` for details.

`RuleBase`

Option, specified as `RuleBase = base`

This option is directly passed to `Simplify` and determines the rule base that is used for the internal simplification process. See the documentation of `Simplify` for details.

The default value of `base` is `Simplify`.

The advanced user can specify her own rule base (see `Simplify`). This allows the construction of specialized and fast tests for special classes of expressions.

`NumberOfRandomTests`

Option, specified as `NumberOfRandomTests = n`

This option determines the number of times `testeq` tries to disprove the equivalence of `ex1` and `ex2` by plugging in some random values for all identifiers.

The default value of `n` is 5. If randomly chosen values of the variables are inconsistent with the assumptions on these variables or the test returns the value `undefined`, the `testeq` function performs an additional test. The total number of tests does not exceed `2n`. See “Example 4” on page 1-2087.

`IgnoreAnalyticConstraints`

This option applies purely algebraic simplifications to expressions `ex1` and `ex2`. For the list of rules, see the documentation of `Simplify`. These simplification rules are not generally valid.

Note that random tests have higher priority than `IgnoreAnalyticConstraints`. When trying to prove the equivalence of two expressions, the `testeq` command runs random tests before applying the `IgnoreAnalyticConstraints` option. If random tests prove the expressions are not equivalent, `testeq` returns the value FALSE. See "Example 3" on page 1-2087.

Return Values

TRUE, FALSE, or UNKNOWN

See Also

MuPAD Functions
`is` | `Simplify` | `simplify`

More About

- "Test Results"

testtype

Syntactical type checking

Syntax

```
testtype(object, T)
```

Description

`testtype(object, T)` checks whether the object is syntactically of type `T`.

The type object `T` may be either a domain type such as `DOM_INT`, `DOM_EXPR` etc., a string as returned by the function `type`, or a Type object. The latter are probably the most useful predefined values for the argument `T`.

Note: `testtype` performs a purely syntactical check. Use `is` for semantical checks taking into account properties of identifiers!

See the Algorithms section below for details on the overloading mechanism.

Examples

Example 1

The following call tests, whether the first argument is an expression. Expressions are basic objects of domain type `DOM_EXPR`:

```
testtype(x + y, DOM_EXPR)
```

```
TRUE
```

The `type` function distinguishes expressions. The corresponding type string is a valid type object for `testtype`:

```
type(x + y), testtype(x + y, "_plus")
```

```
"_plus", TRUE
```

The following call tests, whether the first argument is an integer by querying, whether it is of domain type `DOM_INT`:

```
testtype(7, DOM_INT)
```

```
TRUE
```

Note that `testtype` performs a purely syntactical test. Mathematically, the integer 7 is a rational number. However, the domain type `DOM_RAT` does not encompass `DOM_INT`:

```
testtype(7, DOM_RAT)
```

```
FALSE
```

The Type library provides more flexible type objects. E.g., `Type::Rational` represents the union of `DOM_INT` and `DOM_RAT`:

```
testtype(7, Type::Rational)
```

```
TRUE
```

The number 7 matches other types as well:

```
testtype(7, Type::PosInt), testtype(7, Type::Prime),
testtype(7, Type::Numeric), testtype(7, Type::Odd)
```

```
TRUE, TRUE, TRUE, TRUE
```

Example 2

Subtypes of expressions can be specified via character strings:

```
type(f(x)),  type(sin(x))
```

```
"function", "sin"
```

```
testtype(sin(x), "function"), testtype(sin(x), "sin"),
testtype(sin(x), "cos")
```

```
TRUE, TRUE, FALSE
```

Example 3

We demonstrate how to implement a customized type object "`div3`" which is to represent integer multiples of 3. One has to create a new domain with a "`testtypeDom`" attribute:

```
div3 := newDomain("divisible by 3?"):
div3::testtypeDom := x -> testtype(x/3, Type::Integer):
```

Via overloading, the command `testtype(object, div3)` calls this slot:

```
testtype(5, div3), testtype(6, div3), testtype(sin(1), div3)
```

```
FALSE, TRUE, FALSE
```

```
delete div3:
```

Parameters

`object`

Any MuPAD object

`T`

A type object

Return Values

TRUE or FALSE.

Overloaded By

object, T

Algorithms

Overloading of `testtype` works as follows: First, it is checked whether `domtype(object) = T` or `type(object) = T` holds. If so, `testtype` returns `TRUE`.

Next, the method `"testtype"` of the domain `object::dom` is called with the arguments `object, T`. If this method returns a result other than `FAIL`, then `testtype` returns this value.

If the method `object::dom::testtype` does not exist or if this method returns `FAIL`, then overloading by the second argument is used:

- If `T` is a domain, then the method `"testtypeDom"` of `T` is called with the arguments `object, T`.
- If `T` is not a domain, then the method `"testtypeDom"` of `T::dom` is called with the arguments `object, T`.

See Also

MuPAD Functions
`coerce` | `domtype` | `hastype` | `is` | `type`

text2expr

Convert a character string to an expression

Syntax

```
text2expr(text)
```

Description

`text2expr(text)` interprets the character string `text` as MuPAD input and generates the corresponding object.

The text must correspond to syntactically correct MuPAD input. Otherwise, `text2expr` produces an error. Typically, strings created from MuPAD objects via `expr2text` can be reconverted to corresponding objects.

The object is returned without being further evaluated. Evaluation can be enforced using the function `eval`.

The `text` does not need to be terminated with a "`;`" or a "`:`" character, respectively.

`text` cannot refer to local variables of an enclosing procedure by their name. The text is parsed as if entered interactively. Cf. "Example 4" on page 1-2097.

Examples

Example 1

A character string is converted to a simple expression. The newly created expression is not evaluated automatically:

```
text2expr("21 + 21")
```

$$21 + 21$$

It may be evaluated via `eval`:

```
eval(%)
```

```
     42
```

Example 2

A character string is converted to a statement sequence:

```
text2expr("x:= 3; x + 2 + 1"); eval(%)
```

```
(x := 3;
x + 2 + 1)
```

```
     6
```

```
x
```

```
     3
```

```
delete x:
```

Example 3

A matrix is converted to a string:

```
matrix([[a11, a12], [a21, a22]])
```

$$\begin{pmatrix} a11 & a12 \\ a21 & a22 \end{pmatrix}$$

```
expr2text(%)
```

```
     "matrix([[a11, a12], [a21, a22]])"
```

The string is reconverted to a matrix:

```
text2expr(%)
```

$$\begin{pmatrix} a11 & a12 \\ a21 & a22 \end{pmatrix}$$

```
eval(%)
```

$$\begin{pmatrix} a11 & a12 \\ a21 & a22 \end{pmatrix}$$

Example 4

`text2expr` will not create a `DOM_VAR` of an enclosing procedure from its name:

```
a := "global identifier":
g := proc()
       local a;
     begin
       a := "local variable";
       print(a);
       print(eval(text2expr("a")));
     end_proc:
g();
```

"local variable"

"global identifier"

Parameters

`text`

A character string

Return Values

MuPAD object.

See Also

MuPAD Functions

`coerce` | `expr2text` | `input` | `int2text` | `tbl2text` | `text2int` | `text2list` | `text2tbl`

text2int

Convert a character string to an integer

Syntax

```
text2int(text, <b>)
```

Description

`text2int(text, b)` converts a character string corresponding to an integer in `b`-adic representation to an integer of type `DOM_INT`.

It must consist of the first `b` characters in 0, 1, …, 9, *A*, *B*, …, *Z*, *a*, *b*, …, *z*. The letters are used to represent the `b`-adic digits larger than 9.

For bases larger than 10 but smaller than 37 the letters are not case sensitive. The lower case letters *a*, *b*, …, *z* are accepted: $a = A = 10, \ldots, z = Z = 35$.

`text2int` is the inverse of `int2text`.

Examples

Example 1

Relative to the default base 10, `text2int` provides a mere datatype conversion from `DOM_STRING` to `DOM_INT`:

```
text2int("123"), text2int("-45678")
```

$$123, -45678$$

Example 2

The characters of the input string are interpreted as digits with respect to the specified base, the return value is a standard MuPAD integer represented with respect to the

decimal system. The following example converts integers from the base 2 and 16, respectively, to the base 10:

```
text2int("101", 2), text2int("101", 16)
```

```
5, 257
```

The digit "3" does not exist in a binary representation:

```
text2int("103", 2)
```

```
Error: The argument is invalid. [text2int]
```

Example 3

For bases larger than 10 but smaller than 37, the letters are not case-sensitive:

```
text2int("3B9ACA00", 16), text2int("Z", 36) = text2int("z", 36)
```

```
1000000000, 35 = 35
```

For bases larger than 37 however, the case makes a difference:

```
text2int("Z", 62) <> text2int("z", 62)
```

```
35 ≠ 61
```

Parameters

`text`

A character string

`b`

The base: an integer between 2 and 62. The default base is 10.

Return Values

Integer.

See Also

MuPAD Functions

`coerce` | `expr2text` | `genpoly` | `int2text` | `numlib::g_adic` | `tbl2text` | `text2expr` | `text2list` | `text2tbl`

text2list

Split a character string into a list of substrings

Syntax

```
text2list(text, separators, <Cyclic>)
```

Description

`text2list` splits a character string into a list of substrings, using the strings in the list `separators` as delimiters. `text2list` returns a list containing the substrings.

Without the option `Cyclic`, the text is split as follows. The first occurrence of one of the delimiters in `separators` is located in `text`. If no delimiter is found, the full text is returned as the only substring. Otherwise, the substring up to the delimiter defines the first substring. The delimiter is the second substring. The remaining text is processed as above until there are no more characters left.

Without `Cyclic`, the result does not depend on the order of the delimiters.

With the option `Cyclic`, the first delimiter in `separators` is used to identify the first substring. The delimiter itself is the second substring. Then the second delimiter in `separators` is used to identify the third substring etc.

After using the last delimiter of the list, the first one is used again, until the whole text is processed or until the current delimiter is not found in the remaining text.

With `Cyclic`, the result depends on the order of the delimiters.

`text2list` is a function of the system kernel.

Examples

Example 1

The following example demonstrates the difference in calling `text2list` with and without the option `Cyclic`:

```
text2list("This is a simple example!", ["is", "mp"])
```

```
["Th", "is", " ", "is", " a si", "mp", "le exa", "mp", "le!"]
```

```
text2list("This is a simple example!", ["is", "mp"], Cyclic)
```

```
["Th", "is", " is a si", "mp", "le example!"]
```

Example 2

The following example demonstrates the difference in calling `text2tbl` with and without the option `Cyclic`:

```
text2tbl("This is a simple example!", ["is", "mp"])
```

1	"Th"
2	"is"
3	" "
4	"is"
5	" a si"
6	"mp"
7	"le exa"
8	"mp"
9	"le!"

```
text2tbl("This is a simple example!", ["is", "mp"], Cyclic)
```

1	"Th"
2	"is"
3	" is a si"
4	"mp"
5	"le example!"

Parameters

`text`

The text to be analyzed: a character string

separators

Delimiters: a list of character strings. The empty string `""` is not accepted as a delimiter.

Options

Cyclic

The delimiter list is used cyclicly

Return Values

List, respectively a table, of character strings.

See Also

MuPAD Functions

`coerce` | `expr2text` | `int2text` | `tbl2text` | `text2expr` | `text2int` | `text2tbl`

text2tbl

Split a character string into a table of substrings

Syntax

```
text2tbl(text, separators, <Cyclic>)
```

Description

`text2tbl` splits a character string into a table of substrings, using the strings in the list `separators` as delimiters. `text2tbl` returns a table, using the indices `1`, `2`, `3` etc.

Without the option `Cyclic`, the text is split as follows. The first occurrence of one of the delimiters in `separators` is located in `text`. If no delimiter is found, the full text is returned as the only substring. Otherwise, the substring up to the delimiter defines the first substring. The delimiter is the second substring. The remaining text is processed as above until there are no more characters left.

Without `Cyclic`, the result does not depend on the order of the delimiters.

With the option `Cyclic`, the first delimiter in `separators` is used to identify the first substring. The delimiter itself is the second substring. Then the second delimiter in `separators` is used to identify the third substring etc.

After using the last delimiter of the list, the first one is used again, until the whole text is processed or until the current delimiter is not found in the remaining text.

With `Cyclic`, the result depends on the order of the delimiters.

`tbl2text` restores strings split by `text2tbl`.

`text2tbl` is a function of the system kernel.

Examples

Example 1

The following example demonstrates the difference in calling `text2list` with and without the option `Cyclic`:

```
text2list("This is a simple example!", ["is", "mp"])
```

```
["Th", "is", " ", "is", " a si", "mp", "le exa", "mp", "le!"]
```

```
text2list("This is a simple example!", ["is", "mp"], Cyclic)
```

```
["Th", "is", " is a si", "mp", "le example!"]
```

Example 2

The following example demonstrates the difference in calling `text2tbl` with and without the option `Cyclic`:

```
text2tbl("This is a simple example!", ["is", "mp"])
```

1	"Th"
2	"is"
3	" "
4	"is"
5	" a si"
6	"mp"
7	"le exa"
8	"mp"
9	"le!"

```
text2tbl("This is a simple example!", ["is", "mp"], Cyclic)
```

1	"Th"
2	"is"
3	" is a si"
4	"mp"
5	"le example!"

Parameters

`text`

The text to be analyzed: a character string

`separators`

Delimiters: a list of character strings. The empty string `""` is not accepted as a delimiter.

Options

`Cyclic`

The delimiter list is used cyclicly

Return Values

List, respectively a table, of character strings.

See Also

MuPAD Functions

`coerce` | `expr2text` | `int2text` | `tbl2text` | `text2expr` | `text2int` | `text2list`

textinput

Interactive input of text

Syntax

```
textinput(<prompt1>)
```

```
textinput(<prompt1>, x1, <prompt2>, x2, …)
```

Description

`textinput` allows interactive input of text.

`textinput()` displays the prompt "`Please enter text:`" and waits for input by the user. The input is converted to a character string, which is returned as the function's return value.

`textinput(prompt1)` uses the character string `prompt1` instead of the default prompt "`Please enter text:`".

`textinput( prompt1 x1)` converts the input to a character string and assigns this string to the identifier or local variable `x1`. The default prompt is used, if no prompt string is specified.

Several input values can be read with a single `textinput` command. Each identifier in the sequence of arguments makes `textinput` return a prompt, waiting for input to be assigned to the identifier or variable. A character string preceeding the identifier or variable in the argument sequence replaces the default prompt. Cf. "Example 3" on page 1-2109. Arguments that are neither prompt strings nor identifiers or variables are ignored.

The input may extend over several lines. In the output string, MuPAD uses the character `\n` (carriage return) to separate lines.

Input characters with a leading `\` are not interpreted as control characters, but as two separate characters.

The identifiers or variables `x1` etc. may have values. These are overwritten by `textinput`.

Examples

Example 1

The default prompt is displayed, the input is converted to a character string and returned:

```
textinput()
```

```
Please enter text input: << myinput >>
```

```
     "myinput"
```

Example 2

A user-defined prompt is used, the input is assigned to the identifier `x`:

```
textinput("enter your name: ", x)
```

```
enter your name: << Turing >>
```

```
     "Turing"
```

```
x
```

```
     "Turing"
```

```
delete x:
```

Example 3

If several values are to be read, separate prompts can be defined for each value:

```
textinput("She: ", hername, "He:  ", hisname)
```

```
She: << Bonnie >> He:  <<
Clyde >>
```

```
     "Clyde"
```

```
hername, hisname
```

```
     "Bonnie", "Clyde"
```

```
delete hername, hisname:
```

Parameters

`prompt1, prompt2, …`

Input prompts: character strings

`x1, x2, …`

identifiers or local variables

Return Values

Last input, converted to a character string.

See Also

MuPAD Functions

`finput` | `fname` | `fprint` | `fread` | `ftextinput` | `import::readbitmap` | `import::readdata` | `input` | `print` | `read` | `text2expr` | `write`

TEXTWIDTH

Maximum number of characters in an output line

Description

The environment variable TEXTWIDTH determines the maximum number of characters in one line of screen output.

Possible values: Positive integer smaller than 2^{31}.

Output is broken into several lines if it needs more than TEXTWIDTH characters per line.

Deletion via the statement "`delete TEXTWIDTH`" resets TEXTWIDTH to its default value. Executing the function `reset` also restores the default value.

The minimal value of TEXTWIDTH is 10.

TEXTWIDTH is set to its maximum value 2^{31} - 1 when printing to a text file using `fprint`. Thus, no additional line breaks occur in the output.

TEXTWIDTH does not influence the typesetting of expressions which is available for some user interfaces of MuPAD.

TEXTWIDTH is set to the new number of available columns every time the console is resized.

Examples

Example 1

Set the maximum number of characters in one line of screen output to 15:

```
TEXTWIDTH := 15:
```

Restore TEXTWIDTH to its default value:

```
delete TEXTWIDTH
```

Example 2

The following procedure adds empty characters to produce output that is flushed right:

```
myprint := proc(x) local l; begin
             if domtype(x) <> DOM_STRING then
                x := expr2text(x);
             end_if;
             l := length(x);
             print(Unquoted, _concat(" " $ TEXTWIDTH - l, x))
           end_proc:

myprint("hello world"):  myprint(30!):  myprint("bye bye"):
```

```
                                                          hello world

                                    265252859812191058636308480000000

                                                            bye bye
```

```
delete myprint:
```

See Also

MuPAD Functions

`fprint` | `PRETTYPRINT` | `print`

theta

Theta series

Syntax

```
theta(x)
```

Description

`theta(x)` represents the value of the theta series $\sum_{n=-\infty}^{\infty} e^{-\pi x n^2}$.

The theta series converges for all complex numbers `x` with positive real part.

Floating-point results are computed for floating-point arguments. For other arguments, the function returns symbolically with the imaginary part of complex numbers normalized to lie between zero and 2.

Environment Interactions

When called with a floating-point argument, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

The theta series takes on large values for small positive arguments. Small values are taken on near *I*:

```
theta(0.001), theta(0.001 + I)
```

$$31.6227766,\ 5.092623095\ 10^{-340}$$

Example 2

Since the theta series is (2 i)-periodic, the imaginary part of complex arguments may be reduced:

```
theta(7 + 5*I)
```

$$\text{theta}(7 + i)$$

For exact or symbolic arguments, a symbolic call is returned:

```
theta(3), theta(x)
```

$$\text{theta}(3), \text{theta}(x)$$

Parameters

x

An arithmetical expression

Return Values

Arithmetical expression

See Also

MuPAD Functions
`dedekindEta` | `zeta`

rtime

Measure real time

Syntax

```
rtime()
```

```
rtime(a1, a2, …)
```

Description

`rtime()` returns the real time in milliseconds that elapsed since the start of the current MuPAD session.

`rtime(a1, a2, ...)` returns the real time needed to evaluate all arguments.

The result of `rtime` is the real time. Thus, `rtime` can be used to measure the total time spent by the MuPAD process as well as by external processes spawned from inside the MuPAD session. Note that an interactive call of `rtime()` is not very useful, since the idle time of the user is included. However, `rtime(a1, a2, ...)` often yields a useful and more realistic timing than `time(a1, a2, ...)` if the evaluation of the arguments spawns external processes. Such a situation may arise in a numerical computation because some routines of the `numeric` library call external numerical tools using hardware floats. Cf. “Example 4” on page 7-93.

If no external process besides MuPAD are running, the timings returned by `rtime(a1, a2, ...)` and `time(a1, a2, ...)` roughly coincide.

On computers without “time-sharing”, such as the Macintosh computer, real time and CPU time roughly coincide.

`rtime` is a function of the system kernel.

Examples

Example 1

This example shows how to do a time measurement and assign the computed value to an identifier at the same time. Note that the assignment needs extra parenthesis when passed as argument:

```
time((a := int(exp(x)*sin(x), x)))
```

464.029

```
a
```

$$-\frac{e^x\,(\cos(x)-\sin(x))}{2}$$

```
delete a:
```

Alternatively, one may time groups of statements in the following way:

```
t0 := time():
command1
command2
...
time() - t0
```

Example 2

Here we use `rtime` to compute the elapsed hours, minutes and seconds since this session was started:

```
t := rtime()/1000:
h := trunc(t/3600):
m := trunc(t/60 - h*60):
s := trunc(t - m*60 - h*3600):

print(Unquoted, "This session is running for " .
                h . " hours, " .
```

```
                m . " minutes and " .
                s . " seconds.")
```

```
This
session is running for 0 hours, 0 minutes and 10 seconds.
```

```
delete t, h, m, s:
```

Example 3

To obtain a nicer output, the measured time can be multiplied with the appropriate time unit:

```
time(isprime(2^1000000000 - 1))*unit::msec
```

```
                    280.018 msec
```

Alternatively, use `stringlib::formatTime`:

```
stringlib::formatTime(time(isprime(2^1000000000 - 1)))
```

```
                  "0.280017 seconds"
```

Parameters

`a1, a2, …`

Arbitrary MuPAD objects

Return Values

Nonnegative integer giving the elapsed time in milliseconds.

See Also

MuPAD Functions

`prog::profile` | `time`

More About

- “Measure Time”

time

Measure CPU time

Syntax

```
time()
```

```
time(a1, a2, …)
```

Description

`time()` returns the total CPU time in milliseconds that was spent by the current MuPAD process.

`time(a1, a2, ...)` returns the CPU time needed by the current MuPAD process to evaluate all arguments.

The result of `time()` comprises all the computation time spent by the MuPAD process. This includes the time for system initialization and reading input (parsing). However, it excludes the time spent by other external processes, even if they were spawned from inside the MuPAD session or if they were started by a `system` command. Further, in an interactive session, the idle time between the execution of MuPAD commands is excluded.

If no external process besides MuPAD are running, the timings returned by `rtime(a1, a2, ...)` and `time(a1, a2, ...)` roughly coincide.

The time returned by `time` is computed in a system-dependent way, usually counting the number of clock ticks of the system clock. Hence, the result is a multiple of the system's time unit and cannot be more precise than 1 such unit. E.g., the time unit is 10 milliseconds for many UNIX systems.

On computers without “time-sharing”, such as the Macintosh computer, real time and CPU time roughly coincide.

`time` is a function of the system kernel.

Examples

Example 1

This example shows how to do a time measurement and assign the computed value to an identifier at the same time. Note that the assignment needs extra parenthesis when passed as argument:

```
time((a := int(exp(x)*sin(x), x)))
```

464.029

```
a
```

$$-\frac{e^x\,(\cos(x)-\sin(x))}{2}$$

```
delete a:
```

Alternatively, one may time groups of statements in the following way:

```
t0 := time():
command1
command2
...
time() - t0
```

Example 2

Here we use `rtime` to compute the elapsed hours, minutes and seconds since this session was started:

```
t := rtime()/1000:
h := trunc(t/3600):
m := trunc(t/60 - h*60):
s := trunc(t - m*60 - h*3600):

print(Unquoted, "This session is running for " .
                h . " hours, " .
```

```
                m . " minutes and " .
                s . " seconds.")
```

```
This
session is running for 0 hours, 0 minutes and 10 seconds.
```

```
delete t, h, m, s:
```

Example 3

To obtain a nicer output, the measured time can be multiplied with the appropriate time unit:

```
time(isprime(2^1000000000 - 1))*unit::msec
```

```
                    280.018 msec
```

Alternatively, use `stringlib::formatTime`:

```
stringlib::formatTime(time(isprime(2^1000000000 - 1)))
```

```
                  "0.280017 seconds"
```

Parameters

`a1, a2, …`

Arbitrary MuPAD objects

Return Values

Nonnegative integer giving the elapsed time in milliseconds.

See Also

MuPAD Functions

`prog::profile` | `rtime`

More About

- “Measure Time”

transpose

Transpose of a matrix

Syntax

```
transpose(A)
```

Description

`transpose(A)` returns the transpose A^t of the matrix A.

The transpose of the $m \times n$ matrix A is the $n \times m$ matrix B with $B_{i,j} = A_{j,i}$.

If the input is a matrix of category `Cat::Matrix`, then `linalg::transpose` is called to compute the result. In contrast to the linalg routines, the function `transpose` also operates on `array`s and `hfarray`s.

If the argument does not evaluate to a matrix of one of the types mentioned above, symbolic call `transpose(A)` is returned.

Examples

Example 1

The following matrix is real. Thus, the Hermitian transpose coincides with the transpose:

```
A := array(1..2, 1..2, [[1, 2], [3, PI]])
```

$$\begin{pmatrix} 1 & 2 \\ 3 & \pi \end{pmatrix}$$

```
transpose(A) = htranspose(A)
```

$$\begin{pmatrix} 1 & 3 \\ 2 & \pi \end{pmatrix} = \begin{pmatrix} 1 & 3 \\ 2 & \pi \end{pmatrix}$$

In general, this does not hold for complex matrices:

```
A := hfarray(1..2, 1..3, [[1, I, 3 + I], [PI*I, 4, 5]])
```

$$\begin{pmatrix} 1.0 & 1.0\,i & 3.0+1.0\,i \\ 3.141592654\,i & 4.0 & 5.0 \end{pmatrix}$$

```
transpose(A) <> htranspose(A)
```

$$\begin{pmatrix} 1.0 & 3.141592654\,i \\ 1.0\,i & 4.0 \\ 3.0+1.0\,i & 5.0 \end{pmatrix} \neq \begin{pmatrix} 1.0 & -3.141592654\,i \\ -1.0\,i & 4.0 \\ 3.0-1.0\,i & 5.0 \end{pmatrix}$$

```
delete A:
```

Example 2

We compute the product $A^H A$ of a matrix given by a `hardware float array`. This data type allows matrix multiplication using the operator `*`:

```
A := hfarray(1..2, 1..3, [[1, I, 3], [PI*I, 4, 5 + I]])
```

$$\begin{pmatrix} 1.0 & 1.0\,i & 3.0 \\ 3.141592654\,i & 4.0 & 5.0+1.0\,i \end{pmatrix}$$

```
AH:= htranspose(A)
```

$$\begin{pmatrix} 1.0 & -3.141592654\,i \\ -1.0\,i & 4.0 \\ 3.0 & 5.0-1.0\,i \end{pmatrix}$$

The product $A^H A$ is Hermitian:

```
AH*A = htranspose(AH*A)
```

$$\begin{pmatrix} 10.8696044 & -11.56637061\,i & 6.141592654 - 15.70796327\,i \\ 11.56637061\,i & 17.0 & 20.0 + 1.0\,i \\ 6.141592654 + 15.70796327\,i & 20.0 - 1.0\,i & 35.0 \end{pmatrix} =$$

$$\begin{pmatrix} 10.8696044 & -11.56637061\,i & 6.141592654 - 15.70796327\,i \\ 11.56637061\,i & 17.0 & 20.0 + 1.0\,i \\ 6.141592654 + 15.70796327\,i & 20.0 - 1.0\,i & 35.0 \end{pmatrix}$$

```
delete A, AH:
```

Example 3

If the input does not evaluate to a matrix, then symbolic calls are returned:

```
delete A, B:
transpose(A) + 2*htranspose(B)
```

$$2\,\overline{B}^{t} + A^{t}$$

Parameters

A

A matrix: either a 2-dimensional `array`, a 2-dimensional `hfarray`, or an object of the category `Cat::Matrix`

Return Values

Matrix of the same domain type as A.

Overloaded By

A

See Also

MuPAD Functions

`htranspose` | `linalg::htranspose` | `linalg::transpose`

More About

- "Transpose Matrices"

htranspose

The Hermitian transpose of a matrix

Syntax

```
htranspose(A)
```

Description

`htranspose(A)` returns the Hermitian transpose A^H of the matrix A (the complex conjugate of the transpose of A).

The Hermitian transpose of the $m \times n$ matrix A is the $n \times m$ matrix B with $B_{i,j} = \overline{A_{j,i}}$.

If the input is a matrix of category `Cat::Matrix`, then `linalg::htranspose` is called to compute the result. In contrast to the linalg routines, the function `htranspose` also operates on `array`s and `hfarray`s.

If the argument does not evaluate to a matrix of one of the types mentioned above, symbolic call `htranspose(A)` is returned.

Examples

Example 1

The following matrix is real. Thus, the Hermitian transpose coincides with the transpose:

```
A := array(1..2, 1..2, [[1, 2], [3, PI]])
```

$$\begin{pmatrix} 1 & 2 \\ 3 & \pi \end{pmatrix}$$

```
transpose(A) = htranspose(A)
```

$$\begin{pmatrix} 1 & 3 \\ 2 & \pi \end{pmatrix} = \begin{pmatrix} 1 & 3 \\ 2 & \pi \end{pmatrix}$$

In general, this does not hold for complex matrices:

```
A := hfarray(1..2, 1..3, [[1, I, 3 + I], [PI*I, 4, 5]])
```

$$\begin{pmatrix} 1.0 & 1.0\,i & 3.0+1.0\,i \\ 3.141592654\,i & 4.0 & 5.0 \end{pmatrix}$$

```
transpose(A) <> htranspose(A)
```

$$\begin{pmatrix} 1.0 & 3.141592654\,i \\ 1.0\,i & 4.0 \\ 3.0+1.0\,i & 5.0 \end{pmatrix} \neq \begin{pmatrix} 1.0 & -3.141592654\,i \\ -1.0\,i & 4.0 \\ 3.0-1.0\,i & 5.0 \end{pmatrix}$$

```
delete A:
```

Example 2

We compute the product $A^H A$ of a matrix given by a `hardware float array`. This data type allows matrix multiplication using the operator `*`:

```
A := hfarray(1..2, 1..3, [[1, I, 3], [PI*I, 4, 5 + I]])
```

$$\begin{pmatrix} 1.0 & 1.0\,i & 3.0 \\ 3.141592654\,i & 4.0 & 5.0+1.0\,i \end{pmatrix}$$

```
AH:= htranspose(A)
```

$$\begin{pmatrix} 1.0 & -3.141592654\,i \\ -1.0\,i & 4.0 \\ 3.0 & 5.0-1.0\,i \end{pmatrix}$$

The product $A^H A$ is Hermitian:

```
AH*A = htranspose(AH*A)
```

$$\begin{pmatrix} 10.8696044 & -11.56637061\,i & 6.141592654 - 15.70796327\,i \\ 11.56637061\,i & 17.0 & 20.0 + 1.0\,i \\ 6.141592654 + 15.70796327\,i & 20.0 - 1.0\,i & 35.0 \end{pmatrix} =$$

$$\begin{pmatrix} 10.8696044 & -11.56637061\,i & 6.141592654 - 15.70796327\,i \\ 11.56637061\,i & 17.0 & 20.0 + 1.0\,i \\ 6.141592654 + 15.70796327\,i & 20.0 - 1.0\,i & 35.0 \end{pmatrix}$$

```
delete A, AH:
```

Example 3

If the input does not evaluate to a matrix, then symbolic calls are returned:

```
delete A, B:
transpose(A) + 2*htranspose(B)
```

$$2\,\overline{B}^{t} + A^{t}$$

Parameters

A

A matrix: either a 2-dimensional `array`, a 2-dimensional `hfarray`, or an object of the category `Cat::Matrix`

Return Values

Matrix of the same domain type as `A`.

Overloaded By

A

See Also

MuPAD Functions

`linalg::htranspose` | `linalg::transpose` | `transpose`

More About

- "Transpose Matrices"

traperror

Trap errors

Syntax

```
traperror(object)
```

```
traperror(object, t)
```

```
traperror(object, MaxSteps = s)
```

Description

`traperror(object)` traps errors produced by the evaluation of `object`.

`traperror(object, t)` does the same. Moreover, it stops the evaluation if it is not finished after a real time of `t` seconds.

`traperror` traps errors caused by the evaluation of the object. Syntactical errors, i.e., errors on parsing the object, cannot be caught. The same holds true for fatal errors causing the termination of MuPAD.

`traperror` returns the error code `0` if no error happened. The error code is `1320` if the given time limit `t` is exceeded ("`Execution time exceeded`") and `1321` if the given number of "execution steps" is exceeded. The error code is `1028` if the error was raised by the command `error`.

If `traperror` is called with a numerical second argument, this number is taken as a time limit, measured in seconds, of "process time" (see the documentation of the `time` function for a discussion of this term).

When using the option `MaxSteps = s`, the caller sets a time limit which is not system-dependent, but rather measured in terms of MuPAD evaluation steps.

The number `s` does *not* refer directly to evaluation steps, but rather to a fixed (large) number of steps which may change from one MuPAD release to the next, but is fixed within one release. The number `s` is twice the number of outputs caused by `Pref::report(9)` for a calculation using the maximum time allowed.

If traperror has no time limit set and an "Execution time exceeded" error is raised by an enclosing traperror(..., t) command, then this error is not trapped by the inner traperror. It is trapped by the traperror call that has set the time limit. Cf. "Example 5" on page 1-2134.

The object can be an assignment which, for syntactical reasons, must be enclosed in additional brackets. The following code fragment demonstrates a typical application of traperror:

```
if traperror((x := SomeErrorProneFunction())) = 0 then
     DoSomethingWith(x);
else RespondToTheError();
end_if;
```

Use lasterror to reproduce the trapped error.

Examples

Example 1

Errors that happen during the execution of kernel functions have various error codes, depending on the problem. E.g., "Division by zero" produces the error code 1025:

```
y := 1/x: traperror(subs(y, x = 0))
```

```
     1025
```

```
lasterror()
```

```
Error: Division by zero. [_power]
```

Example 2

All errors raised using the function error have the error code 1028. Errors during the execution of library functions are of this kind:

```
traperror(error("My error!"))
```

```
     1028

lasterror()

Error: My error!
```

Example 3

We try to factor a polynomial, but give up after ten seconds:

```
traperror(factor(x^1000 + 4*x + 1), 10)

     1320

lasterror()

 Error: Execution time exceeded;   Evaluating:
faclib::univ_mod_gcd
```

Example 4

For use inside other routines, it is preferable to use `MaxSteps` instead of a time limit, to achieve consistent results across slower and faster machines:

```
traperror(factor(x^1000 + 4*x + 1), MaxSteps=10)

     1321

lasterror()

 Error: Execution MaxSteps exceeded [traperror];
  Evaluating: faclib::ddf
```

Note that evaluation steps may take vastly different amounts of time, so even on the same machine, different expressions evaluated with the same value of `MaxSteps` may be terminated after very different lengths of time:

```
time(traperror(factor(x^1000 + 4*x + 1), MaxSteps=1));
time(traperror(while TRUE do 1 end_while, MaxSteps=1));
```

```
time(traperror(int(1/sqrt(1/r-1/r0), r=0..r0), MaxSteps=1))
```

```
                                2204
```

```
                                40
```

```
                                468
```

Example 5

Here we have two nested `traperror` calls. The inner call contains an unterminated loop and the outer call has a time limit of `2` seconds. When the execution time is exceeded, this special error is not trapped by the inner `traperror` call. Because of the error, `print(1)` is never executed:

```
traperror((traperror((while TRUE do 1 end)); print(1)), 2)
```

```
                                1320
```

```
lasterror()
```

```
Error: Execution time is exceeded.
```

Parameters

`object`

Any MuPAD object

`t`

The time limit: a positive integer

`s`

The execution limit: a positive integer

Return Values

Nonnegative integer.

See Also

MuPAD Functions
`error` | `getlasterror` | `lasterror`

triangularPulse

Triangular pulse function

Syntax

```
triangularPulse(a, b, c, x)
```

```
triangularPulse(a, c, x)
```

Description

`triangularPulse(a, b, c, x)` represents the triangular function.

`triangularPulse(a, c, x)` is a shortcut for `triangularPulse(a, (a + c)/2, c, x)`.

`triangularPulse(x)` is a shortcut for `triangularPulse(-1, 0, 1, x)`.

`triangularPulse` represents the triangular pulse function. This function is also called the triangle function, hat function, tent function, or sawtooth function.

If `a`, `b`, and `c` are variables or expressions with variables, `triangularPulse` assumes that `a <= b <= c`. If `a`, `b`, and `c` are numerical values that do not satisfy this condition, `triangularPulse` throws an error.

If `a < x < b`, the triangular function equals `(x - a)/(b - a)`. If `b < x < c`, the triangular function equals `(c - x)/(c - b)`. Otherwise, it equals 0. See "Example 1" on page 1-2137 and "Example 2" on page 1-2137.

If `a = b` or `b = c`, the triangular function can be expressed in terms of the rectangular function. See "Example 3" on page 1-2137.

If `a = b = c`, `triangularPulse` returns 0. See "Example 4" on page 1-2138.

`triangularPulse(x)` is equivalent to `triangularPulse(-1, 0, 1, x)`. See "Example 5" on page 1-2138.

`triangularPulse(a, c, x)` is equivalent to `triangularPulse(a, (a + c)/2, c, x)`. See "Example 6" on page 1-2138.

`triangularPulse` also accepts infinities as its arguments. See “Example 9” on page 1-2140.

`triangularPulse` and `tripulse` are equivalent.

Examples

Example 1

Compute the triangular pulse function for these input arguments:

```
[triangularPulse(-2, 0, 2, -3),
 triangularPulse(-2, 0, 2, -1/2),
 triangularPulse(-2, 0, 2, 0),
 triangularPulse(-2, 0, 2, 3/2),
 triangularPulse(-2, 0, 2, 3)]
```

$$\left[0, \frac{3}{4}, 1, \frac{1}{4}, 0\right]$$

Example 2

Compute the triangular pulse function for `a < x < b`:

```
triangularPulse(a, b, c, x) assuming a < x < b
```

$$\frac{a-x}{a-b}$$

Compute the triangular pulse function for `b < x < c`:

```
triangularPulse(a, b, c, x) assuming b < x < c
```

$$-\frac{c-x}{b-c}$$

Example 3

Compute the triangular pulse function for `a = b` and `c = b`:

```
triangularPulse(b, b, c, x) assuming b < c
```

$$\_\ \frac{(c-x)\ \text{rectangularPulse}(b, c, x)}{b-c}$$

```
triangularPulse(a, b, b, x) assuming a < b
```

$$\frac{(a-x)\ \text{rectangularPulse}(a, b, x)}{a-b}$$

Example 4

For `a = b = c`, the triangular pulse function returns 0:

```
triangularPulse(a, a, a, x)
```

$$0$$

Example 5

Use `triangularPulse` with one input argument as a shortcut for computing `triangularPulse(-1, 0, 1, x)`:

```
triangularPulse(x)
```

$$\text{triangularPulse}(-1, 0, 1, x)$$

```
[triangularPulse(-10),
 triangularPulse(-3/4),
 triangularPulse(0),
 triangularPulse(2/3),
 triangularPulse(1)]
```

$$\left[0, \frac{1}{4}, 1, \frac{1}{3}, 0\right]$$

Example 6

Use `triangularPulse` with three input arguments as a shortcut for computing `triangularPulse(a, (a + c)/2, c, x)`:

```
triangularPulse(a, c, x)
```

$$\text{triangularPulse}\left(a, \frac{a}{2}+\frac{c}{2}, c, x\right)$$

```
[triangularPulse(-10, 10, 3),
 triangularPulse(-1/2, -1/4, -2/3),
 triangularPulse(2, 4, 3),
 triangularPulse(2, 4, 6),
 triangularPulse(-1, 4, 0)]
```

$$\left[\frac{7}{10}, 0, 1, 0, \frac{2}{5}\right]$$

Example 7

Rewrite the triangular pulse function in terms of the Heaviside step function:

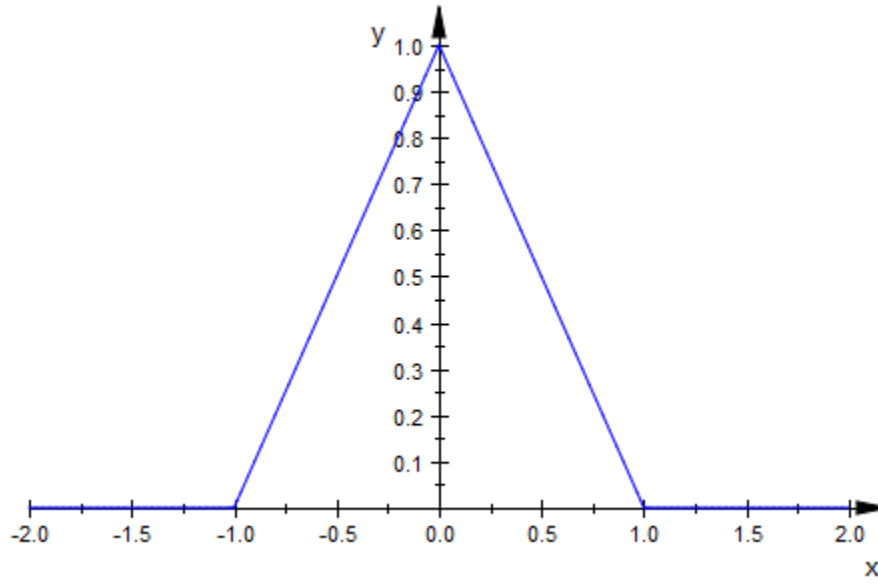
```
rewrite(triangularPulse(a, b, c, x), heaviside)
```

$$\frac{\text{heaviside}(x-a)\,(a-x)}{a-b} - \frac{\text{heaviside}(x-b)\,(a-x)}{a-b} - \frac{\text{heaviside}(x-b)\,(c-x)}{b-c} + \frac{\text{heaviside}(x-c)\,(c-x)}{b-c}$$

Example 8

Plot the triangular pulse function:

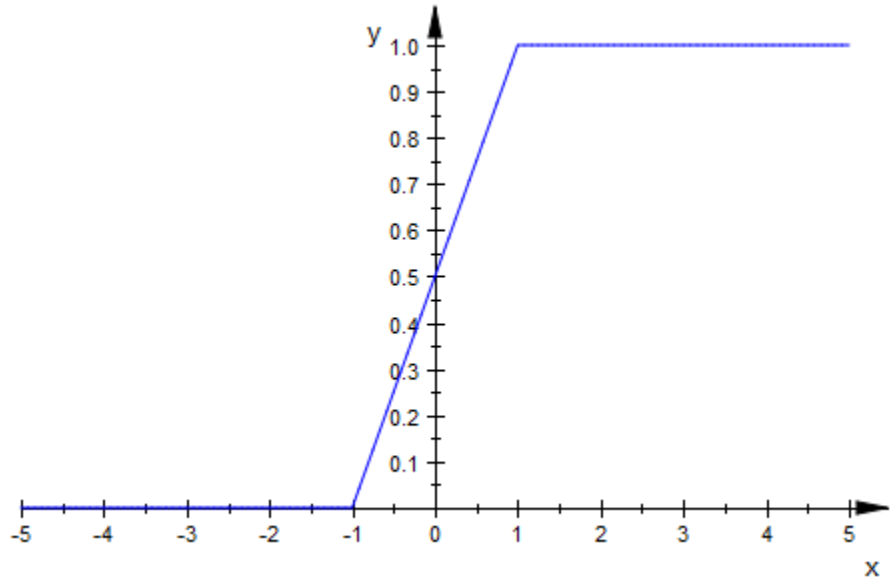
```
plot(triangularPulse(x), x = -2..2)
```



Example 9

Plot the triangular pulse function for which the argument `c` is a positive infinity:

```
plot(triangularPulse(-1, 1, infinity, x))
```



Parameters

a, b, c, x

Arithmetical expressions.

Return Values

Arithmetical expression.

Overloaded By

x

See Also

MuPAD Functions
`heaviside` | `piecewise` | `rectangularPulse`

tripulse

Triangular pulse function

Syntax

```
tripulse(a, b, c, x)
```

```
tripulse(a, c, x)
```

Description

`tripulse(a, b, c, x)` represents the triangular function.

`tripulse(a, c, x)` is a shortcut for `tripulse(a, (a + c)/2, c, x)`.

`tripulse(x)` is a shortcut for `tripulse(-1, 0, 1, x)`.

`tripulse` and `triangularPulse` are equivalent. These functions represent the triangular pulse function. For details and examples, see `triangularPulse`.

Parameters

`a`, `b`, `c`, `x`

Arithmetical expressions.

Return Values

Arithmetical expression.

Overloaded By

x

See Also

MuPAD Functions
`heaviside` | `piecewise` | `rectangularPulse` | `triangularPulse`

TRUE

Boolean constant TRUE

Description

MuPAD uses a three state logic with the Boolean constants `TRUE`, `FALSE`, and `UNKNOWN`.

The Boolean constants `TRUE`, `FALSE`, `UNKNOWN` are of domain type `DOM_BOOL`.

See and, or, not for the logical rules of the MuPAD three state logic.

Boolean constants are returned by system functions such as `bool` and `is`. These functions evaluate Boolean expressions such as equations and inequalities.

Examples

Example 1

The Boolean constants may be combined via `and`, `or`, and `not`:

```
(TRUE and (not FALSE)) or UNKNOWN
```

```
TRUE
```

Example 2

The function `bool` serves for reducing Boolean expressions such as equations or inequalities to one of the Boolean constants:

```
bool(x = x and 2 < 3 and 3 <> 4 or UNKNOWN)
```

```
TRUE
```

The function `is` evaluates symbolic Boolean expressions with properties:

```
assume(x > 2): is(x^2 > 4), is(x^3 < 0), is(x^4 > 17)
```

```
TRUE, FALSE, UNKNOWN
```

```
unassume(x):
```

Example 3

Boolean constants occur in the conditional part of program control structures such as `if`, `repeat`, or `while` statements. The following loop searches for the smallest Mersenne prime larger than 500 (see `numlib::mersenne` for details). The function `isprime` returns `TRUE` if its argument is a prime, and `FALSE` otherwise. Once a Mersenne prime is found, the `while`-loop is interrupted by the `break` statement:

```
p := 500:
while TRUE do
  p := nextprime(p + 1):
  if isprime(2^p - 1) then
     print(p);
     break;
  end_if;
end_while:
```

```
    521
```

Note that the conditional part of `if`, `repeat`, and `while` statements must evaluate to `TRUE` or `FALSE`. Any other value leads to an error:

```
if UNKNOWN then "true" else "false" end_if
```

```
Error: Cannot evaluate to Boolean. [if]
```

```
delete p:
```

See Also

MuPAD Domains
DOM_BOOL

MuPAD Functions
`_lazy_and` | `_lazy_or` | `and` | `bool` | `FALSE` | `if` | `is` | `not` | `or` | `repeat` | `UNKNOWN` | `while`

FALSE

Boolean constant FALSE

Description

MuPAD uses a three state logic with the Boolean constants TRUE, FALSE, and UNKNOWN.

The Boolean constants TRUE, FALSE, UNKNOWN are of domain type DOM_BOOL.

See and, or, not for the logical rules of the MuPAD three state logic.

Boolean constants are returned by system functions such as bool and is. These functions evaluate Boolean expressions such as equations and inequalities.

Examples

Example 1

The Boolean constants may be combined via and, or, and not:

```
(TRUE and (not FALSE)) or UNKNOWN
```

```
TRUE
```

Example 2

The function bool serves for reducing Boolean expressions such as equations or inequalities to one of the Boolean constants:

```
bool(x = x and 2 < 3 and 3 <> 4 or UNKNOWN)
```

```
TRUE
```

The function is evaluates symbolic Boolean expressions with properties:

```
assume(x > 2): is(x^2 > 4), is(x^3 < 0), is(x^4 > 17)
```

```
TRUE, FALSE, UNKNOWN
```

```
unassume(x):
```

Example 3

Boolean constants occur in the conditional part of program control structures such as `if`, `repeat`, or `while` statements. The following loop searches for the smallest Mersenne prime larger than 500 (see `numlib::mersenne` for details). The function `isprime` returns `TRUE` if its argument is a prime, and `FALSE` otherwise. Once a Mersenne prime is found, the `while`-loop is interrupted by the `break` statement:

```
p := 500:
while TRUE do
  p := nextprime(p + 1):
  if isprime(2^p - 1) then
     print(p);
     break;
  end_if;
end_while:
```

```
    521
```

Note that the conditional part of `if`, `repeat`, and `while` statements must evaluate to `TRUE` or `FALSE`. Any other value leads to an error:

```
if UNKNOWN then "true" else "false" end_if
```

```
Error: Cannot evaluate to Boolean. [if]
```

```
delete p:
```

See Also

MuPAD Domains
DOM_BOOL

MuPAD Functions
`_lazy_and` | `_lazy_or` | `and` | `bool` | `if` | `is` | `not` | `or` | `repeat` | `TRUE` | `UNKNOWN` | `while`

UNKNOWN

Boolean constant UNKNOWN

Description

MuPAD uses a three state logic with the Boolean constants `TRUE`, `FALSE`, and `UNKNOWN`.

The Boolean constants `TRUE`, `FALSE`, `UNKNOWN` are of domain type `DOM_BOOL`.

See and, or, not for the logical rules of the MuPAD three state logic.

Boolean constants are returned by system functions such as `bool` and `is`. These functions evaluate Boolean expressions such as equations and inequalities.

Examples

Example 1

The Boolean constants may be combined via `and`, `or`, and `not`:

```
(TRUE and (not FALSE)) or UNKNOWN
```

```
TRUE
```

Example 2

The function `bool` serves for reducing Boolean expressions such as equations or inequalities to one of the Boolean constants:

```
bool(x = x and 2 < 3 and 3 <> 4 or UNKNOWN)
```

```
TRUE
```

The function `is` evaluates symbolic Boolean expressions with properties:

```
assume(x > 2): is(x^2 > 4), is(x^3 < 0), is(x^4 > 17)
```

```
TRUE, FALSE, UNKNOWN
```

```
unassume(x):
```

Example 3

Boolean constants occur in the conditional part of program control structures such as `if`, `repeat`, or `while` statements. The following loop searches for the smallest Mersenne prime larger than 500 (see `numlib::mersenne` for details). The function `isprime` returns `TRUE` if its argument is a prime, and `FALSE` otherwise. Once a Mersenne prime is found, the `while`-loop is interrupted by the `break` statement:

```
p := 500:
while TRUE do
  p := nextprime(p + 1):
  if isprime(2^p - 1) then
     print(p);
     break;
  end_if;
end_while:
```

```
    521
```

Note that the conditional part of `if`, `repeat`, and `while` statements must evaluate to `TRUE` or `FALSE`. Any other value leads to an error:

```
if UNKNOWN then "true" else "false" end_if
```

```
Error: Cannot evaluate to Boolean. [if]
```

```
delete p:
```

See Also

MuPAD Domains
DOM_BOOL

MuPAD Functions
`_lazy_and` | `_lazy_or` | `and` | `bool` | `FALSE` | `if` | `is` | `not` | `or` | `repeat` | `TRUE` | `while`

type

Type of an object

Syntax

```
type(object)
```

Description

`type(object)` returns the type of the object.

If `object` is not an expression of domain type `DOM_EXPR`, then `type(object)` is equivalent to `domtype(object)`, i.e., `type` returns the domain type of the object.

If `object` is an expression of domain type `DOM_EXPR`, then its type is determined by its 0-th operand (the "operator"). If the operator has a `"type"` slot, then `type` returns this value, which usually is a string. If the operator has no `"type"` slot, then `type` returns the string `"function"`.

In contrast to most other functions, `type` does not flatten arguments that are expression sequences. Cf. "Example 4" on page 1-2152.

Examples

Example 1

If an object is not an expression, its type equals its domain type:

```
type(3)
```

```
DOM_INT
```

Example 2

The operator of a sum is `_plus`; the type slot of that operator is `"_plus"`:

```
type(x + y*z)
```

```
"_plus"
```

`type` evaluates its argument: thereby, the difference of `x` and `y` becomes the sum of `x` and `(-1)*y`. Its type is not `"_subtract"`, but `"_plus"`:

```
type(x - y)
```

```
"_plus"
```

Example 3

If the operator of an expression is not a function environment having a type slot, the expression is of type `"function"`:

```
type(f(2))
```

```
"function"
```

Example 4

The following call to `type` is *not* regarded as a call with two arguments, because expression sequences in the argument are not flattened:

```
type((2, 3))
```

```
"_exprseq"
```

Parameters

`object`

Any MuPAD object

Return Values

Domain type of type `DOM_DOMAIN` or a character string.

Overloaded By

object

See Also

MuPAD Functions
`coerce` | `domtype` | `hastype` | `testtype`

unassume

Delete the properties of an identifier

Syntax

```
unassume(x)
```

Description

`unassume(x)` deletes the properties of the identifier `x`.

`unassume` serves for deleting properties of identifiers set via `assume`. See "Properties" for a description of the property mechanism.

If `x` is a list or a set of identifiers, then the properties of all specified identifiers are deleted.

The command `delete x` deletes the value and the properties of the identifier `x`.

Examples

Example 1

Properties are attached to the identifiers `x` and `y`:

```
assume(x > 0): assume(y < 0): getprop(x), getprop(y)
```

$$(0, \infty), (-\infty, 0)$$

```
sign(x), sign(y)
```

$$1, -1$$

`unassume` or `delete` deletes the properties:

```
unassume(x): delete y: getprop(x), getprop(y)
```

$$\mathbb{C},\ \mathbb{C}$$

```
sign(x), sign(y)
```

$$\mathrm{sign}(x),\ \mathrm{sign}(y)$$

The properties of several identifiers can be deleted simultaneously by passing a list or a set to `unassume`:

```
assume(x > y): unassume([x, y]): getprop(x), getprop(y)
```

$$\mathbb{C},\ \mathbb{C}$$

Parameters

x

An identifier or a list or a set of identifiers

Return Values

Void object `null()`.

See Also

MuPAD Functions

`assume` | `delete` | `getprop` | `is`

undefined

Undefined value

Description

`undefined` indicates results of mathematically undefined operations.

MuPAD uses the special value `undefined` to indicate the results of operations that are not defined in mathematics.

You can use the `undefined` value as an input. Arithmetical operations involving `undefined` also return `undefined`. Multiplying infinities by 0 returns `undefined`.

For floating-point intervals, MuPAD uses the special value `RD_NAN` instead of `undefined`. If you use typeset mode, MuPAD displays `RD_NAN` as `NaN` in output regions. Multiplying infinities `RD_INF` and `RD_NINF` by 0 returns `RD_NAN`.

Examples

Example 1

Perform the following operations with infinities. MuPAD returns the `undefined` value for these operations:

```
0*infinity, infinity - infinity, infinity/infinity
```

```
undefined, undefined, undefined
```

Example 2

Compute the limit of the sine function at infinity. Since this limit does not exist (is not mathematically defined), MuPAD returns `undefined`:

```
limit(sin(x), x = infinity)
```

```
undefined
```

Example 3

Multiply infinities by 0:

```
infinity*0, -infinity*0
```

```
undefined, undefined
```

```
RD_INF*0, RD_NINF*0
```

```
NaN, NaN
```

See Also

MuPAD Functions
FAIL

More About

- “Mathematical Constants Available in MuPAD”

unit

Physical units

Syntax

```
unit::nam
```

Description

Objects such as `unit::m`, `unit::kg`, `unit::sec` etc. represent the physical units "meters", "kilograms", "seconds" etc.

`unit` domain provides some methods for simplifying and converting arithmetical expressions involving such units.

`unit` objects such as `unit::m` or `unit::kg` serve for representing physical units. They are domain objects of domain type `unit`.

These objects behave like symbolic names (identifiers) and can be used to build arithmetical expressions involving numbers and symbols such as `3*unit::m^2` or `a*unit::cm + b*unit::inch^2/unit::mm`. Units must be used consistently in expressions, and should be specified for unknown variables so that the variables themselves are dimensionless. In calls to `solve`, you should always solve for dimensionless variables by specifying the dimensions. Cf. "Example 3" on page 1-2167.

Expressions such as `20*unit::cm + 0.3*unit::m` involving several units of the same type ('length,' 'mass,' 'time' etc.) are *not* simplified automatically. Use `unit::convert`, `unit::simplify`, `simplify`, or `Simplify` to convert all subexpressions to common units. These routines are described further down below.

On input, use the prefix `unit::` as in `unit::mm`, `unit::km` for millimeters, kilometers etc. On the screen, this prefix is stripped off. For example, `1.23*unit::mm` is displayed as `1.23 mm`.

The available units are listed further down below.

Note that some units such as `unit::mm`, `unit::millimeter` and `unit::millimeters` represent the same physical unit. Use an interactive command such as `info(unit::oz)`

to find information on `unit::oz` including all the alternative names that can be used in MuPAD. Cf. “Example 7” on page 1-2170.

Note: Beware: If you mix different MuPAD units representing the same physical unit, *no automatic simplifications* are done! Use `simplify` to simplify an expression such as `2*unit::m + 3*unit::meter` to `5*unit::m`.

Expressions such as `unit::kg*(unit::m/unit::s)^2` can be used to represent a composite unit. You can convert them to other units such as `unit::Joule` via `unit::convert`.

Some of the conversion factors between the various units are given by exact rational numbers (e.g., `unit::inch = 127/50*unit::cm`), while others are linked by floating-point factors (e.g., `unit::cal = 4.1868*unit::Joule`). Use `float` to approximate exact values by floats. Use `numeric::rationalize` to approximate floats by rational numbers.

Most system functions such as `diff`, `factor`, `normal` etc. accept expressions containing units, treating the units like symbolic identifiers. See “Example 9” on page 1-2171.

The available units are listed below. If the required unit is not available, you can use `unit::newUnit` to add your own unit to the `unit` domain. See “Example 4” on page 1-2167.

Length:

- `am`, `f` (= `Fermi` = `fermi`), `XU` (= `Xu` = `xu` = `XE`), `pm`
- `Ao` (= `Angstroem` = `angstroem` = `Angstrom` = `angstrom`)
- `nm` (= `nanometer` = `nanometers`)
- `My` (= `micron` = `microns` = `micrometer` = `micrometers`)
- `mm` (= `millimeter` = `millimeters`), `cm` (= `centimeter` = `centimeters`)
- `dm` (= `decimeter` = `decimeters`), `m` (= `meter` = `meters`)
- `dam`, `hm`, `km` (= `kilometer` = `kilometers`), `Mm`, `Gm`, `Tm`, `Pm`, `Em`
- `pt` (= `point` = `points`), `inch` (= `inches` = `zoll` = `Zoll`)
- `ft` (= `foot` = `feet`), `ft_US` (= `foot_US` = `feet_US`)
- `yd` (= `yard` = `yards` = `Elle` = `Ellen`), `mile` (= `miles`), `nmile`, `inm` (= `INM`)

- AU (= AE), ly (= lightyear = lightyears = Lj = lj), pc (= parsec)
- ch, fm (= fathom = fathoms), fur (= furlong = furlongs), gg, hand
- li (= link = links), line, mil, rod (= perch = pole), span

Mass:

- ag, fg, pg, ng, mcg (= mcgram = mcgrams = microgram = micrograms)
- mg (= milligram = milligrams), cg, dg, g (= gram = grams), hg
- kg (= kilogram = kilograms), Mg, Gg, Tg, Pg, Eg
- t, kt, Mt, ct (= carat = Kt = Karat = karat)
- oz (= ounce = ounces = Unze = Unzen = unze = unzen)
- lb (= pound = pounds), stone, cwt (= sh_cwt)
- cwt_UK (= long_cwt = gross_cwt), tn (= ton = short_ton), ton_UK
- long_ton (= gross_ton), slug, gr, dr, quarter, cental
- Pfd (= Pfund = pfund), Ztr (= Zentner = zentner)
- dz (= Doppelzentner = doppelzentner)

Time:

- as, fs, ps, ns (= nsec = nanosec = nanosecond = nanoseconds)
- mcsec (= mcsecond = mcseconds = microsec = microsecond = microseconds)
- ms (= msec = millisec = millisecond = milliseconds), cs, ds
- s (= sec = second = seconds = Sekunde = Sekunden), das, hs, ks
- Ms, Gs, Ts, Es, Ps, min (= minute = minutes = Minute = Minuten)
- h (= hour = hours = Stunde = Stunden), d (= day = days = Tag = Tage)
- week (= weeks = Woche = Wochen), month (= months = Monat = Monate)
- year (= years = Jahr = Jahre)

Temperature:

- K (= kelvin = Kelvin), Fahrenheit (= fahrenheit), Celsius (= celsius)
- Rankine (= rankine), Reaumur (= reaumur)

Plain Angle:

- degree (= degrees), rad (= radian)

Solid Angle:

- `sr` (= `steradian`)

Data Size, Storage Capacity:

- `bit` (= `Bit`) `kbit` (= `kBit`), `Mbit` (= `MBit`), `Gbit` (= `GBit`), `Tbit` (= `TBit`)
- `byte` (= `Byte`), `kbyte` (= `kByte`), `Mbyte` (= `MByte`), `Gbyte` (= `GByte`)
- `Tbyte` (= `TByte`)

Data Rate (Bits per Second):

- `bps`

Signal Rate (= Frequency):

- `Bd` (= `Baud` = `baud`)

Velocity:

- `knot`, `knot_UK`, `mach`

Acceleration:

- `Gal`, `gn`

Force:

- `aN`, `fN`, `nN`, `pN`, `mcN`, `mN`, `cN`, `dN`, `N` (= `Newton` = `newton`), `daN`, `hN`, `kN`, `MN`, `GN`
- `TN`, `PN`, `EN`, `p` (= `pond` = `Pond`), `kp` (= `kilopond` = `Kilopond`), `dyn`, `pdl`, `lbf`
- `ozf`, `tonf`

Torque (= Energy):

- `aNm`, `fNm`, `pNm`, `nNm`, `mcNm`, `mNm`, `cNm`, `dNm`, `Nm` (= `Newtonmeter` = `newtonmeter`)
- `daNm`, `hNm`, `kNm`, `MNm`, `GNm`, `TNm`, `PNm`, `ENm`, `kpm`

Angular Momentum:

- `aNms`, `fNms`, `pNms`, `nNms`, `mcNms`, `mNms`, `cNms`, `dNms`
- `Nms` (= `Newtonmetersec` = `newtonmetersec`)

- daNms, hNms, kNms, MNms, GNms, PNms, ENms, TNms

Energy, Work:

- aJ (= aWs), fJ (= fWs), pJ (= pWs), nJ (= nWs), mcJ (= mcWs = microWs)
- mJ (= mJoule = mjoule = mWs), cJ (= cWs), dJ (= dWs)
- J (= Joule = joule = Ws), daJ (= daWs), hJ (= hWs)
- kJ (= kJoule = kjoule = kWs), MJ (= MJoule = Mjoule = MWs)
- GJ (= GWs), TJ (= TWs), PJ (= PWs), EJ (= EWs), Wh, kWh, MWh, GWh
- cal (= Calory = calory), kcal, aeV, feV, peV, neV, mceV, meV, ceV, deV, eV
- daeV, heV, keV, MeV, GeV, TeV, PeV, EeV, PSh, erg, Btu, therm

Power:

- aW, fW, pW, nW, mcW (= mcWatt = mcwatt = microW = microWatt = microwatt)
- mW (= mWatt = mwatt), cW, dW, W (= Watt = watt), daW, hW
- kW (= kWatt = kwatt), MW (= MWatt = Mwatt), GW (= GWatt = Gwatt)
- TW, PW, EW, PS, hp (= bhp)

Frequency:

- aHz, fHz, pHz, nHz, mcHz, mHz, cHz, dHz, Hz (= Hertz = hertz), daHz, hHz
- kHz (= kHertz = khertz), MHz (= MHertz = Mhertz)
- GHz (= GHertz = Ghertz), THz, PHz, EHz

Pressure, Stress:

- aPa, fPa, pPa, nPa, mcPa, mPa, cPa, dPa, Pa (= Pascal = pascal)
- daPa, hPa (= hPascal = hpascal), kPa, MPa, GPa, TPa, PPa, EPa
- mcbar (= mcBar = microbar = microBar), mbar (= mBar), bar (= Bar)
- kbar (= kBar), at (= ata = atu), atm, mmH2O (= mmWS), mH2O (= mWS)
- inH2O, ftH2O, mmHg, mHg, inHg, psi, Torr

Area:

- a (= are = Ar), ac (= acre), b (= barn), ha (= hectare = Hektar)
- ro (= rood), township, circ_mil, circ_inch

Volume:

- `al`, `fl`, `pl`, `nl`, `mcl`, `ml`, `cl`, `dl`, `l` (= `Liter` = `liter` = `Litre` = `litre`), `dal`
- `hl`, `kl`, `Ml`, `Gl`, `Tl`, `Pl`, `El`, `gal` (= `gallon`), `gal_UK`, `barrel`, `bu_UK`, `chaldron`
- `pottle`, `pint_UK`, `pk_UK`, `qt_UK`, `gill`, `gill_UK`, `floz`, `floz_UK`, `fldr`, `fldr_UK`
- `minim`, `minim_UK`, `liq_qt`, `liq_pt`, `dry_bu`, `dry_pk`, `bbl`, `dry_gal`, `dry_qt`, `dry_pt`

European Currency:

- `cent` (= `Cent`), `EUR` (= `EURO` = `Euro`), `ATS`, `DEM` (= `DM`), `BEF`, `ESP`, `FIM`, `FRF`, `LUF`
- `NLG`, `PTE`, `IEP`, `ITL`

Molecular Substance:

- `fmol`, `amol`, `pmol`, `nmol`, `mcmol` (= `mcMol` = `micromol` = `microMol`)
- `mmol` (= `mMol`), `cmol`, `dmol`, `mol` (= `Mol`), `damol`, `hmol`, `kmol` (= `kMol`), `Mmol`
- `Gmol`, `Tmol`, `Pmol`, `Emol`

Electric Current, Amperage:

- `aA`, `fA`, `pA`, `nA` (= `nAmpere` = `nampere`)
- `mcA` (= `microA` = `microAmpere` = `microampere`), `mA` (= `mAmpere` = `mampere`)
- `cA`, `dA`, `A` (= `ampere` = `Ampere`), `daA`, `hA`, `kA` (= `kAmpere` = `kampere`), `MA`, `GA`
- `TA`, `PA`, `EA`, `Bi` (= `Biot` = `biot`), `Gb` (= `Gilbert` = `gilbert`)

Electric Voltage:

- `aV`, `fV`, `pV`, `nV` (= `nanoV` = `nVolt` = `nvolt`)
- `mcV` (= `microV` = `mcVolt` = `mcvolt`), `mV` (= `mVolt` = `mvolt`), `cV`, `dV`
- `V` (= `Volt` = `volt`), `daV`, `hV`
- `kV` (= `kVolt` = `kvolt`)
- `MV` (= `MVolt` = `Mvolt`), `GV` (= `GVolt` = `Gvolt`), `TV`, `PV`, `EV`

Electric Resistance:

- `aOhm` (= `aohm`), `fOhm` (= `fohm`), `pOhm` (= `pohm`), `nOhm` (= `nohm`)
- `mcOhm` (=`mcohm` = `microOhm` = `microohm`)

- mOhm (= mohm = milliOhm = milliohm), dOhm (= dohm), cOhm (= cohm)
- Ohm (= ohm), daOhm (= daohm), hOhm (= hohm), kOhm (= kohm), MOhm (= Mohm)
- GOhm (= Gohm), TOhm (= Tohm), POhm (= Pohm), EOhm (= Eohm)

Electric Charge:

- aC, fC, pC, nC, mcC, mC, cC, dC, C (= Coulomb = coulomb), daC, hC, kC, MC, GC, TC
- PC, EC

Electric Capacity:

- aF, fF, pF (= pFarad = pfarad), nF (= nFarad = nfarad)
- mcF (= mcFarad = mcfarad = microF = microFarad = microfarad)
- mF (= mFarad = mfarad), cF, dF, F (= Farad = farad), daF, hF
- kF (= kFarad = kfarad) , MF, GF, TF, PF, EF

Electric Conductance:

- S (= Siemens = siemens)

Magnetic Inductance:

- H (= Henry = henry)

Magnetic Flux Density, Magnetic Inductivity:

- T (= Tesla = tesla), G (= Gauss = gauss)

Magnetic Flux:

- Wb (= Weber = weber), M (= Maxwell = maxwell)

Magnetic Field Strength:

- Oe (= Oersted = oersted)

Magnetomotive Force (= Electric Current):

- Gb (= Gilbert = gilbert)

Luminous Intensity:

- `fcd`, `acd`, `pcd`, `ncd`, `mccd`, `mcd`, `ccd`, `dcd`, `cd` (= `candela` = `Candela`), `dacd`, `hcd`
- `kcd`, `Mcd`, `Gcd`, `Tcd`, `Pcd`, `Ecd`, `HK`, `IK`

Luminance:

- `sb` (= `stilb`), `asb` (= `apostilb`)

Luminous Flux:

- `lm` (= `lumen`)

Illuminance:

- `lx` (= `lux`), `ph` (= `phot`), `nx`

Radiation:

- `langley`

Radioactivity:

- `aBq`, `fBq`, `pBq`, `nBq`, `mcBq`, `mBq`, `cBq`, `dBq`, `Bq` (= `Becquerel` = `becquerel`), `daBq`
- `hBq`, `kBq`, `MBq`, `GBq`, `TBq`, `PBq`, `EBq`, `Ci` (= `Curie`)

Equivalent Dosage:

- `aSv`, `fSv`, `pSv`, `nSv`, `mcSv`, `mSv`, `cSv`, `dSv`
- `Sv` (= `Sievert` = `sievert`), `daSv`, `hSv`, `kSv`, `MSv`, `GSv`, `TSv`, `PSv`, `ESv`
- `arem`, `frem`, `prem`, `nrem`, `mcrem`, `mrem`, `crem`, `drem`
- `rem` (= `Rem`), `darem`, `hrem`, `krem`, `Mrem`, `Grem`, `Trem`, `Prem`, `Erem`

Absorbed Dosage:

- `aGy`,`fGy`,`pGy`,`nGy`,`mcGy`,`mGy`,`cGy`,`dGy`, `Gy` (= `Gray` = `gray`), `daGy`,`hGy`,`kGy`, `MGy`, `GGy`
- `TGy`,`PGy`,`EGy`, `rd`

Ionising Dosage:

- `R` (= `Roentgen`)

Lens Power:

- `dpt` (= `diopter` = `dioptre`)

Dynamic Viscosity:

- `P` (= `Poise`)

Kinematic Viscosity:

- `St` (= `Stokes`)

Mass Per Length:

- `tex`, `den` (= `denier`)

Examples

Example 1

Units not convertible are left alone by `unit::convert`:

```
unit::convert(1.23*unit::kg*unit::inch^2/unit::mm, unit::cm)
```

$$79.35468\ \text{cm kg}$$

```
unit::convert(unit::km/unit::hour, unit::m/unit::sec)
```

$$\frac{5\ \frac{m}{sec}}{18}$$

Example 2

`unit::simplify` favors kg over pounds:

```
unit::simplify(1.23*unit::kg^2/unit::pound*unit::inch^2/unit::mm)
```

$$68.87681995\ \text{inch kg}$$

Example 3

Specify units for unknown variables so that the variables you solve for are dimensionless. If you apply units inconsistently, you get incorrect results.

Demonstrate this by omitting the unit for an unknown variable in an equation and solving the equation. The `solve` function cannot solve the equation.

```
L := 1*unit::m:
y := 0.1*unit::m:
f := y - x*cos(L/x):
solve(f,x)
```

$$\varnothing$$

Solve the equation for the unknown variable by specifying the correct unit for the variable.

```
f := y - x*unit::m*cos(L/(x*unit::m)):
solve(f,x)
```

$$\{0.1898572041\}$$

Example 4

We add new velocity units to the `unit` domain:

```
unit::newUnit(SpeedOfLight = 300000*unit::km/unit::sec)
```

$$\text{SpeedOfLight}$$

Now, the unit `unit::SpeedOfLight` exists and can be used like any other unit in the `unit` domain. We use it to define yet another velocity unit:

```
unit::newUnit(Warp9 = 1.516*unit::SpeedOfLight)
```

$$\text{Warp9}$$

We convert the velocity of 123.4 miles per hour into the new speed units:

```
unit::convert(123.4*unit::mile/unit::hour, unit::SpeedOfLight)
```

$$0.0000001838824533\ \text{SpeedOfLight}$$

```
unit::convert(123.4*unit::mile/unit::hour, unit::Warp9)
```

$$0.0000001212944943\ \text{Warp9}$$

We verify the new units:

```
unit::convert(unit::SpeedOfLight, unit::km/unit::sec)
```

$$300000\ \frac{\text{km}}{\text{sec}}$$

```
unit::convert(unit::Warp9, unit::SpeedOfLight)
```

$$1.516\ \text{SpeedOfLight}$$

```
unit::convert(unit::Warp9, unit::km/unit::sec)
```

$$454800.0\ \frac{\text{km}}{\text{sec}}$$

Example 5

We create a symbolic expression involving different units of type 'length':

```
27*unit::cm + 30*unit::mm
```

$$27\ \text{cm} + 30\ \text{mm}$$

This expression is not simplified automatically. We apply `simplify`:

```
simplify(%)
```

$$30\ \text{cm}$$

We convert this length to inch:

```
unit::convert(%, unit::inch)
```

$$\frac{1500\ \text{inch}}{127}$$

```
float(%)
```

$$11.81102362\ \text{inch}$$

Here is another example for simplification and conversion:

```
1234*unit::g + 1.234*unit::kg*unit::m^2/unit::inch^2
```

$$1234\ g + 1.234\ \frac{\text{kg}\ m^2}{\text{inch}^2}$$

```
simplify(%)
```

$$1913937.825\ g$$

```
unit::convert(%, unit::ounce)
```

$$67512.17003\ \text{ounce}$$

The target unit in `unit::convert` may be an expression:

```
unit::convert(unit::pound*unit::km/unit::hour,
              unit::kg*unit::m/unit::s)
```

$$\frac{45359237\ \frac{\text{kg}\ m}{s}}{360000000}$$

Example 6

The probably most interesting method of the `unit` domain is the conversion routine `unit::convert`. Given any expression involving units, you can specify a target unit which is to be used to express the units:

```
unit::convert(unit::ounce, unit::kilogram)
```

$$\frac{45359237\ \text{kilogram}}{1600000000}$$

The target unit needs not be of the same physical type as the expression that is to be rewritten. In the following example, we wish to rewrite a torque (given in "Newton meters") in terms of units involving the power unit "Watt." Note that a torque is an energy, i.e., "power" multiplied by "time":

```
unit::convert(1.23*unit::Nm, unit::W)
```

$$1.23\ W\ s$$

We wish to rewrite "Newton meters" in terms of units involving "centimeters":

```
unit::convert(unit::Nm, unit::cm)
```

$$10000\ \frac{\text{cm}^2\ \text{kg}}{s^2}$$

The target unit may be a composite expression. We wish to rewrite "Newton meters" in terms of "grams," "centimeters," and "milliseconds":

```
unit::convert(unit::Nm, unit::g*unit::cm^2/unit::msec^2)
```

$$10\ \frac{\text{cm}^2\ g}{\text{msec}^2}$$

Example 7

The `info` command provides information on units. In particular, it lists alternative names that can be used in MuPAD:

```
info(unit::nm)
```

```
nanometer: a length unit

Alternative names: unit::nanom, unit::nanometer, unit::nanometers, unit::nm
```

```
info(unit::Joule)
```

```
joule: a unit of energy

Alternative names: unit::J, unit::Joule, unit::Newtonmeter, unit::Nm, unit::Ws, unit::
```

Example 8

We use `unit::convert2SIunits` to convert a mass expressed in non-metric units to SI base units:

```
mass := 2*unit::cal*unit::msec^2/unit::inch^2 - 45*unit::carat
```

$$2\,\frac{\mathrm{cal}\,\mathrm{msec}^2}{\mathrm{inch}^2} - 45\ \mathrm{carat}$$

```
unit::convert2SIunits(mass)
```

$$0.003979105958\ \mathrm{kg}$$

```
delete mass:
```

Example 9

Most system functions such as `diff`, `factor`, `normal` etc. treat units like ordinary symbolic identifiers:

```
diff(x/unit::m*exp(-x^2/unit::m^2), x)
```

$$e^{-x^2\,\frac{1}{m^2}}\,\frac{1}{m} - \left(2\,x^2\,e^{-x^2\,\frac{1}{m^2}}\right)\,\frac{1}{m^3}$$

```
factor(%)
```

$$-\frac{2\,x^2 - m^2}{e^{\frac{x^2}{m^2}}}\,\frac{1}{m^3}$$

```
normal((4*unit::m^2 - a^2*unit::m^2)/(2*unit::m - a*unit::m))
```

$$2\,m + a\,m$$

Parameters

`nam`

The name of the physical unit, see the list below. Some units such as `unit::mm` and `unit::millimeter` represent the same physical unit.

Methods

`convert` — Convert an expression to other units

`convert(x, targetunit)` converts all units in the arithmetical expression x to multiples of the targetunit if possible. The targetunit may be one of the unit objects of type 'length,' 'mass' etc. It may also be an arithmetical expression such as `unit::km/unit::sec`. In this case, x is rewritten in terms of the units found in targetunit.

`convert2SIunits` — Rewrite to SI units

`convert2SIunits(x)` rewrites all units in the arithmetical expression x in terms of corresponding SI base units.

`simplify` — Combine like units in an expression

`unit:: simplify(x)` converts all units in the arithmetical expression x to some basic unit found in x, i.e., all length units are expressed by the same length unit, all mass units are expressed by the same mass unit, all time units are expressed by the same time unit etc.

`newUnit` — Define a new unit

```
newUnit(newname = expression)
```

`unit::newUnit(newname = f*oldunit)` creates a new unit that may be addressed by `unit::newname`. Its name newname must be an identifier. It is declared as the `f`-fold of

some unit `oldunit` that must be an expression such as `unit::mm/unit::sec` involving units provided by the unit domain. The conversion factor f must be an arithmetical expression (typically, a numerical conversion factor).

`display` — Format for output

`display(x)` formats the screen output of the arithmetical expression x such that the units appear as a separate factor at the end of each term.

`findUnits` — Find all units in expression

`findUnits(x)` returns a set of all units found in the arithmetical expression x.

`Celsius2Fahrenheit` — Convert degrees Celsius to degrees Fahrenheit

`unit::Celsius2Fahrenheit(x)` converts a numerical value x representing a temperature in degrees Celsius into a numerical value representing this temperature in degrees Fahrenheit.

`Celsius2Kelvin` — Convert degrees Celsius to Kelvin

`unit::Celsius2Kelvin(x)` converts a numerical value x representing a temperature in degrees Celsius into a numerical value representing this temperature in degrees Kelvin.

`Celsius2Rankine` — Convert degrees Celsius to degrees Rankine

`unit::Celsius2Rankine(x)` converts a numerical value x representing a temperature in degrees Celsius into a numerical value representing this temperature in degrees Rankine.

`Celsius2Reaumur` — Convert degrees Celsius to degrees Reaumur

`unit::Celsius2Reaumur(x)` converts a numerical value x representing a temperature in degrees Celsius into a numerical value representing this temperature in degrees Reaumur.

`Fahrenheit2Celsius` — Convert degrees Fahrenheit to degrees Celsius

`unit::Fahrenheit2Celsius(x)` converts a numerical value x representing a temperature in degrees Fahrenheit into a numerical value representing this temperature in degrees Celsius.

`Fahrenheit2Kelvin` — Convert degrees Fahrenheit to Kelvin

`unit::Fahrenheit2Kelvin(x)` converts a numerical value x representing a temperature in degrees Fahrenheit into a numerical value representing this temperature in degrees Kelvin.

`Fahrenheit2Rankine` — Convert degrees Fahrenheit to degrees Rankine

`unit::Fahrenheit2Rankine(x)` converts a numerical value x representing a temperature in degrees Fahrenheit into a numerical value representing this temperature in degrees Rankine.

`Fahrenheit2Reaumur` — Convert degrees Fahrenheit to degrees Reaumur

`unit::Fahrenheit2Reaumur(x)` converts a numerical value x representing a temperature in degrees Fahrenheit into a numerical value representing this temperature in degrees Reaumur.

`Kelvin2Fahrenheit` — Convert Kelvin to degrees Fahrenheit

`unit::Kelvin2Fahrenheit(x)` converts a numerical value x representing a temperature in degrees Kelvin into a numerical value representing this temperature in degrees Fahrenheit.

`Kelvin2Celsius` — Convert Kelvin to degrees Celsius

`unit::Kelvin2Celsius(x)` converts a numerical value x representing a temperature in degrees Kelvin into a numerical value representing this temperature in degrees Celsius.

`Kelvin2Rankine` — Convert Kelvin to degrees Rankine

`unit::Kelvin2Rankine(x)` converts a numerical value x representing a temperature in degrees Kelvin into a numerical value representing this temperature in degrees Rankine.

`Kelvin2Reaumur` — Convert Kelvin to degrees Reaumur

`unit::Kelvin2Reaumur(x)` converts a numerical value x representing a temperature in degrees Kelvin into a numerical value representing this temperature in degrees Reaumur.

`Rankine2Fahrenheit` — Convert degrees Rankine to degrees Fahrenheit

`unit::Rankine2Fahrenheit(x)` converts a numerical value x representing a temperature in degrees Rankine into a numerical value representing this temperature in degrees Fahrenheit.

`Rankine2Kelvin` — Convert degrees Rankine to Kelvin

`unit::Rankine2Kelvin(x)` converts a numerical value x representing a temperature in degrees Rankine into a numerical value representing this temperature in degrees Kelvin.

`Rankine2Celsius` — Convert degrees Rankine to degrees Celsius

`unit::Rankine2Celsius(x)` converts a numerical value x representing a temperature in degrees Rankine into a numerical value representing this temperature in degrees Celsius.

`Rankine2Reaumur` — Convert degrees Rankine to degrees Reaumur

`unit::Rankine2Reaumur(x)` converts a numerical value x representing a temperature in degrees Rankine into a numerical value representing this temperature in degrees Reaumur.

`Reaumur2Fahrenheit` — Convert degrees Reaumur to degrees Fahrenheit

`unit::Reaumur2Fahrenheit(x)` converts a numerical value x representing a temperature in degrees Reaumur into a numerical value representing this temperature in degrees Fahrenheit.

`Reaumur2Kelvin` — Convert degrees Reaumur to Kelvin

`unit::Reaumur2Kelvin(x)` converts a numerical value x representing a temperature in degrees Reaumur into a numerical value representing this temperature in degrees Kelvin.

`Reaumur2Rankine` — Convert degrees Reaumur to degrees Rankine

`unit::Reaumur2Rankine(x)` converts a numerical value x representing a temperature in degrees Reaumur into a numerical value representing this temperature in degrees Rankine.

`Reaumur2Celsius` — Convert degrees Reaumur to degrees Celsius

`unit::Reaumur2Celsius(x)` converts a numerical value x representing a temperature in degrees Reaumur into a numerical value representing this temperature in degrees Celsius.

universe

Set-theoretical universe

Description

`universe` represents the set-theoretical universe of all objects.

`universe` is the only element of the domain `stdlib::Universe`.

The standard set operations such as union, intersection and subtraction can be used with `universe`.

Examples

Example 1

We show some basic set operations involving `universe`:

```
universe union {a}
```

$$\text{universe}$$

```
universe intersect {a}
```

$$\{a\}$$

```
{a} minus universe
```

$$\varnothing$$

See Also

MuPAD Domains
DOM_SET

MuPAD Functions
`intersect | minus | union`

unprotect

Remove protection of identifiers

Syntax

```
unprotect(x)
```

Description

`unprotect(x)` removes any write protection of the identifier `x`.

`unprotect(x)` is equivalent to `protect(x, ProtectLevelNone)`.

`unprotect` does not evaluate its argument. Cf. “Example 2” on page 1-2179.

Examples

Example 1

`unprotect` allows to assign values to system functions:

```
unprotect(sign): sign(x) := 1
```

```
    1
```

However, we strongly advise not to change identifiers protected by the system. We undo the previous assignment:

```
delete sign(x): protect(sign, ProtectLevelError):
```

Example 2

`unprotect` does not evaluate its argument. Here the identifier `x` is unprotected and not its value `y`:

```
x := y:  protect(y): unprotect(x): y := 1
```

```
Warning: The protected variable 'y' is overwritten. [_assign]

     1

 Warning: Protected variable
'y' overwritten. [_assign]

     1

unprotect(y): delete x, y:
```

Example 3

The identifier a is protected with various levels. `unprotect` returns the previous protection level:

```
protect(a):
unprotect(a)

     ProtectLevelWarning

protect(a, ProtectLevelError):
unprotect(a)

     ProtectLevelError
```

At this place, a is not protected:

```
unprotect(a)

     ProtectLevelNone
```

Parameters

x

An identifier

Return Values

Previous protection level of `x`: either `ProtectLevelError` or `ProtectLevelWarning` or `ProtectLevelNone` (see `protect`).

See Also

MuPAD Functions
`protect`

use

Use library functions by a short name

Syntax

```
use(L, <Alias>, f1, f2, …)
use(L, <Alias>)
```

Description

`use(L, f)` 'exports' the public function `L::f` of the library `L` to the global namespace such that it can be accessed as `f` without the prefix `L`.

`use(L)` exports all public functions of the library `L`.

A library contains *public* functions which may be called by the user. The collection of these functions forms the *interface* of the library. (There may be other, private, functions, too, which are not intended to be called by the user directly, and are not documented.) The standard way of accessing the public function `f` from the library `L` is via `L::f`. When the function `f` is *exported*, it can be accessed more briefly as `f`. Technically, exporting means that the global identifier `f` is assigned the value `L::f`. Alternatively, when the option `Alias` is used, an `alias` is created.

Unexporting the library function `f` means that the value of the global identifier `f` is deleted. Afterwards, the library function is available only as `L::f`.

`use(L, f1, f2, ...)` exports the given functions `f1, f2, ...` of `L`. However, if one of the identifiers already has a value, the corresponding function is not exported. A warning is printed instead. An error is returned if one of the identifiers is not the name of a public library function.

`use(L)` exports all public functions of `L`.

A function that is already exported will not be exported twice.

`use` evaluates its first argument `L`, but it does not evaluate the remaining arguments `f1, f2, ...`, if any.

The function `info` displays the interface functions and the exported functions of a library.

Some libraries have functions that are always exported. These functions cannot be unexported. The function `append` from the library `listlib` is such an example.

Environment Interactions

When a function is exported, it is assigned to the corresponding global identifier. When it is unexported, the corresponding identifier is deleted.

Examples

Example 1

Export the function `invphi` of the library numlib, and then undo the export:

```
numlib::invphi(4!)
```

```
      [35, 39, 45, 52, 56, 70, 72, 78, 84, 90]
```

```
use(numlib, invphi):
invphi(4!)
```

```
      [35, 39, 45, 52, 56, 70, 72, 78, 84, 90]
```

```
unuse(numlib, invphi):
invphi(4!)
```

```
      invphi(24)
```

Export all functions of the library numlib:

```
use(numlib):
```

```
invphi(100)
```

```
Warning: Identifier 'divisors' already has a value. It is not exported. [use]
```

```
Warning: Identifier 'contfrac' already has a value. It is not exported. [use]
```

$$[101, 125, 202, 250]$$

Here, `use` issues warnings because `contfrac` and `divisors` are already available as global functions. For example, there is the global `contfrac` function that uses `numlib::contfrac` for numerical arguments. Undo the export of the `numlib` functions.

```
unuse(numlib):
invphi(100)
```

$$\text{invphi}(100)$$

Example 2

`use` issues a warning if a function cannot be exported since the corresponding identifier already has a value:

```
invphi := 17:
use(numlib, invphi)
```

```
Warning: Identifier 'invphi' already has a value. It is not exported. [use]
```

Parameters

`L`

The library: a domain

`f1, f2, …`

Public functions of `L`: identifiers

Options

`Alias`

Use `alias(f = L::f)` to create an alias `f` for `L::f` rather than exporting `L::f` by the assignment `f:= L::f`.

Return Values

Void object `null()` of type `DOM_NULL`.

Algorithms

The names of the public functions of a library `L` are stored in the set `L::interface`. This set is used by the function `info` and for exporting.

The names of functions exported from a library `L` are stored in the set `L::_exported`.

See Also

MuPAD Functions

`:=` | `alias` | `delete` | `info` | `unuse`

unuse

Undo the use command

Syntax

```
unuse(L, f1, f2, …)
unuse(L)
```

Description

`unuse(L, f)` undoes the export of the public function `L::f` of the library `L` such that it is no longer available as `f`.

`unuse(L)` undoes the export of all previously exported public functions of the library `L`.

A library contains *public* functions which may be called by the user. The collection of these functions forms the *interface* of the library. (There may be other, private, functions, too, which are not intended to be called by the user directly, and are not documented.) The standard way of accessing the public function `f` from the library `L` is via `L::f`. When the function `f` is *exported*, it can be accessed more briefly as `f`. Technically, exporting means that the global identifier `f` is assigned the value `L::f`. Alternatively, when the option `Alias` is used, an `alias` is created.

Unexporting the library function `f` means that the value of the global identifier `f` is deleted. Afterwards, the library function is available only as `L::f`.

`unuse(L, f1, f2, ...)` unexports all given functions of `L`. Note that `unuse` does not evaluate the identifiers. Thus, it is not necessary to use `hold` to protect them from being evaluated.

`unuse(L)` unexports all public functions of the library `L`.

`unuse` evaluates its first argument `L`, but it does not evaluate the remaining arguments `f1, f2, ...`, if any.

The function `info` displays the interface functions and the exported functions of a library.

Some libraries have functions that are always exported. These functions cannot be unexported. The function `append` from the library `listlib` is such an example.

Environment Interactions

When a function is exported, it is assigned to the corresponding global identifier. When it is unexported, the corresponding identifier is deleted.

Examples

Example 1

Export the function `invphi` of the library numlib, and then undo the export:

```
numlib::invphi(4!)
```

```
[35, 39, 45, 52, 56, 70, 72, 78, 84, 90]
```

```
use(numlib, invphi):
invphi(4!)
```

```
[35, 39, 45, 52, 56, 70, 72, 78, 84, 90]
```

```
unuse(numlib, invphi):
invphi(4!)
```

```
invphi(24)
```

Export all functions of the library numlib:

```
use(numlib):
invphi(100)
```

```
Warning: Identifier 'divisors' already has a value. It is not exported. [use]
```

```
Warning: Identifier 'contfrac' already has a value. It is not exported. [use]
```

```
[101, 125, 202, 250]
```

Here, `use` issues warnings because `contfrac` and `divisors` are already available as global functions. For example, there is the global `contfrac` function that uses `numlib::contfrac` for numerical arguments. Undo the export of the `numlib` functions.

```
unuse(numlib):
invphi(100)
```

```
invphi(100)
```

Example 2

`use` issues a warning if a function cannot be exported since the corresponding identifier already has a value:

```
invphi := 17:
use(numlib, invphi)
```

```
Warning: Identifier 'invphi' already has a value. It is not exported. [use]
```

Parameters

`L`

The library: a domain

`f1, f2, …`

Public functions of `L`: identifiers

Options

`Alias`

Use `alias(f = L::f)` to create an alias `f` for `L::f` rather than exporting `L::f` by the assignment `f:= L::f`.

Return Values

Void object `null()` of type `DOM_NULL`.

Algorithms

The names of the public functions of a library `L` are stored in the set `L::interface`. This set is used by the function `info` and for exporting.

The names of functions exported from a library `L` are stored in the set `L::_exported`.

See Also

MuPAD Functions

`:=` | `alias` | `delete` | `info` | `use`

More About

- "Use the MuPAD Libraries"

val

Value of an object

Syntax

```
val(object)
```

Description

`val(object)` replaces every identifier in `object` by its value.

`val` does not perform any simplification of the result.

If the result of `val` is a set, duplicate elements are removed from that set.

`val` does not work recursively, i.e., if the value of an identifier in turn contains identifiers, then these are not replaced by their values. See “Example 3” on page 1-2191.

`val` does not flatten its argument. Hence, an expression sequence is accepted as argument. Cf. “Example 2” on page 1-2191.

Examples

Example 1

`val` replaces identifiers by their values, but does not call arithmetical functions such as `_plus`:

```
a := 0: val(a*b + 4 + 0)
```

$$0\,b + 4 + 0$$

Duplicate elements in sets are removed:

```
a := b: val({a, b, a*0})
```

$\{b, 0\ b\}$

```
delete a:
```

Example 2

`val` does not flatten its argument, nor does it remove void objects of type `DOM_NULL`:

```
a := null(): val((a, null()))
```

```
 null(), null()
```

However, it is not legal to pass several arguments:

```
val(a, null())
```

```
Error: The number of arguments is incorrect. [val]
```

```
delete a:
```

Example 3

`val` does not recursively substitute values for the identifiers:

```
delete a, b: a := b: b := c: val(a)
```

b

Parameters

`object`

Any MuPAD object

Return Values

“evaluated” object.

See Also

MuPAD Functions
`eval` | `hold` | `LEVEL` | `level` | `MAXLEVEL`

vectorPotential

Vector potential of a three-dimensional vector field

Syntax

```
vectorPotential(j, [x1, x2, x3], <Test>)
```

Description

`vectorPotential(j, x)` returns the vector potential of the vector field $\vec{j}(\vec{x})$ with respect to $\vec{x}$. This is a vector field $\vec{v}$ with $\mathrm{curl}_{\vec{v}}(\vec{x}) = \vec{j}$.

The vector potential of a vector function `j` exists if and only if the divergence of `j` is zero. It is uniquely determined.

If the vector potential of `j` does not exist, then `vectorPotential` returns `FALSE`.

If `j` is a vector then the component ring of `j` must be a field (i.e., a domain of category `Cat::Field`) for which definite integration can be performed.

If `j` is given as a list of three arithmetical expressions, then `vectorPotential` returns a vector of the domain `Dom::Matrix()`.

Examples

Example 1

We check if the vector function $\vec{j}(x, y, z) = \left(x^2\, y,\ -\frac{y^2\, x}{2},\ -x\, y\, z\right)$ has a vector potential:

```
delete x, y, z:
vectorPotential(
  [x^2*y, -1/2*y^2*x, -x*y*z], [x, y, z], Test
)
```

```
TRUE
```

The answer is yes, so let us compute the vector potential of $\vec{j}$:

```
vectorPotential(
  [x^2*y, -1/2*y^2*x, -x*y*z], [x, y, z]
)
```

$$\begin{pmatrix} -\frac{x\,y^2\,z}{2} \\ -x^2\,y\,z \\ 0 \end{pmatrix}$$

We check the result:

```
curl(%, [x, y, z])
```

$$\begin{pmatrix} x^2\,y \\ -\frac{x\,y^2}{2} \\ -x\,y\,z \end{pmatrix}$$

Example 2

The vector function $\vec{j} = (x^2, 2\,y, z)$ does not have a vector potential:

```
vectorPotential([x^2, 2*y, z], [x, y, z])
```

```
FALSE
```

Parameters

j

A list of three arithmetical expressions, or a 3-dimensional vector (i.e., a 3×1 or 1 ×3 matrix of a domain of category `Cat::Matrix`)

x_1, x_2, x_3

(indexed) identifiers

Options

`Test`

Check whether the vector field `j` has a vector potential and return `TRUE` or `FALSE`, respectively.

Return Values

Vector with three components, i.e., an 3×1 or $1 \times n$ matrix of a domain of category `Cat::Matrix`, or a boolean value.

See Also

MuPAD Functions

`curl` | `divergence` | `gradient` | `laplacian` | `potential`

version

Version number of the MuPAD library

Syntax

```
version()
```

Description

`version()` returns the version number of the installed MuPAD library.

The call `Pref::kernel()` returns the version number of the installed MuPAD kernel.

The version numbers of the kernel and the library may differ: `version` refers to the library, whereas the call `Pref::kernel()` returns the version number of the kernel.

Examples

Example 1

The version of this MuPAD library is:

```
version()
```

```
[6, 2, 0]
```

Return Values

Version number: a list of three nonnegative integers.

See Also

MuPAD Functions

`buildnumber` | `Pref::kernel`

warning

Print a warning message

Syntax

```
warning(message)
```

Description

`warning(message)` prints the message with the prefix "`Warning:`".

`warning` may be used to print information about potential problems in an algorithm. E.g., it is used in `limit` to provide hints.

Examples

Example 1

A warning:

```
warning("You should not do this!"):
```

```
Warning: You should not do this!
```

Example 2

This example shows a simple procedure which divides two numbers. If the second argument is omitted, a warning is printed and the computation continues:

```
mydivide := proc(x, y)
begin
  if args(0) < 2 then
    warning("Denominator not given, using 1.");
    y := 1;
  end_if:
```

```
  x/y
end_proc:
mydivide(10)

Warning: Denominator not given, using 1. [mydivide]

     10
```

Parameters

`message`

A character string

Return Values

Void object of type `DOM_NULL`.

See Also

MuPAD Functions

`error` | `print`

whittakerM

The Whittaker M function

Syntax

```
whittakerM(a, b, z)
```

Description

The `whittakerM` function $M_{a,\,b}(z)$ is related to the confluent hypergeometric function ${}_pF_q(a,\,b,\,z) = \Phi(a,\,b,\,z)$ by the formula:

$$M_{a,\,b}(z) = \mathrm{e}^{-\frac{z}{2}}\, z^{\frac{1}{2}+b}\, \Phi\left(\frac{1}{2}+b-a,\,1+2\,b,\,z\right),$$

The WhittakerM function is defined for complex arguments a, b, and z.

For most of the values of the parameters, an unevaluated function call is returned. Cf. “Example 1” on page 1-2200.

Explicit symbolic expressions are returned for some particular values of the parameters. Cf. “Example 2” on page 1-2200.

Note: MuPAD defines ${}_1F_1(a,\,a,\,z) = \mathrm{e}^z$ for all complex numbers a. As a consequence, the MuPAD `whittakerM` differs from the corresponding function in M. Abramowitz and I. A. Stegun, “Handbook of Mathematical Functions” when $\frac{1}{2}+b-a$ and $1 + 2\,b$ are negative integers and $\frac{1}{2}+b-a \geq 1+2\,b$. Some of the formulas in Chapter 13 of the “Handbook of Mathematical Functions” do not hold for the MuPAD `whittakerM` with such arguments. Cf. “Example 4” on page 1-2201.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Unevaluated calls are returned for exact or symbolic arguments:

```
whittakerM(a, b, x), whittakerW(-3/2, 1/2, 1)
```

$$\mathrm{M}_{a,b}(x),\ \mathrm{W}_{-\frac{3}{2},\frac{1}{2}}(1)$$

Floating point values are returned for floating-point arguments:

```
whittakerM(-2, 0.5, -50), whittakerW(-3/2, 1/2, 0.0)
```

$$0.00000001666553264,\ 0.7522527781$$

Example 2

Explicit expressions are returned for some specific values of the parameters:

```
whittakerM(0, b, x), whittakerW(0, b, x), whittakerW(-3/2, 1/2, 0),
whittakerM(-3/2, 0 ,x), whittakerW(a, -a + 1/2, x)
```

$$4^b\sqrt{x}\,\Gamma(b+1)\,\mathrm{I}_b\left(\frac{x}{2}\right),\ \frac{x^{b+\frac{1}{2}}\,\mathrm{K}_b\left(\frac{x}{2}\right)}{\sqrt{\pi}\,x^b},\ \frac{4}{3\sqrt{\pi}},\ \sqrt{x}\,\mathrm{e}^{\frac{x}{2}}\,(x+1),\ x^{1-a}\,x^{2a-1}\,\mathrm{e}^{-\frac{x}{2}}$$

Example 3

The functions `diff`, `float`, `limit`, and `series` handle expressions involving the Whittaker functions

```
diff(whittakerM(a,b,z),z), float(whittakerW(-3/2,1/2,0))
```

$$\frac{M_{a+1,b}(z)\left(a+b+\frac{1}{2}\right)}{z}-\left(\frac{a}{z}-\frac{1}{2}\right)M_{a,b}(z),\ 0.7522527781$$

```
series(whittakerW(-3/2,1/2,x),x,3)
```

$$\frac{4}{3\sqrt{\pi}}+x\left(\frac{2\sigma_1}{\sqrt{\pi}}-\frac{2}{3\sqrt{\pi}}\right)+x^2\left(\frac{2\left(\frac{5\,\mathrm{EULER}}{4}-\frac{5\ln(2)}{2}+\frac{5\ln(x)}{4}+\frac{17}{24}\right)}{\sqrt{\pi}}-\frac{\sigma_1}{\sqrt{\pi}}+\frac{1}{6\sqrt{\pi}}\right)+O(x^3)$$

where

$$\sigma_1=\mathrm{EULER}-2\ln(2)+\ln(x)+\frac{5}{3}$$

Example 4

For some values of the input parameters, recurrence and differential relations in Chapter 13 of M. Abramowitz and I. A. Stegun, "Handbook of Mathematical Functions" may not hold for the MuPAD `whittakerM` functions. For example, Formula 13.4.32

$$z\frac{\partial}{\partial z}M_{a,b}(z)=\left(\frac{z}{2}-a\right)M_{a,b}(z)+\left(a+b+\frac{1}{2}\right)M_{a+1,b}(z)$$

is not satisfied for $a = 0$ and $b=-\frac{3}{2}$:

```
expand(x*diff(whittakerM(0, -3/2, x), x) <>
x/2*whittakerM(0, -3/2, x) - whittakerM(1, -3/2, x))
```

$$-\frac{e^{-\frac{x}{2}}}{2}-\frac{x\,e^{-\frac{x}{2}}}{4}-\frac{e^{-\frac{x}{2}}}{x}\neq\frac{e^{-\frac{x}{2}}}{2}+\frac{x\,e^{-\frac{x}{2}}}{4}-\frac{e^{\frac{x}{2}}}{x}$$

Parameters

a, b, z

arithmetical expressions

Return Values

Arithmetical expression.

Overloaded By

z

Algorithms

$M_{a,\,b}(z)$ and $W_{a,\,b}(z)$ satisfy Whittaker's differential equation:

$$\frac{\partial^2}{\partial z^2}\,y+\left(-\frac{1}{4}+\frac{a}{z}+\frac{\frac{1}{4}-b^2}{z^2}\right)y=0$$

.

See Also

MuPAD Functions

`hypergeom` | `kummerU` | `whittakerW`

whittakerW

The Whittaker W function

Syntax

```
whittakerW(a, b, z)
```

Description

The `whittakerW` function $W_{a,b}(z)$ is related to the confluent hypergeometric function $kummerU(a, b, z) = U(a, b, z)$ by the formula:

$$W_{a,b}(z) = e^{-\frac{z}{2}} z^{\frac{1}{2}+b} U\left(\frac{1}{2}+b-a, 1+2b, z\right).$$

The WhittakerW function is defined for complex arguments a, b, and z.

For most of the values of the parameters, an unevaluated function call is returned. Cf. "Example 1" on page 1-2203.

Explicit symbolic expressions are returned for some particular values of the parameters. Cf. "Example 2" on page 1-2204.

Environment Interactions

When called with floating-point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Unevaluated calls are returned for exact or symbolic arguments:

```
whittakerM(a, b, x), whittakerW(-3/2, 1/2, 1)
```

$$M_{a,b}(x),\ W_{-\frac{3}{2},\frac{1}{2}}(1)$$

Floating point values are returned for floating-point arguments:

```
whittakerM(-2, 0.5, -50), whittakerW(-3/2, 1/2, 0.0)
```

$$0.00000001666553264,\ 0.7522527781$$

Example 2

Explicit expressions are returned for some specific values of the parameters:

```
whittakerM(0, b, x), whittakerW(0, b, x), whittakerW(-3/2, 1/2, 0),
whittakerM(-3/2, 0 ,x), whittakerW(a, -a + 1/2, x)
```

$$4^b \sqrt{x}\,\Gamma(b+1)\,I_b\left(\frac{x}{2}\right),\ \frac{x^{b+\frac{1}{2}}\,K_b\left(\frac{x}{2}\right)}{\sqrt{\pi}\,x^b},\ \frac{4}{3\sqrt{\pi}},\ \sqrt{x}\,e^{\frac{x}{2}}\,(x+1),\ x^{1-a}\,x^{2a-1}\,e^{-\frac{x}{2}}$$

Example 3

The functions `diff`, `float`, `limit`, and `series` handle expressions involving the Whittaker functions

```
diff(whittakerM(a,b,z),z), float(whittakerW(-3/2,1/2,0))
```

$$\frac{M_{a+1,b}(z)\left(a+b+\frac{1}{2}\right)}{z}-\left(\frac{a}{z}-\frac{1}{2}\right)M_{a,b}(z),\ 0.7522527781$$

```
series(whittakerW(-3/2,1/2,x),x,3)
```

$$\frac{4}{3\sqrt{\pi}} + x\left(\frac{2\sigma_1}{\sqrt{\pi}} - \frac{2}{3\sqrt{\pi}}\right) + x^2\left(\frac{2\left(\frac{5\,\mathrm{EULER}}{4} - \frac{5\ln(2)}{2} + \frac{5\ln(x)}{4} + \frac{17}{24}\right)}{\sqrt{\pi}} - \frac{\sigma_1}{\sqrt{\pi}} + \frac{1}{6\sqrt{\pi}}\right) + O(x^3)$$

where

$$\sigma_1 = \mathrm{EULER} - 2\ln(2) + \ln(x) + \frac{5}{3}$$

Example 4

For some values of the input parameters, recurrence and differential relations in Chapter 13 of M. Abramowitz and I. A. Stegun, "Handbook of Mathematical Functions" may not hold for the MuPAD `whittakerM` functions. For example, Formula 13.4.32

$$z\frac{\partial}{\partial z}M_{a,b}(z) = \left(\frac{z}{2} - a\right)M_{a,b}(z) + \left(a + b + \frac{1}{2}\right)M_{a+1,b}(z)$$

is not satisfied for $a = 0$ and $b = -\frac{3}{2}$:

```
expand(x*diff(whittakerM(0, -3/2, x), x) <>
x/2*whittakerM(0, -3/2, x) - whittakerM(1, -3/2, x))
```

$$-\frac{e^{-\frac{x}{2}}}{2} - \frac{x\,e^{-\frac{x}{2}}}{4} - \frac{e^{-\frac{x}{2}}}{x} \neq \frac{e^{-\frac{x}{2}}}{2} + \frac{x\,e^{-\frac{x}{2}}}{4} - \frac{e^{\frac{x}{2}}}{x}$$

Parameters

a, b, z

arithmetical expressions

Return Values

Arithmetical expression.

Overloaded By

z

Algorithms

$M_{a,\,b}(z)$ and $W_{a,\,b}(z)$ satisfy Whittaker's differential equation:

$$\frac{\partial^2}{\partial z^2}\,y + \left(-\frac{1}{4} + \frac{a}{z} + \frac{\frac{1}{4} - b^2}{z^2}\right) y = 0$$

.

See Also

MuPAD Functions

`hypergeom` | `kummerU` | `whittakerM`

wrightOmega

The Wright ω function

Syntax

```
wrightOmega(x)
```

Description

$\omega(x)$ is defined in terms of Lambert's W function as $\omega(x) = \mathrm{W}_{\left\lceil \frac{\Im(x)-\pi}{2\pi} \right\rceil}(e^x)$.

For $x \neq t$ - i π with $t \leq$ - 1, $y = \omega(x)$ is a solution of the equation $y + ln(y) = x$. The complete solution set of this equation is

$$y = \begin{cases} \varnothing & \text{if } \Re(x) \leq -1 \wedge \Im(x) = -\pi \\ \{\omega(x),\ \omega(x - 2\pi i)\} & \text{if } \Re(x) \leq -1 \wedge \Im(x) = \pi \\ \{\omega(x)\} & \text{otherwise} \end{cases}.$$

A floating-point value is computed if the argument is a floating point value. Unevaluated symbolic calls are returned for most exact arguments. For some special cases explicit symbolic representations are returned.

Environment Interactions

When called with a floating-point argument, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

Most calls with exact arguments return themselves unevaluated:

```
wrightOmega(1/2); wrightOmega(I*PI);
```

$$\omega\left(\frac{1}{2}\right)$$

$$\omega(\pi\, i)$$

Some special arguments return explicit symbolic representations:

```
wrightOmega(-1+I*PI); wrightOmega(ln(2)+6*PI*I);
```

$$-1$$

$$W_3(2)$$

If the argument is a floating-point value, then a floating-point result will be returned:

```
wrightOmega(0.5)
```

$$0.7662486082$$

Parameters

x

An arithmetical expression, the “argument”

Return Values

Arithmetical expression

See Also

MuPAD Functions
`lambertW`

write

Write values of variables into file

Syntax

```
write(<Bin | Text>, <Encoding = "encodingValue">, filename, <x1, x2, …>)
```

```
write(<Encoding = "encodingValue">, n, <x1, x2, …>)
```

Description

`write` serves for storing information from the current MuPAD session in a file. The file contains the values of identifiers of the current session. These identifiers are assigned the stored values when this file is read into another MuPAD session via the function `read`.

`write(filename, x1, x2, ...)` stores the current values of the identifiers x_1, x_2 etc. to the file filename.

`write(filename)` stores the values of all identifiers defined in the current session to the file filename.

`write(n)` and `write(n, x1, x2, ...)` store the data in the file associated with the file descriptor `n`.

`write(..., Encoding = "encodingValue", ...)` stores the current values of identifiers in the specified encoding only when writing in `Text` mode. For the supported encodings, see “Options” on page 1-2215. You can use this option with any of the previously specified syntaxes.

If the file is specified by its name, `write` creates a new file or overwrites an existing file; `write` opens and closes the file automatically.

If `WRITEPATH` does not have a value, `write` interprets the file name as a pathname relative to the “working folder”.

Note that the meaning of “working folder” depends on the operating system. On Microsoft Windows systems and on Mac OS X systems, the “working folder” is the folder

whereMATLAB is installed. On UNIX systems, it is the current working folder in which MATLAB was started. When started from a menu or desktop item, this is typically the user's home folder.

Also absolute path names are processed by `write`.

Instead of a file name, also a file descriptor of a file opened via `fopen` can be used. See “Example 2” on page 1-2212. In this case, the data written by `write` are appended to the corresponding file. The file is not closed automatically by `write` and must be closed by a subsequent call to `fclose`.

Note that `fopen(filename)` opens the file in read-only mode. A subsequent `write` command to this file causes an error. Use the `Write` or `Append` option of `fopen` to open the file for writing.

The file descriptor 0 represents the screen.

`write` stores procedures with the option `noExpose` in encrypted format.

Note: `write` stores the *values* of the given identifiers, not *their full evaluation*! See “Example 3” on page 1-2213.

Environment Interactions

The function is sensitive to the environment variable `WRITEPATH`. If this variable has a value, the file is created in the corresponding folder. Otherwise, the file is created in the “working folder.”

Examples

Example 1

The variable `a` and its value `b + 1` are stored in a file named `test`:

```
a := b + 1:
fid := fopen(TempFile, Write, Text):
write(fid, a):
```

Use `fname` to return the name of the temporary file you created:

```
file := fname(fid):
```

The content of this file is displayed via `ftextinput`:

```
ftextinput(file)
```

```
"a := hold(_plus)(hold(b), 1):"
```

Delete the value of `a`. Reading the file `test` restores the previous value:

```
delete a:
read(file):
a
```

$$b + 1$$

For identifiers that have no value, `write` writes a `delete` command to the file:

```
delete a:
write(Text, 0, a):

delete a:
```

Example 2

The file `test` is opened for writing using the MuPAD binary format:

```
fid := fopen(TempFile):
file := fname(fid):
n := fopen(file, Write)
```

```
17
```

This number is the descriptor of the file and can be used in a write command:

```
a := b + 1:
write(n, a):
fclose(n):

delete a:
```

```
read(file):
a
```

$$b + 1$$

Clean up:

```
delete n, a:
```

Example 3

The value `b + 1` is assigned to the identifier `a`. After assigning the value `2` to `b`, complete evaluation of `a` yields `3`:

```
a := b + 1:
b := 2:
a
```

$$3$$

Note, however, that the value of `a` is the expression `b + 1`. This value is stored by a `write` command:

```
fid := fopen(TempFile, Write, Text):
write(fid, a):
file := fname(fid):
ftextinput(file)
```

"a := hold(_plus)(hold(b), 1):"

Consequently, this value is restored after reading the file into a MuPAD session:

```
delete a, b:
read(file):
a
```

$$b + 1$$

```
delete a:
```

Example 4

`write`, when writing binary format, can store procedures with the option `noExpose` set. They are encrypted before writing:

```
f := proc(a)
  option noExpose;
begin
  print(a, a^2, a*a);
end_proc:

write("hidden_proc.mb", f):

delete f:

read("hidden_proc.mb"):

f(-2...3);
expose(f)
```

$$-2.0 \ldots 3.0,\ 0.0 \ldots 9.0,\ -6.0 \ldots 9.0$$

```
proc(a)
  name f;
  option noDebug, noExpose;
begin
  /* Hidden */
end_proc
```

This is the intention behind option `noExpose`: You can develop code you wish not to publish, then include `option noExpose` in your sources, rerun your tests, use `write` to write a binary version of your library and distribute that.

Example 5

To specify the encoding to write data, use Encoding. The Encoding option applies only to text files that are opened using a file name and not a file descriptor. Write the value of the identifier `a:="abcäöü"` into a temporary file in the encoding “`UTF-8`”:

```
a:="abcäöü":
write(Text,Encoding="UTF-8","write_test",a):
```

Specify the correct encoding to read the file:

```
read("write_test", Encoding="UTF-8")
```

```
"abcäöü"
```

If you do not specify an encoding, the default system encoding is used. Thus, your output might vary from that shown next. Characters unrecognized by the default system encoding are replaced by the default substitution character for that encoding:

```
a:="abcäöü":
write(Text, "write_test", a):
read("write_test")
```

```
"abc"
```

Parameters

`filename`

The name of a file: a character string

x_1, x_2, …

identifiers

`n`

A file descriptor provided by `fopen`: a nonnegative integer

Options

`Bin`, `Text`

With `Bin`, the data are stored in the MuPAD binary format. With `Text`, standard ASCII format is used. The default is `Bin`.

In ASCII format, assignments of the form `identifier := hold(value):` or `delete identifier:` are written into the file. See “Example 1” on page 1-2211.

Encoding

This option lets you specify the character encoding to use. The allowed encodings are:

"Big5"	"ISO-8859-1"	"windows-932"
"EUC-JP"	"ISO-8859-2"	"windows-936"
"GBK"	"ISO-8859-3"	"windows-949"
"KSC_5601"	"ISO-8859-4"	"windows-950"
"Macintosh"	"ISO-8859-9"	"windows-1250"
"Shift_JIS"x	"ISO-8859-13"	"windows-1251"
"US-ASCII"	"ISO-8859-15"	"windows-1252"
"UTF-8"		"windows-1253"
		"windows-1254"
		"windows-1257"

The default encoding is system dependent. If you specify the encoding incorrectly, characters might read incorrectly. Characters unrecognized by the encoding are replaced by the default substitution character for the specified encoding.

Encodings not listed here can be specified but might not produce correct results.

Return Values

Void object of type DOM_NULL.

See Also

MuPAD Functions

doprint | fclose | finput | fname | fopen | fprint | fread | ftextinput | import::readbitmap | import::readdata | pathname | print | protocol | read | READPATH | WRITEPATH

zeta

The Riemann zeta function

Syntax

```
zeta(z)
zeta(z, n)
```

Description

`zeta(z)` represents the Riemann zeta function $\zeta(z) = \sum_{k=1}^{\infty} \frac{1}{k^z}$.

`zeta(z, n)` represents the n-th derivative $\sum_{k=1}^{\infty} \frac{(-\ln(k))^n}{k^z}$ of the zeta function.

The series converge only if the real part of `z` is greater than 1. The definition of the zeta function is extended to the entire complex plane, except for a simple pole `z = 1`, by analytic continuation.

The calls `zeta(z)` and `zeta(z, 0)` are equivalent.

A floating-point result is returned for floating-point arguments *z*.

The following special exact values are implemented:

$\zeta(0) = -\frac{1}{2}$, $\zeta(0, 1) = -\frac{\ln(\pi)}{2} - \frac{\ln(2)}{2}$,

$\zeta(z) = 0$ for even integers $z < 0$,

$\zeta(z) = -\frac{\text{bernoulli}(1-z)}{1-z}$ for odd integers z satisfying - *Pref*::*autoExpansionLimit*() $\le z < 0$,

$\zeta(z) = \frac{(2\pi)^z \,|\text{bernoulli}(z)|}{2\,z!}$ for even integers z satisfying $0 \le z \le$ *Pref*::*autoExpansionLimit*(),

$\zeta(\infty) = 1$, $\zeta(\infty, n) = 0$ for $n > 0$.

`zeta` returns a symbolic function call, if the argument does not evaluate to one of the above numbers.

Note: Floating point evaluation is rather slow for large values of n. Further, for large n, evaluation for $\Re(z) < 0$ is much slower than the evaluation for $\Re(z) \geq 0$.

Environment Interactions

When called with a floating-point argument z, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We demonstrate some calls with exact and symbolic input data:

```
zeta(-6), zeta(-5), zeta(-4), zeta(-3), zeta(-2), zeta(-1)
```

$$0,\ -\frac{1}{252},\ 0,\ \frac{1}{120},\ 0,\ -\frac{1}{12}$$

```
zeta(0), zeta(2), zeta(3), zeta(4), zeta(5), zeta(6), zeta(7)
```

$$-\frac{1}{2},\ \frac{\pi^2}{6},\ \zeta(3),\ \frac{\pi^4}{90},\ \zeta(5),\ \frac{\pi^6}{945},\ \zeta(7)$$

```
zeta(1/2), zeta(1 + I, 1), zeta(z^2 -I, 2)
```

$$\zeta\left(\frac{1}{2}\right),\ \zeta'(1+i),\ \zeta''(z^2-i)$$

Here are some values of the derivative of the zeta function:

```
zeta(0, 1), zeta(infinity, 1)
```

$$-\frac{\ln(2)}{2}-\frac{\ln(\pi)}{2},\ 0$$

Floating point values are computed for floating-point arguments:

```
zeta(-1001.0), zeta(12.3, 1), zeta(0.5 + 14.13472514*I, 2)
```

$$-1.348590824\,10^{1771},\ -0.0001389996909,\ -0.614409794-0.2297836439\,i$$

`zeta` has a pole at the point $z = 1$:

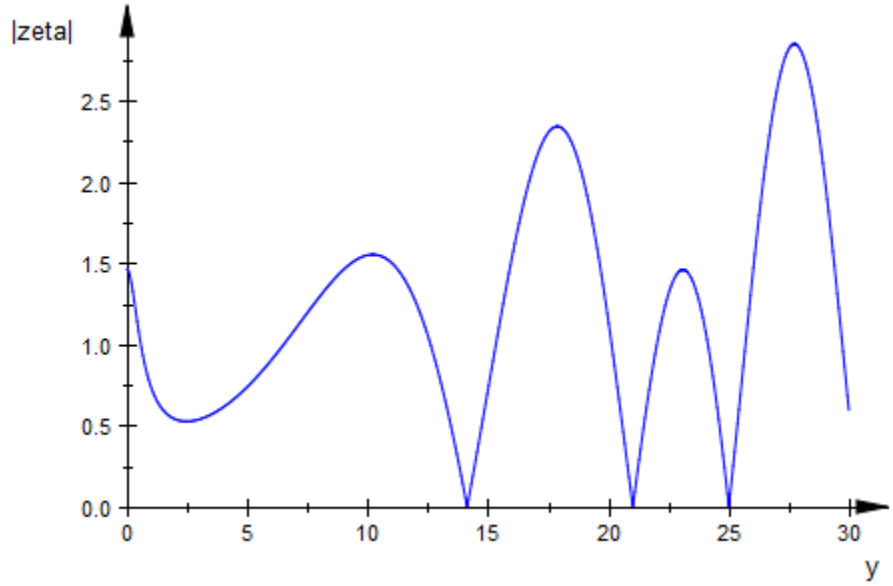
```
zeta(1)
```

```
Error: Singularity. [zeta]
```

Example 2

Looking for nontrivial roots of the Zeta function, we plot the function $f(z) = |\zeta(z)|$ along the "critical line" of complex numbers with real part $\frac{1}{2}$:

```
plotfunc2d(abs(zeta(1/2 + y*I)), y = 0..30,
                                 Mesh = 500,
                                 AxesTitles = ["y", "|zeta|"])
```



The following calls search for numerical roots along the critical line:

```
numeric::solve(zeta(1/2 + I*y), y = 10..20),
numeric::solve(zeta(1/2 + I*y), y = 20..22),
numeric::solve(zeta(1/2 + I*y), y = 22..26)
```

$$\{14.13472514\}, \{21.02203964\}, \{25.01085758\}$$

Parameters

z

An arithmetical expression

n

An arithmetical expression representing a nonnegative integer

Return Values

Arithmetical expression.

Overloaded By

z

See Also

MuPAD Functions

`bernoulli`

zip

Combine lists

Syntax

```
zip(list1, list2, f)
```

```
zip(list1, list2, f, default)
```

Description

`zip(list1, list2, f)` combines two lists via a function `f`. It returns a list whose i-th entry is `f(list1[i], list2[i])`. Its length is the minimum of the lengths of the two input lists.

`zip(list1, list2, f, default)` returns a list whose length is the maximum of the lengths of the two input lists. The shorter list is padded with the `default` value.

If `f` produces the void object of type `DOM_NULL`, then this element is removed from the resulting list.

`zip` is recommended for fast manipulation of lists. It is a function of the system kernel.

Examples

Example 1

The fastest way of adding the entries of two lists is to 'zip' them via the function `_plus`:

```
zip([a, b, c, d], [1, 2, 3, 4], _plus)
```

$$[a+1,\ b+2,\ c+3,\ d+4]$$

If the input lists have different lengths, then the shorter list determines the length of the returned list:

```
zip([a, b, c, d], [1, 2], _plus)
```

$$[a+1,\ b+2]$$

The longer list determines the length of the returned list if a value for padding the shorter list is provided:

```
zip([a, b, c, d], [1, 2], _plus, 17)
```

$$[a+1,\ b+2,\ c+17,\ d+17]$$

Parameters

`list1, list2`

lists of arbitrary MuPAD objects

`f`

Any MuPAD object. Typically, a function of two arguments.

`default`

Any MuPAD object

Return Values

List.

Overloaded By

list1, list2

See Also

MuPAD Functions

`map` | `op` | `select` | `split`

ztrans

Z transform

Syntax

```
ztrans(f, k, z)
```

Description

`ztrans(f, k, z)` computes the Z transform of the expression `f = f(k)` with respect to the index `k` at the point `z`.

The Z transform `F(z)` of the function `f(k)` is defined as follows:

$$F(z) = \sum_{k=0}^{\infty} \frac{f(k)}{z^k}$$

If `ztrans` cannot find an explicit representation of the transform, it returns an unevaluated function call. See “Example 4” on page 1-2225.

If `f` is a matrix, `ztrans` applies the Z transform to all components of the matrix.

To compute the inverse Z transform, use `iztrans`.

Examples

Example 1

Compute the Z transform of these expressions:

```
ztrans(1/k!, k, z)
```

$$e^{\frac{1}{z}}$$

```
ztrans(sin(k), k, z)
```

$$\frac{z \sin(1)}{z^2 - 2\cos(1)\, z + 1}$$

Example 2

Compute the Z transform of this expression and then simplify the result:

```
ztrans(cos(a*k + b), k, z)
```

$$\frac{z \cos(b)\,(z - \cos(a))}{z^2 - 2\cos(a)\, z + 1} - \frac{z \sin(a) \sin(b)}{z^2 - 2\cos(a)\, z + 1}$$

```
Simplify(%)
```

$$-\frac{z\,(\cos(a - b) - z\cos(b))}{z^2 - 2\cos(a)\, z + 1}$$

Example 3

Compute the Z transform of this expression with respect to the variable `k`:

```
F := ztrans(2*k + 3, k, z)
```

$$\frac{3z}{z-1} + \frac{2z}{(z-1)^2}$$

Evaluate the Z transform of the expression at the points $z = 2a + 3$ and $z = 1 + i$. You can evaluate the resulting expression `F` using `|` (or its functional form `evalAt`):

```
F | z = 2*a + 3
```

$$\frac{3\,(2a+3)}{2a+2} + \frac{2\,(2a+3)}{(2a+2)^2}$$

Also, you can evaluate the Z transform at a particular point directly:

```
ztrans(2*k + 3, k, 1 + I)
```

$$1 - 5\,i$$

Example 4

If `ztrans` cannot find an explicit representation of the transform, it returns an unevaluated call:

```
ztrans(f(k), k, z)
```

$$\text{ztrans}(f(k), k, z)$$

`iztrans` returns the original expression:

```
iztrans(%, z, k)
```

$$f(k)$$

Example 5

Compute the following Z transforms that involve Kronecker's Delta function and the Heaviside function:

```
ztrans(f(k)*kroneckerDelta(k, 1) +
       g(k)*kroneckerDelta(k, -5), k, z)
```

$$\frac{f(1)}{z}$$

```
ztrans(binomial(k, 2)*heaviside(5 - k), k, z)
```

$$\frac{z}{(z-1)^3} + \frac{5}{z^5} + \frac{6\,z - \frac{z^6}{(z-1)^3} + 3\,z^2 + z^3}{z^5}$$

Simplify the last expression using `simplify`:

```
simplify(%)
```

$$\frac{z^3 + 3\,z^2 + 6\,z + 5}{z^5}$$

Example 6

Compute the Z transforms of this expression that involves the Heaviside function:

```
ztrans(heaviside(k - 3), k, z)
```

$$\frac{\frac{1}{z-1} + \frac{1}{2}}{z^3}$$

Note that MuPAD uses the value `heaviside(0) = 1/2`. You can define a different value for `heaviside(0)`:

```
unprotect(heaviside):
heaviside(0) := 1:
```

For better performance, MuPAD remembers the previously computed value of the Z transform. To force the system to recalculate the transform, clear its remember table:

```
ztrans(Remember, Clear):
```

For details about the remember mechanism, see Remember Mechanism.

Defining a different value for `heaviside(0)` produces a different value of the Z transform:

```
ztrans(heaviside(k - 3), k, z)
```

$$\frac{\frac{1}{z-1} + 1}{z^3}$$

For further computations, restore the original value:

```
heaviside(0):= 1/2:
protect(heaviside):
```

Example 7

Compute the Z tranforms of these expressions:

```
ztrans(k*f(k), k, z)
```

$$-z\,\frac{\partial}{\partial z}\,\mathrm{ztrans}(f(k),\,k,\,z)$$

```
ztrans(f(k + 1), k, z)
```

$$z\,\mathrm{ztrans}(f(k),\,k,\,z) - z\,f(0)$$

Parameters

f

Arithmetical expression or matrix of such expressions

k

Identifier or indexed identifier

z

Arithmetical expression representing the evaluation point

Return Values

Arithmetical expression or unevaluated function call of type `ztrans`. An explicit result can be a `piecewise` object. If the first argument is a matrix, the result is returned as a matrix.

Overloaded By

f

See Also

MuPAD Functions

`iztrans` | `iztrans::addpattern` | `ztrans::addpattern`

ztrans::addpattern

Add patterns for the Z transform

Syntax

```
ztrans::addpattern(pat, k, z, res, <vars, <conds>>)
```

Description

`ztrans::addpattern(pat, k, z, res)` teaches `ztrans` to return $\text{ztrans}(\text{pat}, k, z) = \sum_{k=0}^{\infty} \frac{\text{pat}}{z^k} = \text{res}$.

The `ztrans` function uses a set of patterns for computing Z transforms. You can extend the set by adding your own patterns. To add a new pattern to the pattern matcher, use `ztrans::addpattern`. MuPAD does not save custom patterns permanently. The new patterns are available in the *current* MuPAD session only.

Variable names that you use when calling `ztrans::addpattern` can differ from the names that you use when calling `ztrans`. See “Example 2” on page 1-2230.

You can include a list of free parameters and a list of conditions on these parameters. These conditions and the result are protected from premature evaluation. This means that you can use `not iszero(a^2 - b)` instead of `hold( _not @ iszero )(a^2 - b)`.

The following conditions treat assumptions on identifiers differently:

- `a^2 - b <> 0` takes into account assumptions on identifiers.
- `not iszero(a^2 - b)` disregards assumptions on identifiers.

See “Example 3” on page 1-2231.

Environment Interactions

Calling `ztrans::addpattern` changes the expressions returned by future calls to `ztrans`.

Examples

Example 1

Compute the Z transform of the function `foo`. By default, MuPAD does not have a pattern for this function:

```
ztrans(foo(k), k, z)
```

$$\text{ztrans}(\text{foo}(k), k, z)$$

Add a pattern for the Z transform of `foo` using `ztrans::addpattern`:

```
ztrans::addpattern(foo(k), k, z, bar(z)):
```

Now `ztrans` returns the Z transform of `foo`:

```
ztrans(foo(k), k, z)
```

$$\text{bar}(z)$$

After you add a new transform pattern, MuPAD can use that pattern indirectly:

```
ztrans(foo(k + 3), k, z)
```

$$z^3\,\text{bar}(z) - z\,\text{foo}(2) - z^2\,\text{foo}(1) - z^3\,\text{foo}(0)$$

Example 2

Define the Z transform of `foo(x)` using the variables `x` and `y` as parameters:

```
ztrans::addpattern(x, x, y, y/(y^2-2*y+1)):
```

The `ztrans` function recognizes the added pattern even if you use other variables as parameters:

```
ztrans(s, s, t)
```

$$\frac{t}{t^2 - 2\,t + 1}$$

Example 3

Use assumptions when adding this pattern for the Z transform:

```
ztrans::addpattern(FOO(x*k), k, z, sin(1/(x - 1/2))*BAR(z),
                                    [x], [abs(x) < 1]):
ztrans(FOO(x*k), k, z) assuming -1 < x < 1
```

$$\sin\left(\frac{1}{x-\frac{1}{2}}\right)\,\mathrm{BAR}(z)$$

If $|x| \geq 1$, you cannot apply this pattern:

```
ztrans(FOO(x*k), k, z) assuming x >= 1
```

$$\mathrm{ztrans}(\mathrm{FOO}(k\,x),\,k,\,z)$$

If MuPAD cannot determine whether the conditions are satisfied, it returns a `piecewise` object:

```
ztrans(FOO(x*k), k, z)
```

$$\begin{cases} \sin\left(\frac{1}{x-\frac{1}{2}}\right)\,\mathrm{BAR}(z) & \text{if } |x| < 1 \end{cases}$$

Note that the resulting expression defining the Z transform of `FOO(x*k)` implicitly assumes that the value of `x` is not 1/2. A strict definition of the pattern is:

```
ztrans::addpattern(BAR(x*k), k, z, sin(1/(x - 1/2))*FOO(z),
                   [x], [abs(x) < 1, x <> 1/2]):
```

If either the conditions are not satisfied or substituting the values into the result gives an error, `ztrans` ignores the pattern. For this particular pattern, you can omit specifying the assumption `x <> 1/2`. If `x = 1/2`, MuPAD throws an internal "Division by zero." error and ignores the pattern:

```
ztrans(FOO(s/2),s,t)
```

$$\mathrm{ztrans}\left(\mathrm{FOO}\left(\frac{s}{2}\right),\,s,\,t\right)$$

Parameters

pat

Arithmetical expression in the variable k representing the pattern to match

k

Identifier or indexed identifier used as a variable in the pattern

z

Identifier or indexed identifier used as a variable in the result

res

Arithmetical expression in the variable k representing the pattern for the result of the transformation

vars

List of identifiers or indexed identifiers used as "pattern variables" (placeholders in pat and res). You can use pattern variables as placeholders for almost arbitrary MuPAD expressions not containing k or z. You can restrict them by conditions given in the optional parameter conds.

conds

List of conditions on the pattern variables

Return Values

Object of type DOM_NULL

See Also

MuPAD Functions

iztrans | iztrans::addpattern | ztrans

2

adt – Abstract Datatypes

adt::Heap
adt::Queue
adt::Stack
adt::Tree

adt::Heap

Abstract data type “Heap”

Syntax

```
adt::Heap()
```

Description

`adt::Heap` implements the abstract data type “Heap”.

A “heap” or “priority queue” is a data type that stores a collection of elements. Elements can be compared and the minimal element can be read and deleted from the heap.

In `adt::Heap`, each element is associated with a comparison key, typically a real number. The keys must be comparable with one another using `<`.

To get access to the largest element in an `adt::Heap`, you can simply negate the comparison keys.

`adt::Heap` returns a function environment. This object has slots `"insert"`, `"nops"`, `"min_pair"`, `"min_element"`, and `"delete_min"` which allow operations on the heap. See the examples.

`adt::Heap` does not allow access to other elements than the minimal one.

Examples

Example 1

`adt::Heap()` creates an empty heap:

```
h := adt::Heap()
```

```
adt::Heap(...)
```

The slot `"nops"` of `h` shows the number of elements in the heap:

```
h::nops()
```

```
     0
```

`h::insert` is the method to insert new elements. It expects two arguments: the comparison key and the data. For now, we simply insert some numbers, so we repeat the number in both arguments:

```
h::insert(3,3):
h::insert(1,1):
h::insert(2,2):

h::nops()
```

```
     3
```

When retrieving the elements with `h::delete_min`, we see that they are returned in increasing order:

```
h::delete_min(), h::delete_min(), h::delete_min()
```

```
     1, 2, 3
```

The heap is now empty:

```
h::nops()
```

```
     0
```

Calling `delete_min` on an empty heap returns `FAIL`:

```
h::delete_min()
```

```
     FAIL
```

Algorithms

`adt::Heap` uses a complete binary tree stored in a list. Insertions operate in expected constant time, with a worst case time logarithmic in the number of elements in the heap. For `"delete_min"`, both the average and the worst-case running time are $O(\log n)$, with n the size of the heap.

See Also

MuPAD Functions

`adt::Queue` | `adt::Stack` | `adt::Tree`

adt::Queue

Abstract data type “Queue”

Syntax

```
adt::Queue(queue)
```

Description

`adt::Queue` implements the abstract data type “Queue”. To create a queue, an expression sequence of any MuPAD objects can be given to initialize the queue, otherwise an empty queue is built.

Note: The methods of all abstract data types must be called especially and will result changing the object itself as side effect.

With `Q := adt::Queue()` an empty queue is built and assigned to the variable `Q`.

Every queue will be displayed as `Queue` followed by a number. This name is generated by `genident`.

Note: *All following methods changes the value of `Q` itself.* A new assignment to the variable (in this example `Q`) is not necessary, in contrast to all other MuPAD functions and data types.

The methods `clear`, `dequeue`, `empty`, `enqueue`, `front`, `length`, `reverse` are available for handling with queues.

Examples

Example 1

Create a new queue with strings as arguments.

```
Q := adt::Queue("1", "2", "3", "4")
```

```
Queue1
```

Show the length of the queue.

```
Q::length()
```

```
4
```

Fill up the queue with a new element. The queue will be changed by the method, no new assignment to `Q` is necessary!

```
Q::enqueue("5")
```

```
"5"
```

Show the front of the queue. This method does not change the queue.

```
Q::front(), Q::front()
```

```
"1", "1"
```

After twice getting an element of the queue, the third element is the new front of the queue, and the length is `3`.

```
Q::dequeue(), Q::dequeue(), Q::front(), Q::length()
```

```
"1", "2", "3", 3
```

Now revert the queue. The last element will be the first element.

```
Q::reverse(): Q::front()
```

```
"5"
```

Enlarge the queue with `"2"`.

```
Q::enqueue("2"):
Q::empty()
```

```
FALSE
```

Finally collect all elements of the queue in the list assigned to ARGS, until the queue is empty.

```
ARGS := []:
while not Q::empty() do ARGS := append(ARGS, Q::dequeue()) end:
ARGS
```

```
["5", "4", "3", "2"]
```

Parameters

queue

An expression sequence of objects to initialize the queue

Methods

clear — Clear the queue

```
clear()
```

dequeue — Get an element from the queue

```
dequeue()
```

empty — I the queue empty

```
empty()
```

enqueue — Fill up the queue

```
enqueue(x)
```

front — Front of the queue

```
front()
```

`length` — Length of the queue

```
length()
```

`reverse` — Revert the queue

```
reverse()
```

adt::Stack

Abstract datatype “Stack”

Syntax

```
adt::Stack(stack)
```

Description

`adt::Stack` implements the abstract data type “Stack.” To create a stack, an expression sequence of any MuPAD objects can be given to initialize the stack, otherwise an empty stack is built.

Note: The methods `adt::Stack`, like those of all abstract data types, change their argument as a side effect.

With `S := adt::Stack()` an empty stack is built and assigned to the variable `S`.

Note: *All following methods change the value of S itself*. A new assignment to the variable (in this example `S`) is not necessary, in contrast to most other MuPAD functions and data types.

The stacks created in a session are named `Stack1, Stack2,...` and printed as such.

Examples

Example 1

We create an empty stack, and fill it with some values:

```
S := adt::Stack();
S::push(a): S::push(b): S::push(c):
```

```
Stack1
```

The stack now contains 3 elements:

```
S::nops()
```

```
3
```

The top of the stack is the last valued pushed:

```
S::top()
```

```
c
```

Now, we fetch successively the values contains in S; they come back in reversed order:

```
S::pop();
S::pop();
S::pop()
```

```
c
```

```
b
```

```
a
```

Now, the stack is empty. Trying to pop again an element from it results in a FAIL value being returned:

```
S::pop()
```

```
FAIL
```

Parameters

`stack`

An expression sequence of objects to initialize the stack

Return Values

Object of the domain `adt::Stack`

Methods

`S::empty` — I the stack empty

```
S::empty()
```

`S::nops` — Size of the stack

```
S::nops()
```

`S::depth` — Depth of the stack

```
S::depth()
```

`S::top` — Top element of the stack

```
S::top()
```

`S::push` — Push an element on the stack

```
S::push(x)
```

`S::pop` — Pop an element from the stack

```
S::pop()
```

`S::reverse` — Revert the stack

```
S::reverse()
```

`S::copy` — Copy of the stack

```
S::copy()
```

adt::Tree

Abstract data type “Tree”

Syntax

```
adt::Tree(tree)
```

Description

`adt::Tree` implements the abstract data type “Tree”.

A tree must be given as a special MuPAD list. The first object of the list is the root of the tree. All further objects are leaves or subtrees of the tree. A subtree is again a special list (as described), and any other MuPAD object will be interpreted as leaf of the tree (see “Example 1” on page 2-13).

A tree can be used to display data in tree structure using the function `output::tree` (or the method `"print"` of a tree). The nodes and leaves of the tree will be printed by MuPAD when the tree will be displayed.

A tree can also be used as datatype to keep and handle any MuPAD data.

Note: The methods of all abstract data types must be called especially and will result changing the object itself as side effect.

`T := adt::Tree([_plus, 3, 4, [_mult, 5, 3], 1])` builds a tree and assigns it to the variable `T`.

Every tree will be displayed as `Tree` followed by a number. This name is generated by `genident`.

To display the content of a tree, the function `expose` or the method `"print"` of the tree itself must be used.

Note: *All following methods changes the value of T itself.* A new assignment to the variable (in this example T) is not necessary, in contrast to all other MuPAD functions and data types.

The methods `nops`, `op`, `expr`, `print`, `indent`, `chars` are now available for handling with trees.

Examples

Example 1

Creating a simple tree with only two leaves. To access and display a tree it must be assigned to a variable:

```
T := adt::Tree(["ROOT", "LEFT", "RIGHT"])
```

```
     Tree1
```

The tree will only be printed by its name. To display the tree, the function `expose` or the method `"print"` of the tree must be used:

```
T::print()
```

```
ROOT
|
+-- LEFT
|
`-- RIGHT
```

```
expose(T)
```

```
ROOT
|
+-- LEFT
|
`-- RIGHT
```

The next tree contains two subtrees as leaves:

```
T := adt::Tree(["ROOT", ["LROOT", "LLEFT", "LRIGHT"],
                        ["RROOT", "RLEFT", "RRIGHT"]]):
T::print()
```

```
ROOT
|
+-- LROOT
|   |
|   +-- LLEFT
|   |
|   `-- LRIGHT
|
`-- RROOT
    |
    +-- RLEFT
    |
    `-- RRIGHT
```

Example 2

Get the operands of a tree: Also a subtree can be an operand:

```
T := adt::Tree(["ROOT", ["LROOT", "LLEFT", "LRIGHT"],
                        "MIDDLE",
                        ["RROOT", "RLEFT", "RRIGHT"]]):
T::op()
```

```
Tree4, "MIDDLE", Tree5
```

Use `expose` to display subtrees:

```
map(%, expose)
```

```
LROOT                  RROOT
|                      |
+-- LLEFT , "MIDDLE", +-- RLEFT
|                      |
`-- LRIGHT             `-- RRIGHT
```

Get all operands including the root:

```
T::op(0..T::nops())
```

"ROOT", Tree6, "MIDDLE", Tree7

Access to various operands:

```
T::op(0);
T::op(2..3);
T::op([1, 2])
```

"ROOT"

"MIDDLE", Tree9

"LRIGHT"

Example 3

The default characters are `["|", "+", "-", „", " "]`:

```
T := adt::Tree(["ROOT", ["LROOT", "LLEFT", "LRIGHT"],
                        ["RROOT", "RLEFT", "RRIGHT"]]):
T::print()
```

```
ROOT
|
+-- LROOT
|   |
|   +-- LLEFT
|   |
|   `-- LRIGHT
|
`-- RROOT
    |
    +-- RLEFT
    |
    `-- RRIGHT
```

The characters can be changed:

```
T::chars(["|", "|", "_", "|", " "]):
T::print()
```

```
ROOT
|
|__ LROOT
|   |
|   |__ LLEFT
|   |
|   |__ LRIGHT
|
|__ RROOT
    |
    |__ RLEFT
    |
    |__ RRIGHT
```

Example 4

A tree visualizes the structure of an expression:

```
T:= adt::Tree([_plus, [_power, [sin, x], 2], [_power, [cos, x], 2]]):
T::print()
```

```
_plus
|
+-- _power
|   |
|   +-- sin
|   |   |
|   |   `-- x
|   |
|   `-- 2
|
`-- _power
    |
    +-- cos
    |   |
    |   `-- x
    |
    `-- 2
```

A tree can be converted to a MuPAD expression:

```
T::expr(), simplify(T::expr())
```

$$\cos(x)^2 + \sin(x)^2,\ 1$$

Parameters

`tree`

The tree, given as a special list (see details)

Methods

`nops` — Number of operands

```
nops()
```

In this example `T` has `4` operands, the numbers `3`, `4`, `1` and the subtree `adt::Tree([_mult, 5, 3])`.

`op` — Operand of a tree

```
op(<n>)
```

`T::op(n)` returns the specified operands of the tree. `n` can be a number between `0` and `T::nops()` (`0` gives the root of the tree), a sequence `i..j` (to return the *i*th to *j*th operand), or a list to specify operands of subtrees (exactly as for the kernel function `op`). `T::op()` returns all operands except the `0`-th as expression sequence. See “Example 2” on page 2-14.

`expr` — Convert a tree to an expression

```
expr()
```

`print` — Display a tree

```
print()
```

`indent` — Indent width of each operand

```
indent(<n>)
```

`chars` — Indent width of each operand

```
chars(<list>)
```

See Also

MuPAD Functions
`output::tree`

3

Ax – Axioms

Ax::canonicalOrder
Ax::canonicalRep
Ax::canonicalUnitNormal
Ax::closedUnitNormals
Ax::efficientOperation
Ax::indetElements
Ax::normalRep
Ax::noZeroDivisors
Ax::systemRep

Ax::canonicalOrder

Axiom of canonically ordered sets

Description

`Ax::canonicalOrder` states that a domain has an order < (`_less`) which is defined by the canonical order of the MuPAD expressions.

This implies that the order of two elements is defined by the system function `_less`.

Ax::canonicalRep

Axiom of canonical representation

Description

`Ax::canonicalRep` states that domain elements are canonically represented, i.e. that each element of the domain has only one unique expression which represents it.

This axiom implies that for an abelian monoid the axiom `Ax::normalRep` also holds. This is not enforced by the category but must be stated by the implementor of a domain.

Ax::canonicalUnitNormal

Axiom of canonical unit normals

Description

`Ax::canonicalUnitNormal` states that the method `"unitNormal"` of an integral domain (category `Cat::IntegralDomain`) returns a unique unit normal.

This means that for each non-zero element `x` of the integral domain there exists an unique associate among the associate class of `x`, i.e. for any `x` and `y` of a domain `dom` of category `Cat::IntegralDomain` where `dom::associates(x, y)` returns `TRUE` the equation `dom::equal(dom::unitNormal(x), dom::unitNormal(y)) = TRUE` must hold.

Note that this axiom does not imply that the unit normals are canonically represented. The unit normals of `x` and `y` must be mathematically equal in the sense of the method `"equal"`, they need not be structurally equal as MuPAD objects.

Ax::closedUnitNormals

Axiom of closed unit normals

Description

`Ax::closedUnitNormals` states that the unit normals of an integral domain are closed under multiplication, i.e., that `dom::equal(x, dom::unitNormal(a) * dom::unitNormal(b)) = TRUE` implies `dom::equal(x, dom::unitNormal(x)) = TRUE` for all elements `x`, `a` and `b` of the domain `dom`.

This axiom may be used only in conjunction with the axiom `Ax::canonicalUnitNormal`. If an integral domain has no unique unit normals, this axiom may not be stated.

Ax::efficientOperation

Axiom of efficient operations

Syntax

```
Ax::efficientOperation(oper)
```

Description

`Ax::efficientOperation(oper)` states that operation `oper` can be performed efficiently.

The string `oper` must be the name of the operation's slot in the domain stating the axiom. Examples are `"_mult"`, `"_invert"` or `"_divide"`.

Parameters

`oper`

A string which defines the efficient operation.

Ax::indetElements

Axiom that indeterminates may be elements

Description

`Ax::indetElements` states that there exist domain elements that may also be regarded as being transcendental over the domain.

`Ax::indetElements` has no mathematical meaning: elements of a ring are always algebraic (of degree 1) over the ring. However, since there are domains in MuPAD that comprise all MuPAD identifiers, insisting on this viewpoint would mean that polynomials over such domains could not be constructed. Hence MuPAD allows the user to regard an identifier as being transcendental over the set of all identifiers.

Ax::normalRep

Axiom of normal representation

Description

`Ax::normalRep` states that an abelian monoid has a canonical representation of its zero element, i.e., that there is only one unique expression to represent zero.

If the axiom `Ax::normalRep` holds for a domain `dom`, one may test for zero by comparing an element with `dom::zero` using the system function `_equal`.

Ax::noZeroDivisors

Axiom of rings with no zero divisor

Description

`Ax::noZeroDivisors` states that a ring without a unit has no zero divisors, i.e., that the product of two non-zero elements is never zero.

Note that an integral domain implicitly has no zero divisors.

Ax::systemRep

Axiom of façade domains

Description

`Ax::systemRep` states that domain elements are represented by elements of built-in domains.

There are principally two ways to represent the elements of a domain: On the one hand the elements may be created explicitly by the system function `new`, on the other hand one may use the built-in (or basic) domains of MuPAD (like `DOM_INT`) to represent the elements.

Domains which don't create elements of their own but use elements of basic domains instead are called *façade domains*.

The usage of basic domains for the representation has the advantage that system functions may be used directly as methods of the domain without the overhead caused by overloading and procedure calls. But it has some severe limitations, see the domain `Dom::Expression` for details.

The axiom `Ax::systemRep` is used to state that the elements of a domain are represented by basic domains and are not created by `new`.

Cat – Categories

Cat::BaseCategory
Cat::AbelianGroup
Cat::AbelianMonoid
Cat::AbelianSemiGroup
Cat::Algebra
Cat::CancellationAbelianMonoid
Cat::CommutativeRing
Cat::DifferentialRing
Cat::EntireRing
Cat::EuclideanDomain
Cat::FactorialDomain
Cat::Field
Cat::FiniteCollection
Cat::GcdDomain
Cat::Group
Cat::HomogeneousFiniteCollection
Cat::HomogeneousFiniteProduct
Cat::IntegralDomain
Cat::LeftModule
Cat::Matrix
Cat::Module
Cat::Monoid
Cat::OrderedSet
Cat::PartialDifferentialRing
Cat::Polynomial
Cat::PrincipalIdealDomain
Cat::QuotientField
Cat::RightModule
Cat::Ring
Cat::Rng
Cat::SemiGroup
Cat::Set

Cat::SkewField
Cat::SquareMatrix
Cat::UnivariatePolynomial
Cat::VectorSpace

Cat::BaseCategory

Base category

Description

`Cat::BaseCategory` is the most general super-category of all categories defined by the Cat package. Any domain in the Dom package is of this category.

The methods defined by `Cat::BaseCategory` are related to type conversion and equality testing, they are not related to an algebraic structure.

Methods

Basic Methods

`convert` — Convert into this domain

```
convert(x)
```

`convert_to` — Convert to certain type

```
convert_to(x, T)
```

`equal` — Test for equality

```
equal(x, y)
```

Note that this method does *not* overload the function `_equal`, i.e. the = operator. The function `_equal` cannot be overloaded.

`expr` — Convert into expression

```
expr(x)
```

Conversion Methods

`coerce` — Coerce into this domain

```
coerce(x)
```

The implementation provided tries to convert `x` into an element of this domain by first calling `dom::convert(x)` and then, if this fails, `x::dom::convert_to(x, dom)`; it retuns `FAIL` if both methods fail.

`equiv` — Test for equivalence

```
equiv(x, y)
```

The implementation provided tries to convert `x` and `y` into elements of this domain and then calls `dom::equal` with these elements. It returns `FAIL` if the conversion fails or the equality test returns `UNKNOWN`.

`new` — Create element of this domain

```
new(x)
```

Given a domain `D`, an expression of the form `D(x,...)` results in a call of the form `D::new(x,...)`.

The implementation provided here tries to convert `x` by calling `dom::convert(x)` and returns the result. It raises an error if `dom::convert` returns `FAIL`.

`print` — Return expression to print an element

```
print(x)
```

Please do *not* print directly in this method by calling the function `print` for example!

The implementation provided here is `dom::expr`.

`testtype` — Test type of object

```
testtype(x, T)
```

This method must return `TRUE` if it can decide that `x` is of type `T`, `FALSE` if it can decide that `x` is not of type `T` and `FAIL` if it can not decide the test.

This method is called in three different situations: Either if the argument `x` is of this domain, or if `T` is this domain, or if `T` is an element of this domain. Thus the following three situations can arise:

- `x` is an element of the current domain.

  In this case it must be tested if `x` may be regarded as an element of the type `T`, which may either be a domain or type expression. By default, this is only true if the domain type of `x` is `T`, or if `T` is a domain constructor for which `x::dom::hasProp(x, T)` is `TRUE`. In particular, `x` is, by default, not of type `T` if `T` is a type of the Type library.
- `T` is the current domain.

  In this case it must be tested if `x` may be regarded as an element of this domain. By the default implementation provided, this is `TRUE` only if the domain type of `x` is `dom`.
- `T` is an element of the current domain.

  In this case `T` is regarded as a type expression. The default implementation provided returns `TRUE` if the domain type of `x` is `T`, and `FAIL` if not. A special rule holds if `T` is a façade domain: in that case, `coerce(x,T)` is called, if this is successful `TRUE` is returned and `FAIL` if not.

Technical Methods

`new_extelement` — Create element of kernel or façade domain

```
new_extelement(x, …)
```

When an expression `new(D,x,...)` is evaluated and `D` is a domain with method `"new_extelement"`, then `D::new_extelement(D,x,...)` is evaluated and returned as result.

Kernel or façade domains must define this method because otherwise the function `new` would return a “container” element of `D` rather than a “raw” element as intended.

The implementation provided here returns the result of `D::new(x,...)`.

Cat::AbelianGroup

Category of Abelian groups

Description

`Cat::AbelianGroup` represents the category of Abelian groups.

A `Cat::AbelianGroup` is an Abelian monoid with cancellation law where the operation + is invertible.

Categories

`Cat::CancellationAbelianMonoid`

Methods

Basic Methods

`_negate` — Return opposite

```
_negate(x)
```

Mathematical Methods

`equal` — Test for equality

```
equal(x, y)
```

`intmult` — Return integer multiple

```
intmult(x, n)
```

_subtract — Subtract two elements

```
_subtract(x, y)
```

Cat::AbelianMonoid

Category of Abelian monoids

Description

`Cat::AbelianMonoid` represents an Abelian monoid.

An `Cat::AbelianMonoid` is an Abelian semi-group with a neutral element `dom::zero` according to the operation + (`_plus`).

Use the axiom `Ax::normalRep` to state that zero is always represented in a unique way (i.e. canonically).

If an Abelian monoid has not the axion `Ax::normalRep` then `dom::zero` is only one possible representation of the neutral element. An Abelian semi-group must at least have the method `"iszero"` to test for zero in such a case.

Axioms

If the domain has `Ax::canonicalRep`, then `Ax::normalRep`.

Categories

`Cat::AbelianSemiGroup`

Entries

"zero"	Must hold the neutral element according to the operation +.

Methods

Mathematical Methods

`intmult` — Return integer multiple

```
intmult(x, n)
```

`iszero` — Test if element is zero

```
iszero(x)
```

Cat::AbelianSemiGroup

Category of Abelian semi-groups

Description

`Cat::AbelianSemiGroup` represents the category of Abelian semi-groups where the operation is written as addition. Hence an `Cat::AbelianSemiGroup` is a set with an associative and commutative operation + (`_plus`).

Note that non-Abelian semi-groups with operation `*` have category `Cat::SemiGroup`.

Categories

`Cat::BaseCategory`

Methods

Basic Methods

`_plus` — Return the sum of its arguments

```
_plus(x, …)
```

Mathematical Methods

`intmult` — Return integer multiple

```
intmult(x, n)
```

Cat::Algebra

Category of associative algebras

Syntax

```
Cat::Algebra(R)
```

Description

`Cat::Algebra(R)` represents the category of associative algebras over the commutative ring `R`.

A `Cat::Algebra(R)` is a module over a commutative ring `R` which also is a ring.

Categories

`Cat::Ring`, `Cat::Module`(R)

Parameters

R

A domain which is a commutative ring. The algebra will be an algebra over this ring.

Cat::CancellationAbelianMonoid

Category of abelian monoids with cancellation

Description

`Cat::CancellationAbelianMonoid` represents the category of Abelian monoids with cancellation.

A `Cat::CancellationAbelianMonoid` is an Abelian monoid where the cancellation law holds according to the operation +, i.e. $a + b = a + c$ implies $b = c$.

Categories

`Cat::AbelianMonoid`

Methods

Basic Methods

`_subtract` — Subtract two elements

```
_subtract(x, y)
```

Mathematical Methods

`equal` — Test for equality

```
equal(x, y)
```

The method `"iszero"` is used to test for zero.

`_negate` — Negate element

```
_negate(x)
```

`intmult` — Return integer multiple

```
intmult(x, n)
```

Cat::CommutativeRing

Category of commutative rings

Description

`Cat::CommutativeRing` represents the category of commutative rings.

A `Cat::CommutativeRing` is a ring with unit `dom::one` where the multiplication `*` (`_mult`) is commutative. It is also a right module over itself.

This implementation additionally assumes that the elements are always constant with respect to differentiation and derivates. One must re-implement the methods `"diff"` and `"D"` if this assumption is false.

Categories

`Cat::Ring`, `Cat::RightModule(dom)`

Methods

Mathematical Methods

`diff` — Differentiate element

```
diff(x, <v, …>)
```

`D` — Return derivative

```
D(l, x)
```

Cat::DifferentialRing

Category of ordinary differential rings

Description

`Cat::DifferentialRing` represents the category of ordinary differential rings.

A `Cat::DifferentialRing` is a commutative ring with a single derivation operator `D`.

A derivation is a linear operator with product rule, i.e. $(f\ g)' = f'\ g + f\ g'$ holds for all f and g.

Categories

`Cat::PartialDifferentialRing`

Methods

Basic Methods

`D` — Return derivative

```
D(f)
```

`diff` — Differentiation with respect to a variable

```
diff(f, x)
```

Cat::EntireRing

Category of entire rings

Description

`Cat::EntireRing` represents the category of entire rings.

An `Cat::EntireRing` is a ring with unit `"one"` which has no zero divisors: Given non-zero ring elements a and b the product a times b is never zero.

Axioms

`Ax::noZeroDivisors`

Categories

`Cat::Ring`, `Cat::RightModule(dom)`

Cat::EuclideanDomain

Category of Euclidean domains

Description

`Cat::EuclideanDomain` represents the category of Euclidean domains.

A `Cat::EuclideanDomain` is a principal ideal domain with an "Euclidean degree" function `"euclideanDegree"` and operations `"quo"` and `"rem"` computing the Euclidean quotient and Euclidean remainder.

The Euclidean degree returns nonnegative integers such that for each non-zero x and y there exist s and r such that $x = y\,s + r$ and either the Euclidean degree of r is less than that of s or r is zero.

In addition s is equal to `quo(x,y)` and r is equal to `rem(x,y)`.

Categories

`Cat::PrincipalIdealDomain`

Methods

Basic Methods

`euclideanDegree` — Return Euclidean degree

```
euclideanDegree(x)
```

`divide` — Division with remainder

```
divide(x, y)
```

Mathematical Methods

`_divide` — Exact division

```
_divide(x, y)
```

`gcd` — Greatest common divisor

```
gcd(x, …)
```

`gcdex` — Extended greatest common divisor

```
gcdex(x, y)
```

`idealGenerator` — Generator of finitely generated ideal

```
idealGenerator(x, …)
```

`quo` — Euclidean quotient

```
quo(x, y)
```

The default implementation provided here uses the basic method `"divide"`.

`rem` — Euclidean remainder

```
rem(x, y)
```

The default implementation provided here uses the basic method `"divide"`.

Cat::FactorialDomain

Category of factorial domains

Description

`Cat::FactorialDomain` represents the category of factorial domains (i.e., unique factorization domains).

A `Cat::FactorialDomain` is an integral domain with gcd where an unique factorization can be computed.

The factorization methods are named `"factor"` and `"sqrfree"` and must return elements of the domain `Factored` over this domain.

Categories

`Cat::GcdDomain`

Methods

Basic Methods

`factor` — Unique factorization

```
factor(x)
```

See `Factored` for details about the representation of the factorization.

Mathematical Methods

`irreducible` — Test if element is irreducible

```
irreducible(x)
```

sqrfree — Square-free factorization

```
sqrfree(x)
```

See `Factored` for details about the representation of the factorization.

The default implementation provided here uses the method `"factor"` and therefore may be very inefficient.

Cat::Field

Category of fields

Description

`Cat::Field` represents the category of fields.

A `Cat::Field` is a factorial domain, an Euclidean domain and a skew field. As a Euclidean domain, it has a commutative multiplication `*` (`_mult`) and as a skew field, the multiplication is invertible.

Many of the methods defined for factorial and Euclidean domains are trivial for a field.

Axioms

`Ax::canonicalUnitNormal`, `Ax::closedUnitNormals`

Categories

`Cat::EuclideanDomain`, `Cat::FactorialDomain`, `Cat::SkewField`

Methods

Mathematical Methods

`associates` — Test for associate elements

```
associates(x, y)
```

`_divide` — Exact division

```
_divide(x, y)
```

divide — Division with remainder

```
divide(x, y)
```

divides — Test if division is exact

```
divides(x, y)
```

euclideanDegree — Return Euclidean degree

```
euclideanDegree(x)
```

factor — Unique factorization

```
factor(x)
```

gcd — Greatest common divisor

```
gcd(x, …)
```

irreducible — Test if element is irreducible

```
irreducible(x)
```

isUnit — Test if element is an unit

```
isUnit(x)
```

quo — Return Euclidean quotient

```
quo(x, y)
```

rem — Return Euclidean remainder

```
rem(x, y)
```

sqrfree — Square-free factorization

```
sqrfree(x)
```

unitNormal — Unit normal form

```
unitNormal(x)
```

`unitNormalRep` — Unit normal representation

```
unitNormalRep(x)
```

Cat::FiniteCollection

Category of finite collections

Description

`Cat::FiniteCollection` represents the category of finite collections, i.e., the category of "universal" bags.

A finite collection is a data structure where each element represents a finite bag of "things" of any type.

The elements are numbered `1,...,nops(c)`, where `nops(c)` is the number of elements in the bag.

Categories

`Cat::BaseCategory`

Methods

Basic Methods

`_index` — Return element given its index

```
_index(x, i)
```

`map` — Map function on elements

```
map(x, f, <a, …>)
```

`nops` — Return number of elements

```
nops(x)
```

`op` — Return certain elements

```
op(x)

op(x, i)
```

Must return the `i`-th element of `x` or `FAIL` if an element with the given index does not exist.

Operand ranges or paths need not be handled by this method because they are handled directly by `op`.

`set_index` — Change element with given index

```
set_index(x, i, v)
```

Overloads the function `_assign` with an `_index` expression on the left hand side. The result is assigned to `x`.

`subs` — Substitute in elements

```
subs(x, e = f)
```

`subsop` — Substitute operands

```
subsop(x, i = v)
```

Technical Methods

`mapCanFail` — Map function on elements

```
mapCanFail(x, f, <a, …>)
```

`testEach` — Test each element with a predicate

```
testEach(x, f, <a, …>)
```

`testOne` — Test if element exists fulfilling a predicate

```
testOne(x, f, <a, …>)
```

Cat::GcdDomain

Category of integral domains with gcd

Description

`Cat::GcdDomain` represents the category of integral domains with a gcd.

A `Cat::GcdDomain` is an integral domain where the greatest common divisor of two elements can be computed by the method `"gcd"`.

Categories

`Cat::IntegralDomain`

Methods

Basic Methods

`gcd` — Greatest common divisor

`gcd(x, …)`

The method must satisfy the following conditions:

1. `x` and `y` must divide `dom::gcd(x,y)`,
2. if `z` divides both `x` and `y`, then `z` must divide `dom::gcd(x,y)`,
3. if a domain has the axiom `Ax::canonicalUnitNormal` then `dom::gcd(x,y)` must be equal to `dom::unitNormal(dom::gcd(x,y))`.

Remember that `x` divides `y` if `_divide(x,y)` does not return `FAIL`.

Mathematical Methods

`lcm` — Least common multiple

```
lcm(x, …)
```

Cat::Group

Category of groups

Description

`Cat::Group` represents the category of groups.

A `Cat::Group` is a (potentially non-Abelian) monoid where the group operation `*` (`_mult`) is invertible.

Categories

`Cat::Monoid`

Methods

Mathematical Methods

`_divide` — Return quotient

```
_divide(x, y)
```

Cat::HomogeneousFiniteCollection

Category of homogeneous finite collections

Syntax

```
Cat::HomogeneousFiniteCollection(T)
```

Description

`Cat::HomogeneousFiniteCollection(T)` represents the category of homogeneous finite collections (i.e. bags) of elements of the domain `T`.

A `Cat::HomogeneousFiniteCollection` is a finite collection where each element of the collection must be from the same domain T.

Categories

`Cat::FiniteCollection`

If T is a `Cat::OrderedSet`, then `Cat::OrderedSet`.

Parameters

`T`

A domain which must be from the category `Cat::BaseCategory`. Only elements of this domain may be contained in the collection.

Entries

"elemDom"	The parameter domain T.

Methods

Mathematical Methods

`_less` — Compare two elements

```
_less(x, y)
```

Returns `TRUE` if `x` is less than `y`.

The collections `x` and `y` are ordered by the lexical ordering of their elements.

Cat::HomogeneousFiniteProduct

Category of homogeneous finite products

Syntax

```
Cat::HomogeneousFiniteProduct(T)
```

Description

`Cat::HomogeneousFiniteProduct(T)` represents the category of homogeneous finite products of elements of the domain `T`.

A `Cat::HomogeneousFiniteProduct(T)` is a homogeneous finite collection where each collection has the same number of elements of the domain `T`.

The number of elements must be given by the entry `"card"`, which must be defined by domains of this category. It is not given as a category parameter simply because it is not needed. Thus no unnecessary instances of the category are created.

One could principally implement all the algebraic operations here, but they will be slow if the methods `"_index"` and `"set_index"` are slow, which most often will be the case. So we avoid the work and let the domain implementors do it.

Categories

`Cat::HomogeneousFiniteCollection`(T)

If `T` is a `Cat::DifferentialRing`, then `Cat::DifferentialRing`.

If `T` is a `Cat::PartialDifferentialRing`, then `Cat::PartialDifferentialRing`.

If `T` is a `Cat::CommutativeRing`, then `Cat::CommutativeRing`.

If `T` is a `Cat::SkewField`, then `Cat::SkewField`.

If `T` is a `Cat::Ring`, then `Cat::Ring`.

If T is a Cat::Rng, then Cat::Rng.

If T is a Cat::AbelianGroup, then Cat::AbelianGroup.

If T is a Cat::CancellationAbelianMonoid, then Cat::CancellationAbelianMonoid.

If T is a Cat::AbelianMonoid, then Cat::AbelianMonoid.

If T is a Cat::AbelianSemiGroup, then Cat::AbelianSemiGroup.

If T is a Cat::Group, then Cat::Group.

If T is a Cat::Monoid, then Cat::Monoid.

If T is a Cat::SemiGroup, then Cat::SemiGroup.

If T is a Cat::CommutativeRing, then Cat::Algebra(T).

If T is a Cat::Ring, then Cat::LeftModule(T).

If T is a Cat::Ring, then Cat::RightModule(T).

Parameters

T

A domain which must be from the category Cat::BaseCategory. This defines the domain of the products elements.

Entries

"card"	Must hold the number of elements of a collection.
"characteristic"	Defined if T is a ring: In this case the characteristic of the product domain is the same as that of T.

Methods

Basic Methods

`zip` — Combine elements

```
zip(x, y, f)
```

`zipCanFail` — Combine elements, may fail

```
zipCanFail(x, y, f)
```

Access Methods

`nops` — Return number of elements

```
nops(x)
```

Cat::IntegralDomain

Category of integral domains

Description

`Cat::IntegralDomain` represents the category of integral domains.

A `Cat::IntegralDomain` is a commutative and entire ring which has a "partial" division method `"_divide"`: If `b` divides `a` then `dom::_divide(a,b)` must return the quotient, otherwise `FAIL`. The result of the method `"_divide"` must be unique.

Use the axiom `Ax::canonicalUnitNormal` to state in addition that there exists a canonical unit normal form for each element of the ring. If a ring has the axiom `Ax::canonicalUnitNormal` the method `"unitNormal"` must return the unique unit normal for a ring element. If the axiom is not valid the method may return any associate.

Use the axiom `Ax::closedUnitNormals` in addition to state that the unit normals which are computed by the method `"unitNormal"` are closed under multiplication, i.e. that the product of two unit normals returns a unit normal.

These two axioms are not implicitly valid for an `Cat::IntegralDomain` because there are integral domains for which one can't compute a canonical unit normal for each element.

Categories

`Cat::EntireRing`, `Cat::CommutativeRing`, `Cat::Algebra(dom)`

Methods

Basic Methods

`_divide` — Return quotient

```
_divide(x, y)
```

The result must be unique:

1. the product `y * dom::_divide(x,y)` must be equal to `x` provided that `y` is not zero and `y` divides `x`,
2. if `x` is equal to `y * z` then `y` must divide `x`.

It is an error if `y` is zero.

`isUnit` — Test if element is a unit

```
isUnit(x)
```

`unitNormal` — Return an associate

```
unitNormal(x)
```

If the ring has the axiom `Ax::canonicalUnitNormal` the method must return the unique unit normal of `x`.

An implementation is provided if the ring has *not* the axiom `Ax::canonicalUnitNormal`: In this case simply `x` is returned.

Mathematical Methods

`associates` — Test if elements are associates

```
associates(x, y)
```

`divides` — Test if elements divides another

```
divides(x, y)
```

`unitNormalRep` — Return the unit normal representation

```
unitNormalRep(x)
```

If the ring has the axiom `Ax::canonicalUnitNormal` the method must return the unique unit normal of `x`. The default implementation uses the method `"unitNormal"` to compute the unit normal `n` in this case.

If the ring dos not have the axiom `Ax::canonicalUnitNormal` the method simply returns `[x, dom::one, dom::one]`.

Cat::LeftModule

Category of left R -modules

Syntax

```
Cat::LeftModule(R)
```

Description

`Cat::LeftModule(R)` represents the category of left `R`-modules.

A `Cat::LeftModule(R)` is an Abelian group together with a rng `R` (a ring without unit) and a left multiplication `*` (`_mult`).

The left multiplication is an operation taking an element of rng `R` and a module element and returning a module element.

Given ring elements a, b and module elements x, y the following 3 distibutive laws must hold:

1. $(a\,b)\,x = a\,(b\,x)$,
2. $(a + b)\,x = a\,x + b\,x$,
3. $a\,(x + y) = a\,x + a\,y$.

Beware: The operation of a non-Abelian semi-group is also written as `*` (`_mult`). The method `"_mult"` must handle the situation if a left module is also a non-Abelian semi-group. In such a case it must both implement the group operation and the left multiplication by elements of the rng.

Categories

`Cat::AbelianGroup`

Parameters

R

A domain which must be from the category `Cat::Rng`.

Methods

Basic Methods

`_mult` — Left multiplication by a rng element

```
_mult(r, x)
```

Cat::Matrix

Category of matrices

Syntax

```
Cat::Matrix(R)
```

Description

`Cat::Matrix(R)` represents the category of matrices over the rng `R`.

A `Cat::Matrix(R)` is a matrix of arbitrary dimension over a component ring `R`.

In the following description of the methods, we use the following notations for a matrix *A* from a domain of category `Cat::Matrix(R)`:

nrows(*A*) denotes the number of rows and *ncols*(*A*) the number of columns of *A*.

Further on, a *row index* is an integer ranges from 1 to *nrows*(*A*), and a *column index* is an integer ranges from 1 to *ncols*(*A*).

Categories

`Cat::BaseCategory`

Parameters

`R`

A domain which must be from the category `Cat::Rng` (a ring without unit).

Entries

"coeffRing"	is set to `R`.

Methods

Basic Methods

`_index` — Matrix indexing

```
_index(A, i, j)
```

`matdim` — Matrix dimension

```
matdim(A)
```

`new` — Matrix definition

```
new(m, n)
```

Of course, this method may implement further possibilites to create matrices (for example, see the method `"new"` of the domain constructor `Dom::Matrix`).

`set_index` — Setting matrix components

```
set_index(A, i, j, x)
```

Mathematical Methods

`_negate` — Negate a matrix

```
_negate(A)
```

`_plus` — Add matrices

```
_plus(A1, A2, …)
```

The matrices must be of the same domain type, otherwise `FAIL` is returned.

`_subtract` — Subtract two matrices

```
_subtract(A, B)
```

`equal` — Test on equality of matrices

```
equal(A, B)
```

`identity` — Identity matrix

```
identity(n)
```

It only exists if `R` is of category `Cat::Ring`, i.e., a ring with unit.

`iszero` — Test on zero matrices

```
iszero(A)
```

Note that there may be more than one representation of the zero matrix of a given dimension if `R` does not have the axiom `Ax::canonicalRep`.

`transpose` — Transpose of a matrix

```
transpose(A)
```

Access Methods

`col` — Extracting columns

```
col(A, c)
```

`concatMatrix` — Horizontal concatenation of matrices

```
concatMatrix(A, B)
```

An error message is issued if the two matrices do not have the same number of rows.

`delCol` — Deleting columns

```
delCol(A, c)
```

If `A` only consists of one column then `NIL` is returned.

`delRow` — Deleting rows

```
delRow(A, r)
```

If A only consists of one row then NIL is returned.

row — Extracting rows

```
row(A, r)
```

setCol — Replacing columns

```
setCol(A, c, v)
```

setRow — Replacing rows

```
setRow(A, r, v)
```

stackMatrix — Appending of matrices vertically

```
stackMatrix(A, B)
```

An error message is issued if the two matrices do not have the same number of columns.

swapCol — Swapping matrix columns

```
swapCol(A, c1, c2)
```

swapRow — Swapping matrix rows

```
swapRow(A, r1, r2)
```

Cat::Module

Category of R-modules

Syntax

```
Cat::Module(R)
```

Description

`Cat::Module(R)` represents the category of `R`-modules.

A `Cat::Module(R)` is a left and right `R`-module over a commutative ring `R`.

Right and left multiplications must be both implemented by the method `"_mult"`.

Categories

`Cat::LeftModule(R)`, `Cat::RightModule(R)`

Parameters

`R`

A domain which must be from the category `Cat::CommutativeRing`.

Cat::Monoid

Category of monoids

Description

`Cat::Monoid` represents the category of monoids.

`Cat::Monoid` is a non-Abelian semi-group with a neutral element one (`dom::one`) according to the group operation `*` (`_mult`).

Categories

`Cat::SemiGroup`

Entries

"one"	Must hold the neutral element according to the operation `*`.

Methods

Basic Methods

`_invert` — Return inverse

```
_invert(x)
```

Mathematical Methods

`isone` — Test if element is one

```
isone(x)
```

_power — Raise to the nth power

```
_power(x, n)
```

This implementation does “repeated squaring”.

Cat::OrderedSet

Category of ordered sets

Description

`Cat::OrderedSet` represents the category of ordered sets.

An `Cat::OrderedSet` is a set with a (complete) order relation < (`_less`).

Use the axiom `Ax::canonicalOrder` to state that elements of a domain are canonically ordered as MuPAD expressions (i.e. ordered with respect to the kernel function `_less`).

Categories

`Cat::BaseCategory`

Methods

Basic Methods

`_less` — Compare if element is less

```
_less(x, y)
```

An implementation is provided if this domain has axiom `Ax::canonicalOrder`.

Mathematical Methods

`_leequal` — Compare if element is less or equal

```
_leequal(x, y)
```

The implementation provided uses the methods `"_less"` and `"equal"`.

`max` — Return maximum

```
max(x, …)
```

`min` — Return minimum

```
min(x, …)
```

`sort` — Sort list of elements

```
sort(l)
```

Cat::PartialDifferentialRing

Category of partial differential rings

Description

`Cat::PartialDifferentialRing` represents the category of partial differential rings.

A `Cat::PartialDifferentialRing` is a commutative ring with a finite set of derivation operators `D_i`.

A derivation is a linear operator with product rule, i.e. `D_i(f * g)` equals `D_i(f) * g + f * D_i(g)` for all `f` and `g`.

For many partial differential rings the derivations are differentiations with respect to some indeterminates. Thus in order to support a natural notion it is also supposed that a method `"diff"` exists, such that `diff(f, x)` returns the partial derivation of `f` with respect to the indeterminate `x`.

Categories

`Cat::CommutativeRing`

Methods

Basic Methods

`D` — Return derivative

```
D(l, x)
```

`diff` — Return partial derivative

```
diff(x, <v, …>)
```

Cat::Polynomial

Category of multivariate polynomials

Syntax

```
Cat::Polynomial(R)
```

Description

`Cat::Polynomial(R)` represents the category of multivariate polynomials over `R`.

A `Cat::Polynomial(R)` is a multivariate polynomial ring over a commutative coefficient ring `R`.

Axioms

If `R` has `Ax::canonicalUnitNormal`, then `Ax::canonicalUnitNormal`.

If `R` has `Ax::closedUnitNormals`, then `Ax::closedUnitNormals`.

Categories

`Cat::PartialDifferentialRing`, `Cat::Algebra(R)`

If `R` is a `Cat::FactorialDomain`, then `Cat::FactorialDomain`.

If `R` is a `Cat::GcdDomain`, then `Cat::GcdDomain`.

If `R` is a `Cat::IntegralDomain`, then `Cat::IntegralDomain`.

Parameters

`R`

A domain which must be from the category `Cat::CommutativeRing`.

Entries

"coeffRing"	The coefficient ring R.
"characteristic"	The characteristic of this domain, which is the same as that of the ring R.

Methods

Basic Methods

coeff — Return coefficients

```
coeff(p)
```

```
coeff(p, x, n)
```

```
coeff(p, n)
```

Must return the coefficient of x^n of p, which is a polynomial in the remaining indeterminates.

Must return the coefficient of x^n of p, where x is the main variable of p.

degree — Return total degree

```
degree(p)
```

```
degree(p, x)
```

Must return the degree of p with respect to the indeterminate x.

degreevec — Return degree vector

```
degreevec(p)
```

evalp — Evaluate at a point

```
evalp(p, x = v, …)
```

More than one evaluation point may be given. The result must be a polynomial in the remaining indeterminates or an element of R.

`indets` — Return indeterminates

```
indets(p)
```

`lcoeff` — Return leading coefficient

```
lcoeff(p)
```

`lmonomial` — Return leading monomial

```
lmonomial(p)
```

`lterm` — Return leading term

```
lterm(p)
```

`mainvar` — Return main variable

```
mainvar(p)
```

`mapcoeffs` — Map coefficients

```
mapcoeffs(p, f, <a, …>)
```

`multcoeffs` — Multiply coefficients

```
multcoeffs(p, c)
```

`nterms` — Return number of terms

```
nterms(p)
```

`nthcoeff` — Return n-th coefficient

```
nthcoeff(p, n)
```

`nthmonomial` — Return n-th monomial

```
nthmonomial(p, n)
```

`nthterm` — Return n-th term

```
nthterm(p, n)
```

`tcoeff` — Return trailing coefficient

```
tcoeff(p)
```

`unitNormal` — Return unit normal

```
unitNormal(p)
```

An implementation is provided if `R` has the axiom `Ax::canonicalUnitNormal`: In this case `p` is multiplied by an unit of `R` such that the leading coefficient has unit normal representation in `R`.

`unitNormalRep` — Return unit normal representation

```
unitNormalRep(p)
```

An implementation is provided if `R` has the axiom `Ax::canonicalUnitNormal`.

Mathematical Methods

`content` — Return content

```
content(p)
```

`isUnit` — Test if element is a unit

```
isUnit(p)
```

`primpart` — Return primitive part

```
primpart(p)
```

`poly2list` — Convert into a list

```
poly2list(p)
```

`solve` — Solve polynomial equation

```
solve(p, x, <opt, …>)
```

```
solve(p, x = T, <opt, …>)

solve(p)
```

Solves the polynomial equation `p = 0` with respect to `x` over the domain `T`. See the function `solve` for details about the optional arguments `opt, ...`

The polynomial `p` must be univariate. Solves the polynomial equation `p = 0` with respect to the indeterminate of `p` over the domain `R`.

Cat::PrincipalIdealDomain

Category of principal ideal domains

Description

`Cat::PrincipalIdealDomain` represents the category of principal ideal domains.

A `Cat::PrincipalIdealDomain` is an integral domain with gcd where each ideal is principal. Note that the method `"idealGenerator"` has to find generators for finitely generated ideals only.

Categories

`Cat::GcdDomain`

Methods

Basic Methods

`idealGenerator` — Return generator of ideal

```
idealGenerator(x, …)
```

Cat::QuotientField

Category of quotient fields

Syntax

```
Cat::QuotientField(R)
```

Description

`Cat::QuotientField(R)` represents the category of quotient fields over `R`.

A `Cat::QuotientField` is the field of fractions over the integral domain `R`.

Categories

`Cat::Field`, `Cat::Algebra(R)`

If `R` has `Cat::OrderedSet`, then `Cat::OrderedSet`.

Parameters

`R`

A domain which must be from the category `Cat::IntegralDomain`.

Entries

"characteristic"	The characteristic of this domain, which is the same as that of `R`.

Methods

Basic Methods

`denom` — Return denominator

```
denom(x)
```

`numer` — Return numerator

```
numer(x)
```

Mathematical Methods

`equal` — Test for equality

```
equal(x, y)
```

`iszero` — Test for zero

```
iszero(x)
```

`_less` — Test if element is less

```
_less(x, y)
```

`retract` — Return retracted element

```
retract(x)
```

The default implementation uses the method "`_divide`" to divide numerator and denominator.

Cat::RightModule

Category of right R-modules

Syntax

```
Cat::RightModule(R)
```

Description

`Cat::RightModule(R)` represents the caregory of right `R`-modules.

A `Cat::RightModule` is an Abelian group together with a ring `R` and a right multiplication `*` (`_mult`).

The right multiplication is an operation taking an element of ring `R` and a module element and returning a module element.

Given ring elements a, b and module elements x, y the following 3 distributive laws must hold:

1. $x\,(a\,b) = (x\,a)\,b$,
2. $x\,(a + b) = x\,a + x\,b$,
3. $(x + y)\,a = x\,a + y\,a$.

Beware: The operation of a non-Abelian semi-group is also written as `*` (`_mult`). The method `"_mult"` must handle the situation if a right module is also a non-Abelian semi-group. In such a case it must both implement the group operation and the right multiplication by elements of the ring.

Categories

`Cat::AbelianGroup`

Parameters

R

A domain which must be from the category `Cat::Ring`.

Methods

Basic Methods

`_mult` — Right multiplication by a ring element

```
_mult(x, r)
```

Cat::Ring

Category of rings

Description

`Cat::Ring` represents the category of rings.

A `Cat::Ring` is a ring with a unit `dom::one`, i.e., an Abelian group according to the operation `+` (`_plus`) and a non-Abelian monoid according to the operation `*` (`_mult`) where in addition the two distributive laws $a\,(b + c) = a\,b + a\,c$ and $(a + b)\,c = a\,c + b\,c$ hold.

A `Cat::Ring` is also a left module over itself. The left multiplication of the module is also written as `*` (`_mult`).

Note that a ring without unit is a `Cat::Rng`.

Categories

`Cat::Rng`, `Cat::Monoid`, `Cat::LeftModule(dom)`

Entries

"characteristic"	Must hold the characteristic of this ring.

Cat::Rng

Category of rings without unit

Description

`Cat::Rng` represents the category of rings without unit.

A `Cat::Rng` is a ring without a unit, i.e. an Abelian group according to the operation `+` (`_plus`) and a non-Abelian semi-group according to the operation `*` (`_mult`) where in addition the two distributive laws $a\,(b + c) = a\,b + a\,c$ and $(a + b)\,c = a\,c + b\,c$ hold.

Use the axiom `Ax::noZeroDivisors` to state that there are no zero divisors according to `*`, i.e. that the product of non-zero elements never is zero.

Categories

`Cat::AbelianGroup`, `Cat::SemiGroup`

Cat::SemiGroup

Category of semi-groups

Description

`Cat::SemiGroup` represents the category of semi-groups.

A `Cat::SemiGroup` represents the category of non-Abelian semi-groups, where the group operation is written as multiplication. Hence a `Cat::SemiGroup` is a set with an associative operation `*` (`_mult`).

Note that Abelian semi-groups with operation + have category `Cat::AbelianSemiGroup`.

Categories

`Cat::BaseCategory`

Methods

Basic Methods

`_mult` — Return product

```
_mult(x, …)
```

Mathematical Methods

`_power` — Return integer power

```
_power(x, n)
```

Cat::Set

Category of sets of complex numbers

Description

`Cat::Set` represents the category of subsets of the complex numbers.

Sets of this category allow set-theoretic operations as well as pointwise arithmetical operations.

The main feature of `Cat::Set` is a particular overloading mechanism. It provides `n`-ary operators that can handle operands from different domains of category `Cat::Set`, as well as mixed input where some operands are of types not belonging to `Cat::Set`. *Hence, in the methods of* `Cat::Set`*, operands of arbitrary type are allowed.*

There are three kinds of operators: `n`-ary (associative and commutative), binary (not assumed to be commutative), and unary (mapping a function). `Cat::Set` provides generic methods for generating these kinds of operators, and uses them to define default methods overloading the common set-theoretic and arithmetical functions.

By default, any operation of sets is defined, but returns unevaluated since the arithmetical or set-theoretic expression cannot be simplified. Each domain of type `Cat::Set` must provide particular slots and tables in order to achieve simplifications in certain special cases.

Arithmetical operations are defined pointwise. It is not an error if some operation is not defined for all elements of a set.

`Cat::Set` is mainly used by domains of sets returned by `solve`.

Categories

`Cat::BaseCategory`

Methods

Mathematical Methods

commassop — Return an n-ary commutative and associative operator for sets

```
commassop(operatorname)
```

The returned procedure first sorts its operands (which it may do because of commutativity). Those operands not belonging to a domain of category `Cat::Set` are handled by the usual overloading mechanism, i.e. by the slot `operatorname` of one of their domains. Out of the others, several operands belonging to the same domain are handled by the slot `"homog".operatorname` of that domain. Finally, the returned method tries to combine each possible pair of operands. If they are from the same domain, `"bin".operatorname` is called for them. The following is done if the operands are from different domains: let `T1` and `T2` be their types; then their `"inhomog".operatorname` slots are used. If such a slot exists in the domain `T1`, it must contain a table indexed by possible types `T2`, and the entry at that index must be a procedure that carries out the operation for exactly two arguments, the first being a `T1`, the second being a `T2`. Conversely, if such a slot exists in the domain `T2`, it must contain a table indexed by possible types `T1`, and the entry at that index must be a procedure that carries out the operation for exactly two arguments, the first being a `T2`, the second being a `T1`.

The slot `"homog".operatorname`, or a table entry in the slot `"inhomog".operatorname`, may return `FAIL` in order to indicate that it could not simplify its input; if they are missing, this indicates that a simplification is generally not possible for input of this type. In these cases, the returned procedure proceeds by trying to combine another two of the given arguments.

A slot `"bin".operatorname` usually won't exist, except for the case that there is no `"homog".operatorname`; usually the latter can also take care for the case of exactly two operands.

The whole process is repeated over and over until no new simplifications occur or only one operand is left. If no more simplifications occur, an unevaluated call to the operator is returned, the arguments being all remaining operands that could not be combined further.

`binop` — Return a binary operator for sets

```
binop(operatorname)
```

The returned procedure uses the slot `"bin".operatorname` of its first argument if both arguments are of the same type. Otherwise it uses the slot `"inhomogleft".operatorname` of its first argument; if that fails, it uses the slot `"inhomogright".operatorname` of its second argument; each of these slots, if it exists, must contain tables, indexed by the type of the other argument, such that `slot(T1, "inhomogleft".operatorname)[T2]` and `slot(T2, "inhomogright".operatorname)[T1]` carry out the operation for objects of type `T1` and `T2`, in this order.

No commutativity of the operation is assumed.

If the slots or table entries do not exist or return `FAIL`, an unevaluated call to the operator is returned.

`homogassop` — Return an n-ary operator for sets belonging to the same domain

```
homogassop(operatorname)
```

`_union` — Union of sets

```
_union(S1, …)
```

`_intersect` — Intersection of sets

```
_intersect(S1, …)
```

`_plus` — Set of sums of set elements

```
_plus(S1, …)
```

The sum of sets is computed by the commutative-associative operator generated by `"commassop"`, using the slots `"homog_plus"` and `"inhomog_plus"` of the domains of its operands.

`_mult` — Set of product of set elements

```
_mult(S1, …)
```

The product of sets is computed by the commutative-associative operator generated by `"commassop"`, using the slots `"homog_mult"` and `"inhomog_mult` of the domains of its operands.

`_minus` — Set of subtractions

`_minus(`S_1`, `S_2`)`

`_power` — Pointwise power

`_power(`S_1`, `S_2`)`

The power of sets is computed by the binary operator generated by `"binop"`, using the slots `"homog_power"`, `"inhomogleft_power"`, and `"inhomogright_power"` of its operands.

`map` — Map an operation to a set

`map(S, f)`

By overloading this method in a particular domain, the behavior of sets changes whenever a special function is applied to them.

Cat::SkewField

Category of skew fields

Description

`Cat::SkewField` represents the category of skew fields (division rings).

A `Cat::SkewField` represents a ring with unit where each non-zero element is invertible. This structure is also called division ring in the literature.

Categories

`Cat::Ring`

Cat::SquareMatrix

Category of square matrices

Syntax

```
Cat::SquareMatrix(R)
```

Description

`Cat::SquareMatrix(R)` represents the category of square matrices over the rng `R`.

A `Cat::SquareMatrix(R)` represents the rng (ring without unit) of square matrices over the coefficient domain `R`.

Categories

`Cat::Rng`, `Cat::Matrix(R)`

If `R` has `Cat::Ring`, then `Cat::Ring`.

Parameters

`R`

A domain which must be from the category `Cat::Rng`.

Entries

"characteristic"	Defined if `R` is a ring: In this case the characteristic of the matrix domain is the same as that of `R`.

Cat::UnivariatePolynomial

Category of univariate polynomials

Syntax

```
Cat::UnivariatePolynomial(R)
```

Description

`Cat::UnivariatePolynomial(R)` represents the category of univariate polynomials over `R`.

A `Cat::UnivariatePolynomial(R)` is a univariate polynomial over the commutative ring `R`.

Categories

`Cat::Polynomial(R)`, `Cat::DifferentialRing`

If `R` has `Cat::Field`, then `Cat::EuclideanDomain`.

Parameters

`R`

A domain which must be from the category `Cat::CommutativeRing`.

Methods

Basic Methods

`pdivide` — Pseudo-divide polynomials

```
pdivide(p, q)
```

Must return a sequence `(b, s, r)` of a ring element `b` and polynomials `s` and `r` such that $multcoeffs(p, b) = s\,q + r$ holds with $b = lcoeff(q)^{degree(p) - degree(q) + 1}$.

`pquo` — Return pseudo-quotient

```
pquo(p, q)
```

`prem` — Return pseudo-remainder

```
prem(p, q)
```

Cat::VectorSpace

Category of vector spaces

Syntax

```
Cat::VectorSpace(F)
```

Description

`Cat::VectorSpace(F)` represents the category of vector spaces over the field `F`.

A vector space is an Abelian group with an operation + (`_plus`).

The scalar product has to be implemented via the method `"_mult"`. Other kinds of multiplication are not defined.

Categories

`Cat::Module(F)`

Parameters

`F`

A domain which must be from the category `Cat::Field`.

Methods

Basic Methods

`_mult` — Return scalar product

```
_mult(c, x)
```

`_mult(x, c)`

Must return the scalar product of `x` and `c`.

combinat – Combinatorics

combinat::bell
combinat::cartesianProduct
combinat::catalan
combinat::choose
combinat::compositions
combinat::modStirling
combinat::partitions
combinat::permute
combinat::powerset
combinat::stirling1
combinat::stirling2
combinat::subwords

combinat::bell

Bell numbers

Syntax

```
combinat::bell(n)
```

```
combinat::bell(expression)
```

Description

`combinat::bell(n)` computes the n-th Bell number.

The n-th Bell number is defined by the exponential generating function:

$$e^{e^x-1} = \sum_{n=0}^{\infty} \frac{\mathrm{bell}(n)}{n!} x^n$$

Often another definition is used. The n-th Bell number is the number of different ways of partitioning the set {1, 2, …, *n*} into disjoint nonempty subsets, and *bell*(0) is defined to be 1.

Bell numbers are computed using the formula:

- *bell*(0) = 1

  $\mathrm{bell}(n+1) = \sum_{i=0}^{n} \binom{n}{i} \mathrm{bell}(i)$ for $n > 0$

Examples

Example 1

The third Bell number is 5:

```
combinat::bell(3)
```

$$5$$

This means that you can partition the set {1, 2, 3} into disjoint subsets in 5 different ways. These are {{1, 2, 3}}, {{1}, {2, 3}}, {{2}, {1, 3}}, {{3}, {1, 2}}, and {{1}, {2}, {3}}. Or, that you can write 105 = 3 5 7 as 5 different products. These are 105 = 3 35 = 5 21 = 7 15 = 3 5 7.

Example 2

If one uses a wrong argument, an error message is returned.

```
combinat::bell(3.4)
```

```
Error: A nonnegative integer is expected. [combinat::bell]
```

Example 3

It can be useful to return the unevaluated function call.

```
a := combinat::bell(x);
x := 4;
a ;
delete(a);
```

$$\text{combinat::bell}(x)$$

$$4$$

$$15$$

Parameters

n

Nonnegative integer

`expression`

An expression of type `Type::Arithmetical` which must be a nonnegative integer if it is a number.

Return Values

Positive integer value if `n` was a nonnegative integer. Otherwise `combinat::bell` returns the unevaluated function call.

combinat::cartesianProduct

Cartesian product

Syntax

```
combinat::cartesianProduct(S1, …)
```

Description

`combinat::cartesianProduct(S1, ...)` returns the cartesian product of the sets or lists *S1*, … as a list of lists.

The cartesian product of S_1 through S_n consists of all lists of length n whose i-th entry is an operand of the set or list S_i, for $1 \le i \le n$.

Any integer k among the arguments is identified with the set of the first k positive integers.

The ordering of the output is unspecified.

Examples

Example 1

The following calls are equivalent:

```
combinat::cartesianProduct({1, 2}, {a, b}),
combinat::cartesianProduct(2, [b, a])
```

$$[[1, a], [2, a], [1, b], [2, b]], [[1, b], [2, b], [1, a], [2, a]]$$

Parameters

S1

Set, list, or nonnegative integer

Return Values

List of lists, each of them having as many operands as there were arguments passed to `combinat::cartesianProduct`.

combinat::catalan

Catalan numbers

Syntax

```
combinat::catalan(n)
```

Description

`combinat::catalan`(n) returns the `n`-th Catalan number.

The Catalan numbers are ubiquitous in combinatorics. For example, `combinat::catalan(n)` counts the Dyck words of size `n`, the ordered trees with `n` nodes, the binary trees with `n+1` nodes, the complete binary trees with `2n+1` nodes, the standard tableaux with two rows of size `n`, the triangulations of a regular `n+2`-gone, or the non-crossing partitions of {1, 2, …, *n*}.

`combinat::catalan(n)` is calculated using the formula

$$\text{catalan}(n) = \frac{1}{n+1}\binom{2n}{n}.$$

Examples

Example 1

We compute the first Catalan numbers:

```
combinat::catalan(n) $ n = 0..6
```

$$1, 1, 2, 5, 14, 42, 132$$

Example 2

If one uses a wrong argument, an error message is returned

```
combinat::catalan(-1)
```

```
Error: The object '-1' is incorrect. The type of argument number 1 must be 'Type::NonNe
  Evaluating: combinat::catalan
```

Parameters

n

Nonnegative integer

Return Values

Positive integer.

combinat::choose

Subsets of a given size

Syntax

```
combinat::choose(S, k)
```

Description

`combinat::choose(S, k)` returns all subsets of `S` that have exactly `k` elements.

If `S` is an integer, it represents the set of the first `S` positive integers.

Examples

Example 1

There are three subsets of a three-element set that have exactly two elements:

```
combinat::choose({a, b, c}, 2)
```

$$\{a, b\}, \{a, c\}, \{b, c\}$$

Parameters

S

Set or nonnegative integer

k

Nonnegative integer

Return Values

Sequence of sets.

combinat::compositions

Compositions of an integer

Syntax

```
combinat::compositions(n, <MinPart = k>, <MaxPart = l>, <Length = m>)
```

Description

`combinat::compositions(n)` returns all compositions of the nonnegative integer `n`.

A *composition* of a nonnegative integer `n` is a list of positive integers with total sum `n`.

Examples

Example 1

We output all compositions of the integer 4:

```
combinat::compositions(4)
```

```
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 3], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
```

Example 2

It is possible to output only the compositions of a certain length:

```
combinat::compositions(4, Length=2)
```

```
[[3, 1], [2, 2], [1, 3]]
```

Example 3

The options `MinPart` and `MaxPart` can be used to set constraints on the sizes of all parts. Using `MaxPart`, you can select compositions having only small entries. This is the list of the compositions of 4 with all parts at most 2:

```
combinat::compositions(4, MaxPart=2)
```

```
[[2, 2], [2, 1, 1], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
```

`MinPart` is complementary to `MaxPart` and selects compositions having only large parts (it takes a non-negative value). This is the list of the compositions of 4 with all parts at least 2:

```
combinat::compositions(4, MinPart=2)
```

```
[[4], [2, 2]]
```

By default, the parts of a composition have to be positive. This can be changed using the option `MinPart`. In the following example, the options `Length` and `MinPart` are combined together to obtain the list of the compositions of 4 with 3 nonnegative parts:

```
combinat::compositions(4, Length=3, MinPart=0)
```

```
[[4, 0, 0], [3, 1, 0], [3, 0, 1], [2, 2, 0], [2, 1, 1], [2, 0, 2], [1, 3, 0], [1, 2, 1], [1, 1, 2], [1, 0, 3],
  [0, 4, 0], [0, 3, 1], [0, 2, 2], [0, 1, 3], [0, 0, 4]]
```

If no length is given, `MinPart=0` is not allowed.

Parameters

`n`

Nonnegative integer

Options

`MinPart`

Option, specified as `MinPart = k`

Return only compositions consisting of integers greater or equal than `k`. The option `MinPart = 0` is only allowed if also the option `Length` is given. Default is 1.

`MaxPart`

Option, specified as `MaxPart = l`

Return only compositions consisting of integers less or equal than `l`.

`Length`

Option, specified as `Length = m`

Return only compositions consisting of exactly `m` integers.

combinat::modStirling

Modified Stirling numbers

Syntax

```
combinat::modStirling(q, n, k)
```

Description

`combinat::modStirling` computes the modified Stirling numbers.

`combinat::modStirling(q,n,k)` takes the elementary symmetric polynomial in `n` variables of degree `k` and evaluates it for the values $q + 1, \ldots, q + n$. Note that `k` must not be greater than `n`.

Examples

Example 1

```
combinat::modStirling(2,4,2)
```

```
     119
```

Parameters

q

The argument: an integer

n

The number of variables: a nonnegative integer

k

The degree: a nonnegative integer

Return Values

Positive integer.

combinat::partitions

Partitions of an integer

Syntax

```
combinat::partitions(n)
```

Description

`cominat::partitions(n)` returns the number of partitions of the integer `n`.

A *partition* of a nonnegative integer `n` is a non-increasing list of positive integers with total sum `n`.

Examples

Example 1

There are 5 partitions of 4:

```
combinat::partitions(4)
```

```
    5
```

Parameters

`n`

Nonnegative integer

Algorithms

Counting is done efficiently with Euler's pentagonal formula for small values of n and Hardy-Ramanujan-Rademacher's formula otherwise.

combinat::permute

Permutations of a list

Syntax

```
combinat::permute(l, <Duplicate>)
```

```
combinat::permute(n, <Duplicate>)
```

Description

For a list `l`, the call `combinat::permute(l)` returns all permutations of `l`.

For an integer `n`, the call `combinat::permute(n)` returns all permutations of the list `[1, ..., n]`.

A permutation of a list is a list that contains the same elements, and each of them the same number of times, as the original list.

Equivalently, a permutation of a list `l` of n elements is any `f(l)` where `f` is an element of the symmetric group `Dom::SymmetricGroup(n)`. Different `f` may produce the same `f(l)`; with the option `Duplicate`, every permutation is listed as many times as it occurs in that way; without that option, every permutation is listed only once.

Examples

Example 1

There are six permutations of three letters:

```
combinat::permute([a, b, c])
```

$$[[b, c, a], [c, b, a], [a, c, b], [c, a, b], [a, b, c], [b, a, c]]$$

To permute the first three integers, the following syntax is also possible:

```
combinat::permute(3)
```

```
[[2, 3, 1], [3, 2, 1], [1, 3, 2], [3, 1, 2], [1, 2, 3], [2, 1, 3]]
```

If some list entry occurs several times, the number of permutations decreases:

```
combinat::permute([a, a, b])
```

```
[[a, b, a], [b, a, a], [a, a, b]]
```

However, the same permutation is listed as often as it can be obtained by applying different elements of the symmetric group S_3 if the option `Duplicate` is given.

```
combinat::permute([a, a, b], Duplicate)
```

```
[[a, b, a], [b, a, a], [a, b, a], [b, a, a], [a, a, b], [a, a, b]]
```

We could have achieved the same by permuting three different symbols and then setting two of them equal:

```
subs(combinat::permute([a, b, c]), c=a)
```

```
[[b, a, a], [a, b, a], [a, a, b], [a, a, b], [a, b, a], [b, a, a]]
```

Parameters

`l`

List

`n`

Positive integer

Options

`Duplicate`

List every permutation as often as it can be produced in different ways by applying some bijective mapping (element of the symmetric group) to the original list.

combinat::powerset

Subsets of a set

Syntax

```
combinat::powerset(S)
```

Description

If `S` is a set, `combinat::powerset(S)` returns the set of all subsets of `S`. If `l` is a list, `combinat::powerset(l)` returns the set of all sublists of `l`.

The powerset of a list `l` is the set of all lists that can be obtained by deleting some elements of `l` and leaving the others in order.

`combinat::powerset` has been overloaded for multisets of type `Dom::Multiset`. The powerset of a multiset `S` consists of all multisets that contain only elements occurring also in `S`, each of them at most as many times as it occurs in `S`.

Examples

Example 1

Given a finite set, `combinat::powerset` returns the powerset (set of all subsets) of the input:

```
combinat::powerset({a, b, c})
```

$$\{\varnothing, \{c\}, \{a\}, \{b\}, \{b, c\}, \{a, b\}, \{a, c\}, \{a, b, c\}\}$$

The same works for multisets:

```
combinat::powerset(Dom::Multiset(a, a, b))
```

$$\{\varnothing, \{[a, 1]\}, \{[a, 2]\}, \{[b, 1]\}, \{[a, 1], [b, 1]\}, \{[a, 2], [b, 1]\}\}$$

Example 2

The powerset of a list `l` of pairwise different elements is the same as the powerset of the set of these elements, except that it consists of lists in which the order of elements is the same as in `l`:

```
combinat::powerset([c, a, b])
```

$$\{[], [a], [b], [c], [a, b], [c, a], [c, b], [c, a, b]\}$$

In general, the powerset of a list `l` is the same as the powerset of the multiset of its elements, except that it consists of lists in which the original order is preserved:

```
combinat::powerset([a, b, a])
```

$$\{[], [a], [b], [a, a], [a, b], [b, a], [a, b, a]\}$$

Parameters

s

Set

l

List

combinat::stirling1

Stirling numbers of the first kind

Syntax

```
combinat::stirling1(n, k)
```

Description

`combinat::stirling1(n,k)` computes the Stirling numbers of the first kind.

Let $S(n, k)$ be the number of permutations of `n` symbols that have exactly `k` cycles. Then `combinat::stirling1(n,k)` computes $(-1)^{(n+k)} S(n, k)$.

Let $S_1(n, k)$ be the Stirling number of the first kind, then we have:

$$\sum_{k=0}^{n} S_1(n, k)\, x^k = x(x-1)\cdots(x-n+1)$$

.

Examples

Example 1

Let us have a look what's the result of $x\,(x-1)\,(x-2)\,(x-3)\,(x-4)\,(x-5)$ written as a sum.

```
expand(x*(x-1)*(x-2)*(x-3)*(x-4)*(x-5))
```

$$x^6 - 15\,x^5 + 85\,x^4 - 225\,x^3 + 274\,x^2 - 120\,x$$

Now let us "prove" the formula mentioned in the "Details" section by calculating the proper Stirling numbers:

```
combinat::stirling1(6,1);
```

```
combinat::stirling1(6,2);
combinat::stirling1(6,3);
combinat::stirling1(6,4);
combinat::stirling1(6,5);
combinat::stirling1(6,6)
```

$$-120$$

$$274$$

$$-225$$

$$85$$

$$-15$$

$$1$$

Example 2

```
combinat::stirling1(3,-1)
```

```
Error: Nonnegative integers are expected. [combinat::stirling1]
```

Parameters

n, k

Nonnegative integers

Return Values

Integer.

References

J.J. Rotman, An Introduction to the Theory of Groups, 3rd Edition, Wm. C. Brown Publishers, Dubuque, 1988

combinat::stirling2

Stirling numbers of the second kind

Syntax

```
combinat::stirling2(n, k)
```

Description

`combinat::stirling2(n,k)` computes the number of ways of partitioning a set of `n` elements into `k` non-empty subsets.

`combinat::stirling2(n,k)` is calculated using the formula

$$\text{stirling2}(n, k) = \frac{1}{k!}\left(\sum_{j=0}^{k}(-1)^{k-j}\binom{k}{j}j^{n}\right).$$

Examples

Example 1

One can partition the set {1, 2, 3} into {1, 2, 3} = {1, 2} ∪ {3} = {1, 3} ∪ {2} = {2, 3} ∪ {1}

```
combinat::stirling2(3,2)
```

```
    3
```

Example 2

```
combinat::stirling2(3)
```

```
Error: Two arguments are expected. [combinat::stirling2]
```

Parameters

n, k

Nonnegative integers

Return Values

Nonnegative integer.

combinat::subwords

Subwords of a word

Syntax

```
combinat::subwords(w)
```

Description

The function `combinat::subwords(w)` returns a list of all subwords (sublists) of the list `w`.

A subword of a word w is a word obtained by deleting the letters at some of the positions in w. A subword is generated as many times as it appears in the word.

To obtain each subword only once, `combinat::powerset` should be used.

Examples

Example 1

There are 8 subwords of the word `[a, b, c]`:

```
combinat::subwords([a, b, c])
```

$$[[], [a], [b], [a, b], [c], [a, c], [b, c], [a, b, c]]$$

Parameters

`w`

List

6

daetools – Analyze and Reduce Differential Algebraic Equations (DAEs)

daetools::findDecoupledBlocks
daetools::incidenceMatrix
daetools::isLowIndexDAE
daetools::massMatrixForm
daetools::reduceDAEIndex
daetools::reduceDAEToODE
daetools::reduceDifferentialOrder
daetools::reduceRedundancies

daetools::findDecoupledBlocks

Search for decoupled blocks in systems of equations

Syntax

```
[eqsBlocks,varsBlocks] := daetools::findDecoupledBlocks(eqs,vars)
```

Description

`[eqsBlocks,varsBlocks] := daetools::findDecoupledBlocks(eqs,vars)` identifies subsets (blocks) of equations that can be used to define subsets of variables. The number of variables `vars` must coincide with the number of equations `eqs`.

The `i`th block is the set of equations determining the variables in `vars[varsBlocks[i]]`. The variables in `vars[varsBlocks[1],...,varsBlocks[i-1]]` are determined recursively by the previous blocks of equations. After you solve the first block of equations for the first block of variables, the second block of equations, given by `eqs[eqsBlocks[2]]`, defines a decoupled subset of equations containing only the subset of variables given by the second block of variables `vars[varsBlock[2]]`, plus the variables from the first block (these variables are known at this time). Thus, if a nontrivial block decomposition is possible, you can split the solution process for a large system of equations involving many variables into several steps, where each step involves a smaller subsystem.

The number of blocks is `nops(eqsBlocks)`. It coincides with `nops(varsBlocks)`. If `nops(eqsBlocks) = nops(varsBlocks) = 1`, then a nontrivial block decomposition of the equations is not possible.

The implemented algorithm requires that for each variable in `vars` there must be at least one matching equation in `eqs` involving this variable. The same equation cannot also be matched to another variable. If the system does not satisfy this condition, then `daetools::findDecoupledBlocks` throws an error. In particular, `daetools::findDecoupledBlocks` requires that `nops(eqs) = nops(vars)`.

Examples

Example 1

Compute a block lower triangular decomposition of a symbolic system of differential algebraic equations (DAEs).

Create the following system of four differential algebraic equations. Here, the expressions `x1(t)`, `x2(t)`, `x3(t)`, and `x4(t)` represent the state variables of the system. The system also contains symbolic parameters `c1`, `c2`, `c3`, `c4`, and the expressions `f(t,x,y)` and `g(t,x,y)`.

```
eqs := [c1*diff(x1(t),t) + c2*diff(x3(t),t) = c3*f(t,x1(t),x3(t)),
        c2*diff(x1(t),t) + c1*diff(x3(t),t) = c4*g(t,x3(t),x4(t)),
        x1(t) = g(t,x1(t),x3(t)),
        x2(t) = f(t,x3(t),x4(t))]:
vars:= [x1(t), x2(t), x3(t), x4(t)]:
```

Use `daetools::findDecoupledBlocks` to find the block structure of the system.

```
[eqsBlocks, varsBlocks] := daetools::findDecoupledBlocks(eqs, vars)
```

$$[[[1, 3], [2], [4]], [[1, 3], [4], [2]]]$$

The first block contains two equations in two variables.

```
eqs[eqsBlocks[1]]
```

$$\left[c1\,\frac{\partial}{\partial t}\,x1(t) + c2\,\frac{\partial}{\partial t}\,x3(t) = c3\,f(t, x1(t), x3(t)),\ x1(t) = g(t, x1(t), x3(t))\right]$$

```
vars[varsBlocks[1]]
```

$$[x1(t), x3(t)]$$

After you solve this block for the state variables `x1(t)`, `x3(t)`, you can solve the next block of equations. This block consists of one equation.

```
eqs[eqsBlocks[2]]
```

$$\left[c2\ \frac{\partial}{\partial t}\,x1(t) + c1\ \frac{\partial}{\partial t}\,x3(t) = c4\ g(t, x3(t), x4(t))\right]$$

This block involves one variable.

```
vars[varsBlocks[2]]
```

$$[x4(t)]$$

After you solve the equation from block 2 for the state variable `x4(t)`, the remaining block of equations `eqs[eqsBlocks[3]]` defines the remaining variable `vars[varsBlocks[3]]`.

```
eqs[eqsBlocks[3]];
vars[varsBlocks[3]]
```

$$[x2(t) = f(t, x3(t), x4(t))]$$

$$[x2(t)]$$

Find the permutations that convert the system to a block lower triangular form.

```
eqsPerm := [op(eqsBlocks[i]) $ i = 1..nops(eqsBlocks)];
varsPerm := [op(varsBlocks[i]) $ i = 1..nops(varsBlocks)]
```

$$[1, 3, 2, 4]$$

$$[1, 3, 4, 2]$$

Convert the system to a block lower triangular system of equations.

```
eqs := eqs[eqsPerm];
vars := vars[varsPerm]
```

$$\left[\mathrm{c1}\,\frac{\partial}{\partial t}\,\mathrm{x1}(t)+\mathrm{c2}\,\frac{\partial}{\partial t}\,\mathrm{x3}(t)=\mathrm{c3}\,f(t,\,\mathrm{x1}(t),\,\mathrm{x3}(t)),\right.$$

$$\mathrm{x1}(t)=g(t,\,\mathrm{x1}(t),\,\mathrm{x3}(t)),\;\mathrm{c2}\,\frac{\partial}{\partial t}\,\mathrm{x1}(t)+\mathrm{c1}\,\frac{\partial}{\partial t}\,\mathrm{x3}(t)=\mathrm{c4}\,g(t,\,\mathrm{x3}(t),\,\mathrm{x4}(t)),$$

$$\left.\mathrm{x2}(t)=f(t,\,\mathrm{x3}(t),\,\mathrm{x4}(t))\right]$$

$$[\mathrm{x1}(t),\,\mathrm{x3}(t),\,\mathrm{x4}(t),\,\mathrm{x2}(t)]$$

Find the incidence matrix of the resulting system. The incidence matrix shows that the system of permuted equations has three diagonal blocks of size 2-by-2, 1-by-1, and 1-by-1.

```
daetools::incidenceMatrix(eqs, vars)
```

$$\begin{pmatrix}1&1&0&0\\1&1&0&0\\1&1&1&0\\0&1&1&1\end{pmatrix}$$

Parameters

eqs

A list or a vector of equations or expressions in the state variables `vars` and their derivatives. Expressions represent equations with `0` right side.

vars

A list or a vector of identifiers or expressions, such as `[x1(t),x2(t)]`.

Return Values

A nested list of integers representing permutations required to convert the original system `eqs,vars` to a block lower triangular form.

See Also

MuPAD Functions

`daetools::incidenceMatrix` | `daetools::isLowIndexDAE` | `daetools::massMatrixForm` | `daetools::reduceDAEIndex` | `daetools::reduceDAEToODE` | `daetools::reduceDifferentialOrder` | `daetools::reduceRedundancies`

Introduced in R2014b

daetools::incidenceMatrix

Find incidence matrix of system of equations

Syntax

```
A := daetools::incidenceMatrix(eqs,vars)
```

Description

`A := daetools::incidenceMatrix(eqs,vars)` for `m` equations `eqs` and `n` variables `vars` returns an `m`-by-`n` matrix `A`, where `A[i,j] = 1` if `eqs[i]` contains `vars[j]` or any derivative of `vars[j]`. All other elements of `A` are `0`s.

Examples

Example 1

Find the incidence matrix of a system of five equations in five variables.

Create the following vector `eqs` containing five symbolic differential equations.

```
eqs := [diff(y1(t),t) = y2(t),
        diff(y2(t),t) = c1*y1(t) + c3*y3(t),
        diff(y3(t),t) = y2(t) + y4(t),
        diff(y4(t),t) = y3(t) + y5(t),
        diff(y5(t),t) = y4(t)]:
```

Create the vector of variables. Here, `c1` and `c3` are symbolic parameters (not variables) of the system.

```
vars := [y1(t), y2(t), y3(t), y4(t), y5(t)]:
```

Find the incidence matrix `A` for the equations `eqs` with respect to the variables `vars`.

```
A := daetools::incidenceMatrix(eqs, vars)
```

Parameters

`eqs`

A list or a vector of equations or expressions in the state variables `vars` and their derivatives. Expressions represent equations with `0` right side.

`vars`

A list or a vector of identifiers or expressions, such as `[x1(t), x2(t)]`.

Return Values

A matrix of `1`s and `0`s.

See Also

MuPAD Functions

`daetools::findDecoupledBlocks` | `daetools::isLowIndexDAE` | `daetools::massMatrixForm` | `daetools::reduceDAEIndex` | `daetools::reduceDAEToODE` | `daetools::reduceDifferentialOrder` | `daetools::reduceRedundancies`

Introduced in R2014b

daetools::isLowIndexDAE

Check if differential index of system equations is lower than 2

Syntax

```
daetools::isLowIndexDAE(eqs,vars)
```

Description

`daetools::isLowIndexDAE(eqs,vars)` checks if the system `eqs` of first-order semilinear differential algebraic equations (DAEs) has a low differential index. If the differential index of the system is `0` or `1`, `isLowIndexDAE` returns `TRUE`. If the differential index of `eqs` is higher than `1`, then `daetools::isLowIndexDAE` returns `FALSE`.

The number of equations `eqs` must match the number of variables `vars`.

Examples

Example 1

Check if a system of first-order semilinear DAEs has a low (`0` or `1`) differential index.

Create the following system of two differential algebraic equations. Here, `x(t)` and `y(t)` are the state variables of the system.

```
eqs := [diff(x(t),t) = x(t) + y(t), x(t)^2 + y(t)^2 = 1];
vars := [x(t), y(t)]
```

$$\left[\frac{\partial}{\partial t} x(t) = x(t) + y(t),\ x(t)^2 + y(t)^2 = 1\right]$$

$$[x(t),\ y(t)]$$

Use `daetools::isLowIndexDAE` to check the differential order of the system. The differential order of this system is `1`. For systems of index `0` and `1`, `daetools::isLowIndexDAE` returns `TRUE`.

```
daetools::isLowIndexDAE(eqs, vars)
```

```
TRUE
```

Example 2

Check if the following DAE system has a low or high differential index. If the index is higher than `1`, then use `daetools::reduceDAEIndex` to reduce it.

Create the following system of two differential algebraic equations. Here, `x(t)`, `y(t)`, and `z(t)` are the state variables of the system.

```
eqs := [diff(x(t),t) = x(t) + z(t),
        diff(y(t),t) = f(t),
        x(t) = y(t)];

vars := [x(t), y(t), z(t)]
```

$$\left[\frac{\partial}{\partial t}x(t) = x(t) + z(t),\ \frac{\partial}{\partial t}y(t) = f(t),\ x(t) = y(t)\right]$$

$$[x(t),\ y(t),\ z(t)]$$

Use `daetools::isLowIndexDAE` to check the differential index of the system. For this system, `daetools::isLowIndexDAE` returns `FALSE`. This means that the differential index of the system is `2` or higher.

```
daetools::isLowIndexDAE(eqs, vars)
```

```
FALSE
```

Use `daetools::reduceDAEIndex` to rewrite the system so that the differential index is `1`. The new system has one additional state variable, `Dyt(t)`.

```
[newEqs, newVars, transform, oldIndex] :=
```

```
daetools::reduceDAEIndex(eqs, vars):

newEqs;
newVars
```

$$[x'(t) - x(t) - z(t),\ \mathrm{Dyt}(t) - f(t),\ x(t) - y(t),\ x'(t) - \mathrm{Dyt}(t)]$$

$$[x(t),\ y(t),\ z(t),\ \mathrm{Dyt}(t)]$$

`daetools::reduceDAEIndex` also returns the differential index of the original system.

```
oldIndex
```

$$2$$

Check if the differential order of the new system is lower than 2.

```
daetools::isLowIndexDAE(newEqs, newVars)
```

$$\mathrm{TRUE}$$

Parameters

`eqs`

A list or a vector of equations or expressions in the state variables `vars` and their derivatives. Expressions represent equations with 0 right side.

`vars`

A list or a vector of identifiers or expressions, such as `[x1(t), x2(t)]`.

Return Values

TRUE or FALSE.

See Also

MuPAD Functions

`daetools::findDecoupledBlocks` | `daetools::incidenceMatrix` | `daetools::massMatrixForm` | `daetools::reduceDAEIndex` | `daetools::reduceDAEToODE` | `daetools::reduceDifferentialOrder` | `daetools::reduceRedundancies`

Introduced in R2014b

daetools::massMatrixForm

Extract mass matrix and right side of semilinear system of differential algebraic equations

Syntax

```
MF := daetools::massMatrixForm(eqs,vars)
```

Description

`MF := daetools::massMatrixForm(eqs,vars)` returns a list containing the mass matrix `M` and the right side of equations `F` of a semilinear system of first-order differential algebraic equations (DAEs). Algebraic equations in `eqs` that do not contain any derivatives of the variables in `vars` correspond to empty rows of the mass matrix `M`.

The mass matrix `M` and the right side of equations `F` refer to the form `M(t,x(t)x'(t)) = F(t,x(t))`.

Examples

Example 1

Convert a semilinear system of differential algebraic equations to mass matrix form.

Create the following system of differential algebraic equations. Here, `x1(t)` and `x2(t)` represent state variables of the system. The system also contains symbolic parameters `r` and `m`, and the parameter `f(t, x1(t), x2(t))`.

```
eqs :=
 [m*x2(t)*diff(x1(t), t) + m*t*diff(x2(t), t) = f(t, x1(t), x2(t)),
  x1(t)^2 + x2(t)^2 = r^2];

vars := [x1(t), x2(t)];
```

$$\left[m\,t\,\frac{\partial}{\partial t}\,\mathrm{x2}(t)+m\,\mathrm{x2}(t)\,\frac{\partial}{\partial t}\,\mathrm{x1}(t)=f(t,\mathrm{x1}(t),\mathrm{x2}(t)),\ \mathrm{x1}(t)^2+\mathrm{x2}(t)^2=r^2\right]$$

$$[\mathrm{x1}(t), \mathrm{x2}(t)]$$

Find the mass matrix form of this system.

```
MF := daetools::massMatrixForm(eqs, vars):
M := MF[1];
F := MF[2]
```

$$\begin{pmatrix} m\,\mathrm{x2}(t) & m\,t \\ 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} f(t, \mathrm{x1}(t), \mathrm{x2}(t)) \\ r^2 - \mathrm{x1}(t)^2 - \mathrm{x2}(t)^2 \end{pmatrix}$$

Parameters

`eqs`

A list or a vector of equations or expressions in the state variables `vars` and their derivatives. Expressions represent equations with `0` right side.

`vars`

A list or a vector of identifiers or expressions, such as `[x1(t), x2(t)]`.

Return Values

A list of two matrices. The first entry is the mass matrix. The number of rows is the number of equations in `eqs`, and the number of columns is the number of variables in `vars`. The second entry is an `n`-by-`1` matrix of the right side of equations, where `n` is the number of equations `eqs`.

See Also

MuPAD Functions

`daetools::findDecoupledBlocks` | `daetools::incidenceMatrix` | `daetools::isLowIndexDAE` | `daetools::reduceDAEIndex` |

`daetools::reduceDAEToODE` | `daetools::reduceDifferentialOrder` | `daetools::reduceRedundancies`

Introduced in R2014b

daetools::reduceDAEIndex

Convert system of first-order differential algebraic equations to equivalent system of differential index 1

Syntax

```
[newEqs,newVars,R,oldIndex] := daetools::reduceDAEIndex(eqs,vars)
```

Description

`[newEqs,newVars,R,oldIndex] := daetools::reduceDAEIndex(eqs,vars)` converts a high-index system of first-order differential algebraic equations `eqs` to an equivalent system `newEqs` of differential index `1`. The `daetools::reduceDAEIndex` function keeps the original equations and variables and introduces new variables and equations. It also returns the matrix R that expresses the new variables in newVars as derivatives of the original variables vars and the differential index oldIndex of the original system of DAEs, eqs.

After conversion, `daetools::reduceDAEIndex` checks the differential index of the new system by calling `daetools::isLowIndexDAE`. If the index of newEqs is `2` or higher, then `daetools::reduceDAEIndex` issues a warning.

The implementation of `daetools::reduceDAEIndex` uses the Pantelides algorithm. This algorithm reduces higher-index systems to lower-index systems by selectively adding differentiated forms of the original equations. The Pantelides algorithm can underestimate the differential index of a new system, and therefore, can fail to reduce the differential index to `1`. In this case, `daetools::reduceDAEIndex` issues a warning and returns the value of oldIndex as `UNKNOWN`. The `daetools::reduceDAEToODE` function uses more reliable, but slower Gaussian elimination. Note that `daetools::reduceDAEToODE` requires the DAE system to be semilinear.

Examples

Example 1

Reduce the differential index of a system that contains two second-order differential algebraic equations. Because the equations are second-order equations, first use `reduceDifferentialOrder` to rewrite the system to a system of first-order DAEs.

Create the following system of two second-order DAEs. Here, `x(t)`, `y(t)`, and `F(t)` are state variables of the system.

```
eqs := [diff(x(t), t, t) = -F(t)*x(t),
        diff(y(t), t, t) = -F(t)*y(t) - g,
        x(t)^2 + y(t)^2 = r^2];
vars := [x(t), y(t), F(t)]
```

$$\left[\frac{\partial^2}{\partial t^2}x(t) = -F(t)\,x(t),\ \frac{\partial^2}{\partial t^2}y(t) = -g - F(t)\,y(t),\ x(t)^2 + y(t)^2 = r^2\right]$$

$$[x(t),\ y(t),\ F(t)]$$

Rewrite this system so that all equations become first-order differential equations. The `daetools::reduceDifferentialOrder` function replaces the second-order DAE by two first-order expressions by introducing the new variables `Dxt(t)` and `Dyt(t)`.

```
[eqs, vars, R] := daetools::reduceDifferentialOrder(eqs, vars):
newEquations = eqs;
newVariables = vars;
relations = R
```

$$\text{newEquations} = [\text{Dxt}'(t) + F(t)\,x(t),\ \text{Dyt}'(t) + g + F(t)\,y(t),\ -r^2 + x(t)^2 + y(t)^2,\ \text{Dxt}(t) - x'(t),\ \text{Dyt}(t) - y'(t)]$$

$$\text{newVariables} = [x(t),\ y(t),\ F(t),\ \text{Dxt}(t),\ \text{Dyt}(t)]$$

$$\text{relations} = [\text{Dxt}(t) = x'(t),\ \text{Dyt}(t) = y'(t)]$$

Use `daetools::reduceDAEIndex` to rewrite the system so that the differential index is 1.

```
[eqs,vars,R,originalIndex] := daetools::reduceDAEIndex(eqs,vars):
newEquations = eqs;
newVariables = vars;
relations = R;
originalDAEIndex = originalIndex
```

$$\text{newEquations} = [\text{Dxtt}(t) + F(t)\,x(t),\ g + \text{Dytt}(t) + F(t)\,y(t),\ -r^2 + x(t)^2 + y(t)^2,$$
$$\text{Dxt}(t) - \text{Dxt1}(t),\ \text{Dyt}(t) - \text{Dyt1}(t),\ 2\,\text{Dxt1}(t)\,x(t) + 2\,\text{Dyt1}(t)\,y(t),$$
$$2\,y(t)\,\text{Dyt1}'(t) + 2\,\text{Dyt1}(t)^2 + 2\,\text{Dxt1}(t)^2 + 2\,\text{Dxt1t}(t)\,x(t),\ \text{Dxtt}(t) - \text{Dxt1t}(t),$$
$$\text{Dytt}(t) - \text{Dyt1}'(t),\ \text{Dyt1}(t) - y'(t)]$$

$$\text{newVariables} = [x(t),\ y(t),\ F(t),\ \text{Dxt}(t),\ \text{Dyt}(t),\ \text{Dytt}(t),\ \text{Dxtt}(t),\ \text{Dxt1}(t),\ \text{Dyt1}(t),\ \text{Dxt1t}(t)]$$

$$\text{relations} = [\text{Dytt}(t) = \text{Dyt}'(t),\ \text{Dxtt}(t) = \text{Dxt}'(t),\ \text{Dxt1}(t) = x'(t),\ \text{Dyt1}(t) = y'(t),\ \text{Dxt1t}(t) = x''(t)]$$

$$\text{originalDAEIndex} = 3$$

Use `daetools::reduceRedundancies` to shorten the system.

```
[eqs, vars, solvedEquations,
 constantVariables,
 replacedVariables,
 otherEquations] := daetools::reduceRedundancies(eqs, vars):
newEquations = eqs;
newVariables = vars;
```

$$\text{newEquations} = [\text{Dxtt}(t) + F(t)\,x(t),\ g + \text{Dytt}(t) + F(t)\,y(t),\ -r^2 + x(t)^2 + y(t)^2,$$
$$2\,\text{Dxt}(t)\,x(t) + 2\,\text{Dyt}(t)\,y(t),\ 2\,y(t)\,\text{Dyt}'(t) + 2\,\text{Dyt}(t)^2 + 2\,\text{Dxt}(t)^2 + 2\,\text{Dxtt}(t)\,x(t),$$
$$\text{Dytt}(t) - \text{Dyt}'(t),\ \text{Dyt}(t) - y'(t)]$$

$$\text{newVariables} = [x(t),\ y(t),\ F(t),\ \text{Dxt}(t),\ \text{Dyt}(t),\ \text{Dytt}(t),\ \text{Dxtt}(t)]$$

Parameters

`eqs`

A list or a vector of equations or expressions in the state variables `vars` and their derivatives. Expressions represent equations with `0` right side.

`vars`

A list or a vector of identifiers or expressions, such as `[x1(t), x2(t)]`.

Return Values

A nested list containing the following four lists: a list of new equations, a list of new variables, a list of relations between new and original variables, and the differential index of the original DAE system. If `daetools::reduceDAEIndex` fails to reduce the differential index to `1`, then it issues a warning and returns `UNKNOWN` instead of the differential index of the original DAE system.

See Also

MuPAD Functions

`daetools::findDecoupledBlocks` | `daetools::incidenceMatrix` | `daetools::isLowIndexDAE` | `daetools::massMatrixForm` | `daetools::reduceDAEToODE` | `daetools::reduceDifferentialOrder` | `daetools::reduceRedundancies`

Introduced in R2014b

daetools::reduceDAEToODE

Convert system of first-order quasilinear differential algebraic equations to equivalent system of differential index 0

Syntax

```
[newEqs,constraintEqs,oldIndex] = daetools::reduceDAEToODE(eqs,vars)
```

Description

`[newEqs,constraintEqs,oldIndex] := daetools::reduceDAEToODE(eqs,vars)` converts a high-index system of first-order semilinear algebraic equations `eqs` to an equivalent system of ordinary differential equations, `newEqs`. It also returns a vector of constraint equations and the differential index oldIndex of the original system of semilinear DAEs, eqs.

The differential index of the new system is `0`, that is, the Jacobian of `newEqs` with respect to the derivatives of the variables in `vars` is invertible.

The implementation of `daetools::reduceDAEToODE` is based on Gaussian elimination. This algorithm is more reliable than the Pantelides algorithm used by `daetools::reduceDAEIndex`, but it can be much slower.

The number of equations `eqs` must coincide with the number of variables `vars`.

Examples

Example 1

Check if a DAE system has a low (`0` or `1`) or high (`>1`) differential index. If the index is higher than `1`, then first try to reduce the index by using `daetools::reduceDAEIndex` and then by using `daetools::reduceDAEToODE`.

Create the following system of differential algebraic equations. Here, `x1(t)`, `x2(t)`, and `x3(t)` represent state variables of the system. The system also contains the expressions `q1(t)`, `q2(t)`, and `q3(t)` that do not represent state variables.

```
eqs := [diff(x2(t),t) = q1(t) - x1(t),
        diff(x3(t),t) = q2(t) - 2*x2(t) - t*(q1(t)-x1(t)),
        q3(t) - t*x2(t) - x3(t) = 0];
vars := [x1(t), x2(t), x3(t)]
```

$$\left[\frac{\partial}{\partial t}\,\mathrm{x2}(t) = \mathrm{q1}(t) - \mathrm{x1}(t),\ \frac{\partial}{\partial t}\,\mathrm{x3}(t) = \mathrm{q2}(t) - 2\,\mathrm{x2}(t) - t\,(\mathrm{q1}(t) - \mathrm{x1}(t)),\ \mathrm{q3}(t) - \mathrm{x3}(t) - t\,\mathrm{x2}(t) = 0\right]$$

$$[\mathrm{x1}(t),\ \mathrm{x2}(t),\ \mathrm{x3}(t)]$$

Use `daetools::isLowIndexDAE` to check the differential index of the system. For this system, `daetools::isLowIndexDAE` returns `FALSE`. This means that the differential index of the system is `2` or higher.

```
daetools::isLowIndexDAE(eqs, vars)
```

$$\mathrm{FALSE}$$

Use `daetools::reduceDAEIndex` as your first attempt to rewrite the system so that the differential index is `1`. For this system, `daetools::reduceDAEIndex` issues a warning because it cannot reduce the differential index of the system to `0` or `1`.

```
[newEqs, newVars, R, oldIndex] :=
    daetools::reduceDAEIndex(eqs, vars):

newEquations = newEqs;
newVariables = newVars;
relations = R;
originalIndex = oldIndex
```

```
Warning: The index of the reduced DAEs is larger than 1. [daetools::reduceDAEIndex]
```

$$\mathrm{newEquations} = [\mathrm{x2}'(t) - \mathrm{q1}(t) + \mathrm{x1}(t),\ \mathrm{Dx3t}(t) - \mathrm{q2}(t) + 2\,\mathrm{x2}(t) + t\,(\mathrm{q1}(t) - \mathrm{x1}(t)),\ \mathrm{q3}(t) - \mathrm{x3}(t) - t\,\mathrm{x2}(t),\ \mathrm{q3}'(t) - t\,\mathrm{x2}'(t) - \mathrm{x2}(t) - \mathrm{Dx3t}(t)]$$

$$\text{newVariables} = [\text{x1}(t), \text{x2}(t), \text{x3}(t), \text{Dx3t}(t)]$$

$$\text{relations} = [\text{Dx3t}(t) = \text{x3}'(t)]$$

$$\text{originalIndex} = \text{UNKNOWN}$$

If `daetools::reduceDAEIndex` cannot reduce the semilinear system so that the index is `0` or `1`, try using `daetools::reduceDAEToODE`. This function can be much slower, therefore it is not recommended as a first choice.

```
[newEqs, constraintEqs, oldIndex] :=
    daetools::reduceDAEToODE(eqs, vars):

ODEs = eqs;
constraintEquations = constraintEqs;
originalIndex = oldIndex
```

$$\text{ODEs} = \left[\frac{\partial}{\partial t}\,\text{x2}(t) = \text{q1}(t) - \text{x1}(t),\ \frac{\partial}{\partial t}\,\text{x3}(t) = \text{q2}(t) - 2\,\text{x2}(t) - t\,(\text{q1}(t) - \text{x1}(t)),\right.$$
$$\left.\text{q3}(t) - \text{x3}(t) - t\,\text{x2}(t) = 0\right]$$

$$\text{constraintEquations} = [\text{q2}'(t) - \text{q3}''(t) - \text{q1}(t) + \text{x1}(t),\ \text{x3}(t) - \text{q3}(t) + t\,\text{x2}(t),$$
$$\text{q3}'(t) - \text{q2}(t) + \text{x2}(t)]$$

$$\text{originalIndex} = 3$$

Parameters

eqs

A list or a vector of equations or expressions in the state variables `vars` and their derivatives. Expressions represent equations with `0` right side.

`vars`

A list or a vector of identifiers or expressions, such as `[x1(t), x2(t)]`.

Return Values

A nested list containing the following three lists: a list of ordinary differential equations, a list of constraint equations encountered during system reduction, and the differential index of the original DAE system.

See Also

MuPAD Functions

`daetools::findDecoupledBlocks` | `daetools::incidenceMatrix` | `daetools::isLowIndexDAE` | `daetools::massMatrixForm` | `daetools::reduceDAEIndex` | `daetools::reduceDifferentialOrder` | `daetools::reduceRedundancies`

Introduced in R2014b

daetools::reduceDifferentialOrder

Reduce systems of higher-order differential equations to systems of first-order differential equations

Syntax

```
[newEqs,newVars,R] := daetools::reduceDifferentialOrder(eqs,vars)
```

Description

`[newEqs,newVars,R] := daetools::reduceDifferentialOrder(eqs,vars)` rewrites a system of higher-order differential equations eqs as a system of first-order differential equations `newEqs` by substituting derivatives in `eqs` with new variables. It also returns the matrix `R` that expresses the new variables in `newVars` as derivatives of the original variables `vars`. Here, `newVars` consists of the original variables `vars` augmented with these new variables.

Examples

Example 1

Reduce a system containing a second- and a third-order expression to a system containing only first-order DAEs, and return a matrix that expresses the variables generated by `daetools::reduceDifferentialOrder` via the original variables of this system.

Create the following system of differential equations, which includes a second- and a third-order expression. Here, `x(t)` and `y(t)` are state variables of the system.

```
eqs := [diff(x(t),t,t) = diff(f(t),t,t,t),
        diff(y(t),t,t,t) = diff(f(t),t,t)];
vars := [x(t), y(t)]
```

$$\left[\frac{\partial^2}{\partial t^2}x(t) = \frac{\partial^3}{\partial t^3}f(t),\ \frac{\partial^3}{\partial t^3}y(t) = \frac{\partial^2}{\partial t^2}f(t)\right]$$

$$[x(t),\ y(t)]$$

Rewrite this system so that all equations become first-order differential equations. The `daetools::reduceDifferentialOrder` function replaces the higher-order DAE with first-order equations by introducing the new variables `Dxt(t)`, `Dyt(t)`, and `Dytt(t)`. This function returns a nested list containing the following three lists: a list of new equations, a list of new variables, and a list of relations between the new and the original variables. Display `newEqs`, `newVars`, and `R` separately.

```
[newEqs, newVars, R] :=
    daetools::reduceDifferentialOrder(eqs, vars):

newEqs;
newVars;
R
```

$$[\mathrm{Dxt}'(t) - f''''(t),\ \mathrm{Dytt}'(t) - f'''(t),\ \mathrm{Dxt}(t) - x'(t),\ \mathrm{Dyt}(t) - y'(t),\ \mathrm{Dytt}(t) - \mathrm{Dyt}'(t)]$$

$$[x(t),\ y(t),\ \mathrm{Dxt}(t),\ \mathrm{Dyt}(t),\ \mathrm{Dytt}(t)]$$

$$[\mathrm{Dxt}(t) = x'(t),\ \mathrm{Dyt}(t) = y'(t),\ \mathrm{Dytt}(t) = y''(t)]$$

Parameters

eqs

A list or a vector of equations or expressions in the state variables `vars` and their derivatives. Expressions represent equations with `0` right side.

vars

A list or a vector of identifiers or expressions, such as `[x1(t), x2(t)]`.

Return Values

A nested list containing the following three lists: a list of new equations, a list of new variables, and a list of relations between the new and the original variables.

See Also

MuPAD Functions

`daetools::findDecoupledBlocks` | `daetools::incidenceMatrix` | `daetools::isLowIndexDAE` | `daetools::massMatrixForm` | `daetools::reduceDAEIndex` | `daetools::reduceDAEToODE` | `daetools::reduceRedundancies`

Introduced in R2014b

daetools::reduceRedundancies

Simplify system of first-order differential algebraic equations by eliminating redundant equations and variables

Syntax

```
[newEqs,
 newVars,
 solvedEquations,
 constantVariables,
 replacedVariables,
 otherEquations
] := daetools::reduceRedundancies(eqs, vars)
```

Description

`[newEqs, newVars, solvedEquations, constantVariables, replacedVariables, otherEquations] := daetools::reduceRedundancies(eqs, vars)` eliminates simple equations from the system of first-order differential algebraic equations `eqs`. It returns a list of remaining equations, a list of remaining variables, and four more lists containing information on the eliminated equations and variables. For details, see “Return Values” on page 6-29.

Examples

Example 1

Use `daetools::reduceRedundancies` to simplify a system of five differential algebraic equations in four variables to a system of two equations in two variables.

Create the following system of five differential algebraic equations for four state variables: `x1(t)`, `x2(t)`, `x3(t)`, and `x4(t)`. This system also contains symbolic parameters `a1`, `a2`, `a3`, `a4`, `b`, `c`, and a parameter function `f(t)` that is not a state variable.

```
eqs := [a1*diff(x1(t),t)+a2*diff(x2(t),t) = b*x4(t),
```

```
        a3*diff(x2(t),t)+a4*diff(x3(t),t) = c*x4(t),
        x1(t) = 2*x2(t),
        x4(t) = f(t),
        f(t) = sin(t)];
vars := [x1(t), x2(t), x3(t), x4(t)]
```

$$\left[a1\ \frac{\partial}{\partial t}\,x1(t) + a2\ \frac{\partial}{\partial t}\,x2(t) = b\ x4(t),\ a3\ \frac{\partial}{\partial t}\,x2(t) + a4\ \frac{\partial}{\partial t}\,x3(t) = c\ x4(t),\ x1(t) = 2\ x2(t),\right.$$

$$\left.x4(t) = f(t),\ f(t) = \sin(t)\right]$$

$$[x1(t),\ x2(t),\ x3(t),\ x4(t)]$$

Use `daetools::reduceRedundancies` to eliminate redundant equations and corresponding state variables.

```
[newEqs, newVars,
 solvedEquations,
 constantVariables,
 replacedVariables,
 otherEquations
] := daetools::reduceRedundancies(eqs, vars):
```

Display the new equations and new variables.

```
newEqs;
newVars
```

$$\left[a1\ x1'(t) - b\ f(t) + \frac{a2\ x1'(t)}{2},\ \frac{a3\ x1'(t)}{2} + a4\ x3'(t) - c\ f(t)\right]$$

$$[x1(t),\ x3(t)]$$

Display the equations that `daetools::reduceRedundancies` used to replace those state variables from `vars` that do not appear in `newEqs`.

```
solvedEquations
```

$$[x1(t) - 2\,x2(t),\ x4(t) - f(t)]$$

Display those state variables from `vars` that `daetools::reduceRedundancies` replaced by constant values.

```
constantVariables
```

$$[x4(t) = f(t)]$$

Display those state variables from `vars` that `daetools::reduceRedundancies` replaced by expressions in terms of other variables.

```
replacedVariables
```

$$\left[x2(t) = \frac{x1(t)}{2}\right]$$

Display those equations from `eqs` that do not contain any of the state variables `vars`.

```
otherEquations
```

$$[f(t) - \sin(t)]$$

Parameters

`eqs`

A list or a vector of equations or expressions in the state variables `vars` and their derivatives. Expressions represent equations with `0` right side.

`vars`

A list or a vector of identifiers or expressions, such as `[x1(t), x2(t)]`.

Return Values

A nested list containing the following lists:

- A list of new equations
- A list of those variables that remain in the new DAE system
- A list of equations that do not appear in `newEqs`
- A list of equations `[y1 = value1(t), y2 = value2(t),...]` defining those of the variables `[y1,y2,...]` (contained in the original equations `eqs` and the original `vars`) that were eliminated from `eqs`. In `newEqs`, they are replaced by the values.
- A list of equations `[y1 = Y1(t,x,diff(x,t),...), y2 = Y2(t,x,diff(x,t),...),...]` defining those of the variables `]y1,y2,...]` (in the original `vars`) that were eliminated in terms of the variables that are still in `newVars`. (Typically, equations involving only two variables are used to eliminate one of the variables.)
- A list of equations that do not contain any of the variables. These equations do not appear in `newEqs`.

See Also

MuPAD Functions

`daetools::findDecoupledBlocks` | `daetools::incidenceMatrix` | `daetools::isLowIndexDAE` | `daetools::massMatrixForm` | `daetools::reduceDAEIndex` | `daetools::reduceDAEToODE` | `daetools::reduceDifferentialOrder`

Introduced in R2014b

Dom – Domains

DOM_ARRAY
DOM_BOOL
DOM_COMPLEX
DOM_DOMAIN
DOM_EXEC
DOM_EXPR
DOM_FLOAT
DOM_FUNC_ENV
DOM_HFARRAY
DOM_IDENT
DOM_INT
DOM_INTERVAL
DOM_LIST
DOM_PROC
DOM_PROC_ENV
DOM_RAT
DOM_SET
DOM_STRING
DOM_VAR
Dom::AlgebraicExtension
Dom::ArithmeticalExpression
Dom::BaseDomain
Dom::Complex
Dom::DenseMatrix
Dom::DihedralGroup
Dom::DistributedPolynomial
Dom::Expression
Dom::ExpressionField
Dom::Float
Dom::FloatIV
Dom::Fraction
Dom::GaloisField

Dom::ImageSet
Dom::Integer
Dom::IntegerMod
Dom::Interval
Dom::LinearOrdinaryDifferentialOperator
Dom::Matrix
Dom::MatrixGroup
Dom::MonomOrdering
Dom::Multiset
Dom::MultivariatePolynomial
Dom::Natural
Dom::Numerical
Dom::Polynomial
Dom::Product
Dom::Quaternion
Dom::Rational
Dom::Real
Dom::SquareMatrix
Dom::SymmetricGroup
Dom::UnivariatePolynomial
Factored
Series::Puiseux
Series::gseries

DOM_ARRAY

(symbolic, multidimensional) arrays

Description

DOM_ARRAY is a multidimensional container type, storing arbitrary MuPAD objects at integer indices.

Arrays are a fundamental data type in many programming languages: For a fixed number of indices ("dimensions"), for each index an integer from a fixed range, an array provides space to store an arbitrary piece of data at this combination.

Function Calls

Using an array as a function symbol creates the list obtained by using each array entry as a function symbol for the operands used, i.e., `array(1..2, [f, g])(x, y)` results in `array(1..2, [f(x, y), g(x, y)])`.

Operations

As with any container, the most important operation on an array is reading and writing its entries, which is performed by indexed access, as in `A[1, 2]` or `B[1, 3, 2] := exp(x)`. Trying to access an element outside the boundaries of an array raises an error.

The function `map` applies some function or transformation to each element of an array, returning an array of the same format as its input, with the results of the calls as its entries.

If `A` is an array, `nops(A)` returns the number of elements in `A`.

Operands

If `A` is an array, the 0th operand of `A`, `op(A, 0)`, will be the sequence starting with the number of dimensions (an integer n) followed by n ranges of integers, which denote the

acceptable ranges of indices for each dimension, including both numbers listed in the range.

For $1 \le i \le nops(A)$, the ith operand of A is the ith entry of A, in the lexicographic order of indices.

Uninitialized entries of arrays will be displayed symbolically while still in the array. When being accessed by `op` or indexed access, `NIL` is returned.

Output

One-dimensional arrays are displayed as row vectors, two-dimensional arrays as matrices. Higher-dimensional arrays are written in functional form, using the `index = value` notation, and do not have a typesetting version. This also causes typesetting to be disabled for any surrounding expression in the same output.

Element Creation

The primary way of creating arrays is the function `array`. Beside that, obviously, `coerce` can convert a number of data types, such as matrices into arrays and a number of MuPAD functions, especially in the `numeric` library, return arrays.

See Also

MuPAD Domains
`DOM_HFARRAY` | `DOM_LIST` | `DOM_TABLE`

MuPAD Functions
`array` | `matrix`

DOM_BOOL

Boolean constants

Description

DOM_BOOL is the data type of the truth values TRUE, FALSE, and UNKNOWN.

MuPAD uses a three-valued logic system, with these values (constants) TRUE, FALSE, and UNKNOWN.

Function Calls

Using a Boolean constant as a function returns that constant unchanged. The arguments of the call are *not* evaluated.

Operations

The most important operations on Boolean values are the logical operators `and`, `not`, `or`, `xor`, `==>`, `<=>`, and using them in conditions of `if` or `piecewise`.

Operands

Boolean constants are atomic.

Output

TRUE is displayed as *TRUE*, FALSE is displayed as *FALSE*, and UNKNOWN is shown as *UNKNOWN*.

Element Creation

The three constants can be types in as shown above. Additionally, many MuPAD functions returns Boolean values, the most generic/prominent two being `bool` and `is`.

See Also

MuPAD Functions
`bool` | `if` | `is`

DOM_COMPLEX

(Simple) Complex Numbers

Description

DOM_COMPLEX is the type of complex numbers with integer, rational, or floating-point components.

Complex numbers of type DOM_COMPLEX have two operands, their real and imaginary part. These are objects of type DOM_FLOAT, DOM_INT, or DOM_RAT. Complex numbers with other components (such as $\sqrt{2} + i\,\pi$) are not of domain type DOM_COMPLEX, but DOM_EXPR.

Function Calls

Calling a complex number as a function returns that number unchanged. The arguments of the call are *not* evaluated.

Operations

Most MuPAD functions operate on complex numbers. Use Re and Im to access the real and imaginary part, respectively.

Operands

Every object of type DOM_COMPLEX has two operands, the real and the imaginary part.

Output

Objects of type DOM_COMPLEX are essentially written as expressions in rectangular form. The imaginary unit is displayed as *I*.

Element Creation

Complex numbers can be constructed by typing in the corresponding expression, such as `3+4*I`. The keyword for typing the imaginary unit I is `I` (a capital letter i).

See Also

MuPAD Domains
`DOM_FLOAT` | `DOM_INT` | `DOM_RAT`

DOM_DOMAIN

Data type of data types

Description

`DOM_DOMAIN` is the data type of datatypes.

Each MuPAD object has a unique data type. Since a data type is a MuPAD object, too, it must itself have a data type; the data type comprising all data types (including itself) is `DOM_DOMAIN`.

There are two kinds of elements of `DOM_DOMAIN`: data types of the kernel, and data types defined in the library or by the user (*domains*). Objects that have a data type of the latter kind are called *domain elements*.

A data type has the same internal structure as a table; its entries are called slots. One particular slot is the *key*; no two different data types can have the same key. Most of the other slots determine how arguments of that data type are handled by functions.

Once a user-defined domain has been constructed, it cannot be destroyed.

Examples

Example 1

Our first example stems from ethnology: some languages in Polynesia do not have words for numbers greater than three; every integer greater than three is denoted by the word “many”. Hence two plus two does not equal four but “many”. We are going to implement a domain for this kind of integers; in other words, we are going to implement a data type for the finite set {1, 2, 3, *many*}.

```
S := newDomain("Polynesian integer")
```

```
Polynesian integer
```

At this point, we have defined a new data type: a MuPAD object can be a Polynesian integer now. No operations are available yet; the domain consists of its key only:

```
op(S)
```

```
"key" = "Polynesian integer"
```

Even though there are no methods for input and output of domain elements yet, Polynesian integers can be entered and displayed right now. You have to use the function `new` for defining domain elements:

```
x := new(S, 5)
```

```
new(Polynesian integer, 5)
```

Now, `x` is a Polynesian integer:

```
type(x)
```

```
Polynesian integer
```

Of course, MuPAD cannot know what meaning a Polynesian integer has and what its internal structure should be. The arguments of the call to the function `new` are just stored as the zeroth, first, etc. operand of the domain element, without checking them. You may call `new` with as many arguments as you want:

```
new(S, 1, 2, 3, 4); op(%)
```

```
new(Polynesian integer, 1, 2, 3, 4)
```

```
1, 2, 3, 4
```

`new` cannot know that Polynesian integers should have exactly one operand and that we want `5` to be replaced by `many`. To achieve this, we implement our own method `"new"`; this also allows us to check the argument. We have one more problem: domain methods should refer to the domain; but they should not depend on the fact that the domain is currently stored in `S`. For this purpose, MuPAD has a special local variable `dom` that always refers to the domain a procedure belongs to:

```
S::new :=
proc(i : Type::PosInt)
begin
```

```
  if args(0) <> 1 then
    error("There must be exactly one argument")
  end_if;
  if i > 3 then
    new(dom, hold(many))
  else
    new(dom, i)
  end_if
end_proc:
```

A function call to the domain such as `S(5)` now implicitly calls the `"new"` method:

```
S(5)
```

```
new(Polynesian integer, many)
```

```
S("nonsense")
```

```
Error: The object '"nonsense"' is incorrect. The type of argument number 1 must be 'Ty
  Evaluating: S::new
```

In the next step, we define our own output method. A Polynesian integer `i`, say, shall not be printed as `new(Polynesian integer, i)`, only its internal value `1`, `2`, `3`, or `many` shall appear on the screen. Note that this value is the first operand of the data structure:

```
S::print :=
proc(x)
begin
  op(x, 1)
end_proc:
S(1), S(2), S(3), S(4), S(5)
```

```
1, 2, 3, many, many
```

By now, the input and output of elements of `S` have been defined. It remains to define how the functions and operators of MuPAD should react to Polynesian integers. This is done by *overloading* them. However, it is not necessary to overload each of the thousands of functions of MuPAD; for some of them, the default behavior is acceptable. For example, expression manipulation functions leave domain elements unaltered:

```
x := S(5): expand(x), simplify(x), combine(x); delete x:
```

```
many, many, many
```

Arithmetical operations handle domain elements like identifiers; they automatically apply the associative and commutative law for addition and multiplication:

```
(S(3) + S(2)) + S(4)
```

```
2 + 3 + many
```

In our case, this is not what we want. So we have to overload the operator +. Operators are overloaded by overloading the corresponding "underline-functions"; hence, we have to write a method `"_plus"`:

```
S::_plus :=
proc()
local argv;
begin
  argv := map([args()], op, 1);
  if has(argv, hold(many)) then
     new(dom, hold(many))
  else
     dom(_plus(op(argv)))
  end_if
end_proc:
```

Now, the sum of Polynesian integers calls this slot:

```
S(1) + S(2), S(2) + S(3) + S(7)
```

```
3, many
```

Deleting the identifier `S` does not destroy our domain. It can still be reconstructed using `newDomain`.

```
delete S:
op(newDomain("Polynesian integer"))
```

```
"key" = "Polynesian integer", "new" = proc S::new(i) ... end, "print" = proc S::print(x) ... end,
  "_plus" = proc S::_plus() ... end
```

Example 2

We could now give a similar example for more advanced Polynesian mathematics with numbers up to ten, say. This leads to the question whether it is necessary to enter all the code again and again whenever we decide to count a bit farther. It is not; this is one of the advantages of *domain constructors*. A domain constructor may be regarded as a function that returns a domain depending on some input parameters. It has several additional features. Firstly, the additional keywords `category` and `axiom` are available for specifying the mathematical structure of the domain; in our case, we have the structure of a commutative semigroup where different domain elements have different mathematical meanings (we call this a domain with a canonical representation). Secondly, an initialization part may be defined that is executed exactly once for every domain returned by the constructor; it should at least check the parameters passed to the constructor. Each domain created in such a way may inherit methods from other domains, and it must at least inherit the methods of `Dom::BaseDomain`.

```
domain CountingUpTo(n : Type::PosInt)
  inherits Dom::BaseDomain;
  category Cat::AbelianSemiGroup;
  axiom Ax::canonicalRep;

  new := proc(x : Type::PosInt)
  begin
    if args(0) <> 1 then
      error("There must be exactly one argument")
    end_if;
    if x > n then
      new(dom, hold(many))
    else
      new(dom, x)
    end_if
  end_proc;

  print := proc(x) begin op(x, 1) end_proc;

  _plus := proc() local argv;
  begin
    argv:= map([args()], op, 1);
    if has(argv, hold(many)) then
       new(dom, hold(many))
    else
       dom(_plus(op(argv)))
    end_if
```

```
  end_proc;

// initialization part
begin
  if args(0) <> 1 then
    error("Wrong number of arguments")
  end_if;
end:
```

Now, `CountingUpTo` is a domain constructor:

```
type(CountingUpTo)
```

DomainConstructor

We have defined the domain constructor `CountingUpTo`, but we have not created a domain yet. This is done by calling the constructor:

```
CountingUpToNine := CountingUpTo(9);
CountingUpToTen := CountingUpTo(10)
```

CountingUpTo(9)

CountingUpTo(10)

We are now able to create, output, and manipulate domain elements as in the previous example:

```
x := CountingUpToNine(3): y := CountingUpToNine(7):
x, x + x, y, x + y, y + y
```

3, 6, 7, many, many

```
x := CountingUpToTen(3): y := CountingUpToTen(7):
x, x + x, y, x + y, y + y
```

3, 6, 7, 10, many

```
delete CountingUpToNine, CountingUpToTen, CountingUpTo, x, y:
```

No domain constructor with the same name may be used again during the same session.

Example 3

Suppose that your domain does not really depend on a parameter, but that you need some of the other features of domain constructors. Then you may define a domain constructor `dc`, say, that is called without parameters. From such a domain constructor, you can construct exactly one domain `dc()`. Instead of defining the constructor via `domain dc() ... end` first and then using `d := dc()` to construct the domain `d`, say, you may directly enter `domain d ... end`, thereby saving some work.

Continuing the previous examples, suppose that we want to count up to three, knowing that we never want to count farther. However, we want to declare our domain to be an Abelian semigroup with a canonical representation of the elements. This is not possible with a construction of the domain using `newDomain` as in "Example 1" on page 7-9: we have to use the keyword domain. You will notice at once that the following source code is almost identical to the one in the previous example—we just removed the dependence on the parameter `n`.

```
domain CountingUpToThree

inherits Dom::BaseDomain;
category Cat::AbelianSemiGroup;
axiom Ax::canonicalRep;

new := proc(x : Type::PosInt)
begin
  if args(0) <> 1 then
    error("There must be exactly one argument")
  end_if;
  if x > 3 then
    new(dom, hold(many))
  else
    new(dom, x)
  end_if
end_proc;

print := proc(x) begin op(x, 1) end_proc;

_plus := proc() local argv;
begin
  argv:= map([args()], op, 1);
  if has(argv, hold(many)) then
     new(dom, hold(many))
  else
```

```
    dom(_plus(op(argv)))
  end_if
end_proc;

end:
```

Now, `CountingUpToThree` is a domain and not a domain contructor:

```
type(CountingUpToThree)
```

```
DOM_DOMAIN
```

You may use this domain in the same way as `CountingUpTo(3)` in "Example 2" on page 7-13.

Function Calls

When called as a function, the data type creates a new object of this data type out of the arguments of the call. E.g., the call `DOM_LIST(1, 2, x)` generates the list `[1, 2, x]` of domain type `DOM_LIST` (although, in this case, you probably prefer to type in `[1, 2, x]` directly which results in the same object). It depends on the particular type which arguments are admitted here.

In the case of a domain, the `"new"` method of that domain is called.

Operations

You can obtain the slots of a domain using `slot`. The function `slot` can also be used on the left hand side of an assignment to define new slots, or to re-define existing slots. Use `delete` to delete slots.

Operands

A data type consists of an arbitrary number of equations (objects of type `"equal"`). If `a = b` is among these equations, we say that the *slot*`a` of the data type equals `b`. By convention, `a` is usually a string. Each domain has at least one slot indexed by `"key"`.

Element Creation

The names of the data types provided by the MuPAD kernel are of the form `DOM_XXX`, such us `DOM_ARRAY`, `DOM_HFARRAY`, `DOM_IDENT`, `DOM_INT`, `DOM_LIST`, `DOM_TABLE` etc.

You can create further data types using the function `newDomain` (cf. “Example 1” on page 7-9) or via the keyword domain (cf. “Example 3” on page 7-15).

You can also create new data types by calling a *domain constructor*. Various pre-defined domain constructors can be found in the library Dom. You can also define your own domain constructors using the keyword domain. Cf. “Example 2” on page 7-13.

The domain type (data type) of any MuPAD object can be queried by the function `domtype`.

Algorithms

Only one domain with a given key may exist. If it is stored in two variables `S` and `T`, say, assigning or deleting a slot `slot(S, a)` implicitly also changes `slot(T, a)` (reference effect). This also holds if `a = "key"`.

Note: You get no warning even if `T` is protected.

DOM_EXEC

Kernel functions

Description

Objects of type DOM_EXEC represent kernel functions implemented in C++.

Unlike functions defined at the library level (which are stored in objects of type DOM_PROC), functions defined in C++ in the MuPAD kernel are represented by objects of type DOM_EXEC.

Users normally need not care about DOM_EXECs except for the cases where explicitly testing the domtype of arguments; in those cases, DOM_EXEC should often be treated identically to DOM_PROC.

Most kernel functions are actually stored inside function environments of type DOM_FUNC_ENV, and therefore, you can see DOM_EXEC only when explicitly accessing the first or second operand of those function environments.

Function Calls

An object of type DOM_EXEC essentially represents a function; using it in this way calls the corresponding function.

Operands

The operands of a DOM_EXEC are used internally, may change at any time and remain undocumented.

See Also

MuPAD Domains
DOM_FUNC_ENV | DOM_PROC

DOM_EXPR

Type of “general expressions”

Description

DOM_EXPR is the data type of symbolic function calls. This includes expressions such as $a + b$ which is internally stored as `_plus(a, b)`.

In MuPAD, non-atomic symbolic expressions which are not elements of special domains have type DOM_EXPR.

Objects of type DOM_EXPR have a 0th operand which contains the functor (the function symbol, the f in $f(x)$). This operand is not counted in the result of nops. The subsequent operands can be of arbitrary type (although most functions will limit the number and type of operands when evaluated).

The 0th operand of a DOM_EXPR will be a procedure or function environment only in exceptional circumstances. In usual circumstances, expressions only have expressions, domain elements, or identifiers as their 0th operands.

Examples

Example 1

Function calls are of type DOM_EXPR:

```
domtype(sin(x))
```

```
DOM_EXPR
```

The 0th operand of a function call is the function symbol:

```
op(sin(x), 0)
```

```
sin
```

This operand is taken into account neither by nops nor by op if called with one argument:

```
nops(sin(x)), op(sin(x))
```

$$1, x$$

Function Calls

The effect of using an expression of type DOM_EXPR as a function to call depends on the 0th operand of the expression. For many system functions, the result is that of using all operands of the expression as functions, passing the *unevaluated* arguments. (These functions may in turn evaluate their arguments.)

Operations

Most MuPAD functions (documented as accepting "arithmetical expressions") are built to work on elements of type DOM_EXPR.

Often, the operands of an expression will be expressions themselves. This creates a so-called "expression tree" which can be visualized using `prog::exprtree`.

Operands

All expressions are internally represented as function calls. The 0th operand is the function symbol of this call.

Evaluation

Evaluating an expression results in calling the 0th operand as a function. For library functions without `option hold`, the operands are evaluated first.

See Also

MuPAD Domains
DOM_BOOL | DOM_FLOAT | DOM_IDENT | DOM_INT | DOM_INTERVAL | DOM_LIST | DOM_POLY | DOM_RAT | DOM_SET | DOM_TABLE

MuPAD Functions
`prog::exprtree`

DOM_FLOAT

Real Floating Point Numbers

Description

DOM_FLOAT is the type of (arbitrary precision) real floating-point numbers.

Apart from exact symbolic calculations, MuPAD can also compute numerical approximations with arbitrary precision.

MuPAD uses the values RD_INF and RD_NINF for real positive and negative infinities in floating-point intervals.

MuPAD uses the value RD_NAN to indicate undefined values in floating-point intervals. If you use typeset mode, MuPAD displays this value as NaN in output regions.

Function Calls

Calling a floating-point number as a function returns the number unchanged. The arguments of the call are *not* evaluated.

Operations

Just about any arithmetical operation can be performed with floating-point numbers.

Operands

DOM_FLOATs are atomic.

Output

The output format of DOM_FLOAT depends on the setting of Pref::floatFormat and is documented there.

Element Creation

Floating point numbers are typed in with an optional sign (an arbitrary number of `+` and `-` signs), an optional integer part (consisting of digits), a decimal point (irrespective of locale settings of the operating system, MuPAD always expects a decimal point), a fractional part (one or more decimal digits) and optionally a decimal shift, written as the letter `e` followed by an optionally signed integer.

The decimal shift is interpreted as a power of ten, i.e., `6.022e23` is the Avogadro number $6.022\ 10^{23}$.

Additionally, the function `float` and most calls to functions of the `numeric` library create floating-point numbers as well.

See Also

MuPAD Domains
`DOM_COMPLEX` | `DOM_INTERVAL` | `DOM_RAT`

MuPAD Functions
`DIGITS` | `float`

DOM_FUNC_ENV

Data type of function environments

Description

DOM_FUNC_ENV is the data type of function environments.

MuPAD uses function environments (domain type DOM_FUNC_ENV) to integrate functions into the system. All MuPAD library functions and most kernel functions are implemented as function environments.

A function environment stores special function attributes (slots) in an internal table. When an overloadable system function, such as `diff`, `expand`, or `float`, encounters an object of type DOM_FUNC_ENV, it searches the function environment for a corresponding slot.

Operands

A function environment consists of three operands. The first operand is a procedure that computes the return value of a function call. The second operand is a procedure for printing a symbolic function call on the screen. The third operand is a table that specifies how the system functions handle symbolic function calls.

Element Creation

`funcenv` and `fp::unapply` (or its equivalent `-->`) create function environments of type DOM_FUNC_ENV.

See Also

MuPAD Domains
DOM_EXEC | DOM_PROC

MuPAD Functions
`fp::unapply` | `funcenv`

DOM_HFARRAY

Hardware floating-point arrays

Description

DOM_HFARRAY is a multidimensional container type, storing hardware floating-point numbers at integer indices.

Unlike generic arrays, objects of type DOM_HFARRAY are containers of hardware floating-point numbers, real or complex. They take up considerably less space than the corresponding arrays of software floats (DOM_FLOAT) would, but the range of hardware floating-point numbers is much more limited.

Function Calls

Using an hf-array as the symbol of a function call returns that hf-array unchanged. The arguments of the call are not evaluated.

Operations

Read and write access to an hf-array is performed using indexed access, as in A[1], which automatically converts between hardware and software floats. Trying to write a value which cannot be converted into a hardware float into an hf-array causes an error to be raised, as does accessing an element out of bounds.

The function map applies some function or transformation to each element of an hf-array, returning an hf-array of the same format as its input, with the results of the calls as its entries. If a result cannot be converted to a hardware float, an error is raised.

If A is an hf-array, nops(A) returns the number of elements in A.

Basic arithmetic works on hf-arrays: Addition and subtraction of hf-arrays of identical format combines the containers element-wise, addition and subtraction of constants is applied to the main diagonal. For two-dimensional hf-arrays, multiplication performs matrix multiplication. Division is possible for completeness, but should be avoided, as it

numerically inverts the dividend first, and this is hardly ever the algorithmically “right” way to handle a numerical problem.

Operands

If A is an hf-array, the 0th operand of A, `op(A, 0)`, will be the sequence starting with the number of dimensions (an integer n) followed by n ranges of integers, which denote the acceptable ranges of indices for each dimension, including both numbers listed in the range.

For $1 \le i \le nops(A)$, the ith operand of A is the ith entry of A, in the lexicographic order of indices.

Output

One-dimensional hf-arrays are displayed as row vectors, two-dimensional hf-arrays as matrices. Higher-dimensional hf-arrays are written in functional form, writing the entries as a flat list, and do not have a typesetting version. This also causes typesetting to be disabled for any surrounding expression in the same output.

Element Creation

The primary way of creating hf-arrays is the function `hfarray`. Other important functions (optionally) returning hardware float arrays include several functions of the `numeric` library and `import::readbitmap`.

See Also

MuPAD Domains
`DOM_ARRAY | DOM_LIST | DOM_TABLE`

MuPAD Functions
`float | hfarray`

DOM_IDENT

Symbolic Identifiers

Description

`DOM_IDENT` is the data type of symbolic identifiers, used for example for indeterminates.

To perform symbolic computations, it is often necessary to represent indeterminates, which may or may not carry assumptions. These indeterminates (which in some contexts may also be bound identifers and which may also be assigned specific values) are called “identifiers” in MuPAD and have the domain type `DOM_IDENT`.

Function Calls

Calling a `DOM_IDENT` as a function creates a `DOM_EXPR`. If the identifier has a value, the evaluation of that `DOM_EXPR` may result in an arbitrary value.

Operations

Identifiers are valid arithmetical expressions, so most MuPAD functions happily accept identifiers.

To get and analyze the name of an identifier, you can use `coerce(identifier, DOM_STRING )` and look at the resulting string. (The call `""."identifier"` returns the same string and is shorter to type.)

Operands

Identifiers are atomic.

Output

Identifiers are displayed with their names, with the following special cases in typesetting:

- Underscores (_) in the middle of identifiers cause subscripting: `x_2` is displayed as x_2.
- Certain constructs of the form `` `&...;` `` in identifiers are replaced by special typeset characters. For example, `` `&alpha;&rarr;` `` is displayed as $\vec{\alpha}$. To generate these identifiers, we suggest using the Symbol library, which would use `Symbol::accentRightArrow(Symbol::alpha)` for the example above.

Element Creation

A sequence of characters, underscores and digits which does not start with a digit is considered an identifier. Examples: `x`, `x0`, `t_0`.

Additionally, an arbitrary string of characters enclosed in 'backticks' `` ` `` is also an identifier. Examples: `` `x+y` ``, `` `a plus 1` ``. If the string of characters between the back ticks is a valid identifier already, this input form creates the same identifier as the one without the backticks.

See Also

MuPAD Domains
`DOM_EXPR` | `DOM_VAR`

MuPAD Functions
`genident` | `indets`

DOM_INT

Integers

Description

DOM_INT is the domain of integer constants such as 42 or - 56412564156717653. The size of integers is limited to $2^{2^{21}} - 1$ in absolute value.

Function Calls

Calling an integer as a function returns that integer unchanged. The arguments are *not* evaluated.

Operations

Integers are arithmetical expressions and thus accepted by almost every MuPAD function.

To represent an integer in a basis different from 10, please use `int2text`.

Operands

Integers are atomic.

Element Creation

Integers are given by an optional sign (an arbitrarily long string of + and - signs) and a sequence of decimal digits. Apart from this direct input method, many MuPAD commands such as `text2int` return integers.

See Also

MuPAD Domains
DOM_FLOAT | DOM_RAT

DOM_INTERVAL

Floating point intervals

Description

Object of type DOM_INTERVAL represents an interval of complex numbers. Either border may be *infinity* or - ∞. The borders are represented by floating point numbers (DOM_FLOAT).

Objects of type DOM_INTERVAL represent numerical enclosures of rectangles in the complex plane or finite unions thereof. Numerical enclosures of real intervals are an important special case.

Because an element of type DOM_INTERVAL contains floating-point numbers of type DOM_FLOAT, its exact value depends on the value of the environment variable DIGITS at the time of creation.

The result of all arithmetical operations on elements of type DOM_INTERVAL is rounded outwards, that is, the resulting (union of) rectangle(s) is *guaranteed* to contain the exact result. If the result interval is purely real, the lower bound of the result is guaranteed to be no larger than the exact value of the exact result, while the upper value of the result is guaranteed to be no smaller than the exact value. The exact values may not be representable as floating-point numbers. In this case, the result of a single operation such as + or * is the smallest representable interval containing the exact result. In other words, operations on DOM_INTERVAL are locally optimal.

Note that the representation of an element of DOM_INTERVAL on the screen is generated with outward rounding, too. This may lead to "apparent overestimation," as you can see in "Example 1" on page 7-31.

For generating matrix or polynomial rings over floating-point intervals, use the façade domain Dom::FloatIV.

Examples

Example 1

An interval of type `DOM_INTERVAL` can only hold floating-point numbers, which are internally stored as binary numbers. For this reason, it cannot hold symbolic expressions as its operands:

```
iv := hull(PI)
```

$$3.141592653 \ldots 3.141592654$$

This intervals certainly does contain π. However, the value printed on the screen does not accurately describe the interval generated, as you can see when you print the same interval with a larger value of `DIGITS`:

```
DIGITS := 15: iv; delete DIGITS:
```

$$3.14159265358979 \ldots 3.1415926535898$$

In the first output, it looked as if the difference between the two borders (the width of the interval) was 10^{-8}, while in the latter output we can see that it is at most 10^{-13}. Actually, the difference is even smaller:

```
op(iv,2) - op(iv,1)
```

$$6.938893904\ 10^{-18}$$

This rounding does not take place for symbolic values which can be represented exactly in both the internal (binary) and the on-screen (decimal) format:

```
iv := hull(1); op(iv,2) - op(iv,1)
```

$$1.0 \ldots 1.0$$

$$0.0$$

However, floating-point values in the input are assumed to be approximations up to the current computing precision:

```
iv := hull(1.0); op(iv,2) - op(iv,1)
```

$$0.9999999999 \ldots 1.000000001$$

$$5.204170428\ 10^{-18}$$

Example 2

If you convert `infinity` or `-infinity` into an interval of type `DOM_INTERVAL`, the resulting interval will contain the corresponding floating-point infinity, which are displayed as `RD_INF` or `RD_NINF`, respectively:

```
hull(infinity), hull(-infinity)
```

$$2.098578716\ 10^{323228496} \ldots \text{RD\_INF},\ \text{RD\_NINF} \ldots -2.098578716\ 10^{323228496}$$

Since the range of floating-point numbers is limited, also conversion of finite values may generate floating-point infinities. The exact limit of floating-point numbers may change from one MuPAD version to the next. Currently, the following command exceeds the representable range::

```
hull(exp(10^9))
```

$$2.098578716\ 10^{323228496} \ldots \text{RD\_INF}$$

As for calculating with intervals with infinities as their borders, note that any multiplication where one factor is exactly zero and the other factor contains either infinity results in the interval encompassing the whole real axis:

```
(0...0) * (1e30...infinity)
```

$$\text{RD\_NINF} \ldots \text{RD\_INF}$$

Example 3

Objects of type DOM_INTERVAL represent rectangular subsets of the complex plane with the sides parallel to the coordinate axes or unions of such rectangles. In the following example, the exact result would be a rotated rectangle. Interval arithmetics returns the smallest non-rotated rectangle enclosing the precise result:

```
(1...2+I)*(1+I)
```

$$0.0 \ldots 2.0 + 1.0 \ldots 3.0\,i$$

Example 4

When computing with floating-point numbers, you can easily run into cancellation and round-off errors without noticing:

```
(1+1e-18) - 1.0
```

$$0.0$$

```
sin(1e42)
```

$$0.4587729978$$

The fundamental problem of numerical computations is that there is no indication to these errors. Of course, you can get around most problems of this type by increasing the number of significant digits, if you know that problems are lurking:

```
DIGITS:=50:
(1+1e-18) - 1.0,
sin(1e42)
```

$$9.9999999999999999999999999999999999994337442995\ 10^{-19},$$

$$-0.79299795477606144587030822830754387995851407627768$$

Interval calculations, amongst other things, provide a way to be notified of cancellation since the result of an operation over DOM_INTERVAL is certain to contain the exact result:

```
DIGITS:=10:
(1+1e-18) - hull(1.0)
```

$$-5.204170428\ 10^{-18} \ldots 5.204170428\ 10^{-18}$$

```
sin(hull(1e42))
```

$$-1.0 \ldots 1.0$$

```
DIGITS:=50:
sin(hull(1e42))
```

$$-0.79299795477606165563526882849270558686988301102149 \ldots -0.79299795477606111453539571969660597818990980366 84$$

So, in the latter case we know that the first 17 digits are correct and that the 18th digit is 3, 4, or 5.

Function Calls

The result of a call to an interval is the interval itself, regardless of the arguments. The arguments are not evaluated.

Operations

You can access the borders of an interval using `op`. See below for the details.

Intervals can be viewed as sets, and the corresponding functions `union`, `intersect`, and `minus` work on intervals, too.

As of version 2.5, MuPAD implements the following operations on elements of type `DOM_INTERVAL`:

- The basic arithmetical operations: `+`, `-`, `*`, `/`, `^`, `sqrt`.
- The trigonometric functions and their inverses: `sin`, `cos`, `tan`, `sec`, `csc`, `cot`, `arcsin`, `arccos`, `arctan`, `arccsc`, `arccot`.

- The exponential function and the logarithm.
- The hyperbolic functions and their inverses: `sinh`, `cosh`, `tanh`, `sech`, `csch`, `coth`, `arcsinh`, `arccosh`, `arctanh`, `arccsch`, `arccoth`.
- The functions `Re`, `Im` (real- and imaginary part), `abs`, `sign` and `arg` (the `argument' = polar angle of a complex number).
- For real intervals, `gamma` and `beta`.
- `ceil`, `floor`, `trunc`, `round`.

For legal combinations of arguments, all computations are carried out in interval arithmetics, see “Example 4” on page 7-33.

Operands

The operands of an interval depend on its value:

- An interval of type `DOM_INTERVAL` may be a union of rectangles in the complex plane. In this case, the 0th operand is the identifier `union`, while the remaining operands are the corresponding rectangles, which are of type `DOM_INTERVAL`.
- Rectangles with non-zero imaginary part, which are not unions, have two operands of type `DOM_INTERVAL`: Their real and imaginary parts, both of which are real intervals.
- Real intervals, i.e., non-union rectangles with vanishing imaginary part, have two operands, their left and right borders.

Output

A real interval is displayed in the form “left `...` right”, where “left” and “right” are the borders of the interval, printed as floating-point numbers.

A complex interval is displayed as “(real part) + (imaginary part) * I”, with the real and imaginary part displayed as real intervals.

A union of rectangles is displayed as “interval union interval”, with the intervals inside written as specified above.

The output of an interval depends on the environment variable `DIGITS` as well as on the preference settings `Pref::floatFormat` and `Pref::trailingZeroes`.

Note that the borders are rounded outwards for printing. “Example 1” on page 7-31 shows how this effects the output.

Element Creation

Elements of type DOM_INTERVAL can be constructed in the following ways:

- With the function hull:

```
hull(PI, -3, 1/2), hull(1/3)
```

$$-3.0 \ldots 3.141592654,\ 0.3333333333 \ldots 0.3333333334$$

- With the operator ... (which in turn calls hull):

```
1 ... 4+I
```

$$1.0 \ldots 4.0 + 0.0 \ldots 1.0\,i$$

- The function interval creates elements of type DOM_INTERVAL as well, but may return expressions:

```
interval(x^2+sin(1))
```

$$x^2 + 0.8414709848 \ldots 0.8414709849$$

Note that floating-point values in the input of hull or interval are considered to be approximations, even if the value displayed in the decimal system can be represented exactly in the internal binary format. This is because hull cannot decide whether, for example, 0.25 has actually been typed in as such or if it should have been some 0.25 + ϵ. If you want zero-width intervals, use a rational number as input which can be represented exactly in binary:

Algorithms

Intervals of type DOM_INTERVAL are always interpreted as *closed* intervals, i.e., the endpoints belong to the set. It is reasonable not to have open intervals included, since most operations will enlarge the resulting interval anyway (although only marginally so).

DOM_LIST

Lists of Objects

Description

Lists (of domain type `DOM_LIST`) are ordered collections of an arbitrary number of arbitrary MuPAD objects, except for sequences and the null object.

In MuPAD, the mathematical construct of an n-tuple is implemented as the data type `DOM_LIST`. Lists consist of an arbitrary (finite) number of arbitrary objects, with the exception of expression sequences, which are split into their operands when placed into a list.

Unlike sets, lists can contain multiple copies of the same element. The order of elements in a list is preserved.

Lists can be empty.

Examples

Example 1

To create a list for our first example, we use the operator `$`:

```
L := [x_.i $ i=1..10]
```

$$[x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_10]$$

This list contains 10 elements:

```
nops(L)
```

$$10$$

The fifth element of the list is x_5 and the list of elements from x_3 through x_6 can also be accessed very easily:

```
L[5], L[3..6]
```

$$x_5,\ [x_3, x_4, x_5, x_6]$$

To change an element of the list, we use the indexed form on the left hand side of an assignment:

```
L[5] := 5
```

$$5$$

```
L
```

$$[x_1, x_2, x_3, x_4, 5, x_6, x_7, x_8, x_9, x_10]$$

Note that this assignment only changes `L`, not x_5:

```
x_5
```

$$x_5$$

Likewise, we can change a sublist by assigning another list to it. This may change the length of the list:

```
L[3..6] := [1, 2]
```

$$[1, 2]$$

```
nops(L), L
```

$$8,\ [x_1, x_2, 1, 2, x_7, x_8, x_9, x_10]$$

Function Calls

Using a list as a function symbol creates the list obtained by using each list element as a function symbol for the operands used, i.e., `[f, g](x, y)` results in `[f(x, y), g(x, y)]`.

Operations

Assuming that `L` is a list, the number of elements in `L` can be determined by calling `nops(L)`.

Individual elements of the list are accessed in the form `L[1]`, `L[2]` etc. when counting from the beginning or `L[-1]`, `L[-2]` etc. when counting from the end. Trying to access an element "outside" the list or `L[0]` raises an error.

Continuous sub-lists can be extracted by using a range in an indexed access: `L[2..4]` returns the list `[L[2], L[3], L[4]]`; `L[2..-2]` returns the list `L` without its first and last element.

Both forms of indexed access can also be used as the left hand side of an assignment, cf. "Example 1" on page 7-37.

Lists can be concatenated with the dot operator, as in `L1 . L2` or its functional form, `_concat`.

The function `contains` finds the first occurrence of a given MuPAD object in a list. `select` and `split` can be used to extract those elements from a list fulfilling an arbitrary predicate.

Lists can be sorted with `sort` or `prog::sort`.

The function `map` applies a function to all elements of a list, returning the list of results. To combine two lists element-wise with some function, use `zip`.

Assigning a list to a list of identifiers is possible and results in a simultaneous assignment, cf. `?_assign`.

Operands

The operands of a list are its elements.

Element Creation

The most direct way of creating a list is to place a sequence of MuPAD objects (separated by commas) between rectangular brackets, as in `[1, 2, 3]`.

See Also

MuPAD Domains
DOM_ARRAY | DOM_HFARRAY | DOM_SET

DOM_PROC

Data type of procedures

Description

DOM_PROC is the data type of procedures.

MuPAD procedures belong to the kernel domain DOM_PROC.

You can enclose procedures of type DOM_PROC into function environments of type DOM_FUNC_ENV.

Element Creation

proc and its equivalent -> create procedures of type DOM_PROC.

See Also

MuPAD Domains
DOM_PROC_ENV

MuPAD Functions
funcenv | proc

DOM_PROC_ENV

Data type of procedure environments

Description

Procedure environments are mostly ephemeral objects and are only rarely seen by the user (and even more rarely useful to examine). A procedure environment represents a procedure that is currently being executed: formal parameters and local variables have values.

Procedure environments do rarely become visible, and you do not need to manipulate them directly. They serve only one purpose: if a procedure is generated inside another procedure, variable names in the body of the inner procedure that are not declared local there refer to names in the outer procedure, provided they are declared local in the outer procedure. (See the Programming Manual for more information on the scoping rules for MuPAD.) Consequently, the inner procedure must contain information on the current values of local variables of the outer procedure. Hence, the status of the outer procedure is encoded into an object of type `DOM_PROC_ENV`, and that object is stored in the returned procedure as its twelfth operand.

You never need to generate objects of this type. There are no operations available.

Examples

Example 1

The only occasion on which you should come across a procedure environment is the following: an outer procedure returns an inner procedure depending on formal parameters or local variables of the outer procedure:

```
outer :=
proc(x)
option escape;
begin
  /* inner procedure to return : */
  y -> x + y
end_proc:
```

```
add5 := outer(5)
```

$$y \to x + y$$

In spite of the (slightly confusing) output, `x` has a special meaning here: it points to the parameter `x` of `outer`. That parameter currently has the value `5` and won't be changed any more. To be able to access that value, the particular instance of `outer` in the status of being executed has to be stored in `add5`:

```
expr2text(op(add5, 12))
```

```
"DOM_PROC_ENV(0x1f9d904)"
```

Operands

The number of operands of a procedure environment depends on the number of local and saved variables of the outer procedure. Details about the operands remain undocumented.

Algorithms

The integers appearing in the output of objects of type `DOM_PROC_ENV` have no mathematical meaning; they denote positions in memory.

DOM_RAT

Rational Numbers

Description

DOM_RAT is the data type of rational numbers.

Examples

Example 1

The operands of a rational number are its numerator and denominator:

```
op(2/3)
```

$$2, 3$$

When substituting an operand, the resulting DOM_RAT is again normalized:

```
subsop(2/3, 2=6)
```

$$\frac{1}{3}$$

Function Calls

Using a rational number as a function returns that number unchanged. The function arguments are *not* evaluated.

Operations

Rational numbers are arithmetical expressions and therefore valid inputs to most MuPAD functions.

The numerator and denominator of a rational number can be accessed using `numer` and `denom` or by using `op` directly.

Elements of `DOM_RAT` are always normalized, cf. “Example 1” on page 7-44.

Operands

A rational number has two operands, which are integers: Its numerator and its denominator.

Element Creation

The division of two integers results in an integer or a rational number.

See Also

MuPAD Domains

`Dom::Rational` | `DOM_COMPLEX` | `DOM_FLOAT` | `DOM_INT`

DOM_SET

Sets of Objects

Description

Set of type `DOM_SET` can store an arbitrary finite number of arbitrary MuPAD objects, except for sequences and the null object.

In MuPAD, finite sets are implemented with the data type `DOM_SET`. Sets are unordered collections of arbitrary objects, with identical objects appearing only once. Sequences (objects separated by commas) are "flattened" when put into a set, i.e., instead of the sequence, its elements are placed into the set. The null object is treated as the empty sequence, i.e., it does not result in an element in the set.

Sets can be empty. The empty set is displayed as ∅.

Function Calls

Using a set as a function symbol creates the set obtained by using each element as a function symbol for the operands used, i.e., `{f, g}(x, y)` results in `{f(x, y), g(x, y)}`.

Operations

Assuming that S is a set, the number of elements in S can be obtained by calling `nops(S)`.

Individual elements of the set can be obtained in two subtly different ways:

1 Using an indexed access, as in `S[2]`, returns the n-th element of the set, counted in the order as the set appears on the screen. This is a potentially slow operation, since it requires determining that order for each access, i.e., sorting the set.

  Negative indices are accepted, counting from the end of the sequence of elements. Trying to access an element "outside" the set or `S[0]` raises an error.

2 Using `op`, as in `op(S, 2)`, returns the n-th element of the set, counted in the internal order. This is a fast operation ($O(n)$ to get the n-th element, irrespective of the size of S), but the internal order of two mathematically identical sets can be completely different and almost any operation changing a set can completely change its internal order, so no assumptions should be made.

Both of these ways also accept ranges as indices. `S[2..4]` returns the set `{S[2], S[3], S[4]}`, while `op(S, 2..4)` returns the sequence `op(S, 2), op(S, 3), op(S, 4)`.

To iterate over all elements of a set in no particular order, using `map` or the `$` operator is highly superior to using a `for`-loop with either of the above element access methods. If a `for`-loop is required, you should use the form `for s in S`, which has linear complexity as well.

The usual set operations are provided as infix operations: `union`, `minus`, `intersect`.

To change an element of a set, the preferred method is to remove it using `minus` and adding a new one using `union`. It is also possible to replace an element with `subsop`; replacing an element with `null()` deletes it from the set. (Note that `subsop` does not do a side-effect assignment.)

The function `contains` checks for occurrence of a given MuPAD object in a set; see also the `in` operator for the same purpose, but with different evaluation semantics. `select` and `split` can be used to extract those elements from a set fulfilling an arbitrary predicate.

To get a list of the elements of a set, use `coerce`. To get such a list with the elements ordered in the same way as printed on the screen, use `DOM_SET::sort(S)`.

Operands

The operands of a set are its elements, in the internal order. (See above for details.)

Output

Sets are ordered for the output.

Element Creation

The most direct way of creating a set is to place a sequence of MuPAD objects (separated by commas) between curly brackets, as in {1, 2, 3}.

See Also

MuPAD Domains
`Dom::ImageSet` | `Dom::Multiset` | `DOM_LIST`

DOM_STRING

Texts (character strings)

Description

Texts (which are not really “mathematical objects”, but useful to the programmer) in MuPAD are of domain type `DOM_STRING`.

MuPAD can manipulate texts (strings of characters). These are primarily used for output and data input.

Examples

Example 1

As far as `op` is concerned, a string cannot be dissected:

```
s := "this is a string":
op(s, 1), op(s, 2)
```

```
"this is a string", FAIL
```

To access individual characters or substrings, use indexed access:

```
s[1], s[6..7]
```

```
"t", "is"
```

Assigning to a substring may change the length of a string:

```
s[6..7] := "changes";
s
```

```
"changes"
```

```
"this changes a string"
```

Function Calls

Using a string as a function returns the string unchanged. The arguments are *not* evaluated.

Operations

Strings can be concatenated using the dot operator or its functional equivalent, `_concat`.

The length of a string can be obtained using `length`.

Substrings and individual characters (which are strings of length 1) can be accessed using `substring` or indexed access, with indices starting at 1 and negative indices counting from the end of the string: `s[1]`, `s[3..-2]`. It is also possible to perform an indexed assignment to a string, cf. “Example 1” on page 7-49.

To convert a string into the MuPAD expression that would be obtained by using the string as an input, use `text2expr`. For simple MuPAD expressions, it is possible to get a string that evaluates to that expression using `expr2text`. Expressions that are not convertible in this way include all expressions containing local variables set with `option escape`. Also, expressions involving floating point numbers usually will change when being converted to strings and back.

Operands

Strings are atomic, i.e., they have exactly one operand, the string itself.

Output

The output form of strings is very similar to their input form. When typesetting, spaces at the beginning and the end of strings are ignored and multiple adjacent blanks as well as newlines are collapsed to a single space.

Element Creation

A string is created by enclosing characters in a pair of typewriter quotes: `"this is a string"`. The following special sequences are supported (but see below for the typeset output; these are useful only for non-typeset output):

- `"\n"` denotes an end-of-line character.
- `"\b"` is almost identical to `"\n"`, except that for "pretty-printing" it encodes the baseline of the current object.
- `"\t"` is a tabulator.
- `"\\"` encodes a backslash.

See the documentation of `print` for details.

DOM_VAR

Local Variables in Procedures

Description

Local variables (variables in the programming sense, with “lexical scoping”) are of domain type `DOM_VAR`.

When writing MuPAD functions, often intermediate results need to be stored and retrieved. Like most programming languages, MuPAD offers “local variables” for this purpose. These local variables do not conflict with global identifiers of the same name nor with other local variables of the same name used at other places.

Local variables use “lexical scoping”, i.e., they can be used in all program code that is written inside the body of the procedure declaring the local variable. Note that returning anything with a reference to a local variable requires the use of `option escape` in the procedure definition.

Operations

Local variables can be assigned values and these values can later be retrieved.

Element Creation

Local variables are created by using either the special names `dom` or `procname` or one of the names declared with the keyword `local` inside a procedure definition.

See Also

MuPAD Domains
`DOM_IDENT | DOM_PROC | DOM_PROC_ENV`

MuPAD Functions
`context | proc`

Dom::AlgebraicExtension

Simple algebraic field extensions

Syntax

Domain Creation

```
Dom::AlgebraicExtension(F, f)
Dom::AlgebraicExtension(F, f, x)
Dom::AlgebraicExtension(F, f1 = f2)
Dom::AlgebraicExtension(F, f1 = f2, x)
```

Element Creation

```
Dom::AlgebraicExtension(F,f)(g)
Dom::AlgebraicExtension(F, f)(rat)
```

Description

Domain Creation

For a given field *F* and a polynomial $f \in F[x]$, `Dom::AlgebraicExtension(F, f, x)` creates the residue class field *F[x]/<f>*.

`Dom::AlgebraicExtension(F, f1=f2, x)` does the same for $f = f_1 - f_2$.

`Dom::AlgebraicExtension(F, f, x)` creates the field *F[x]/<f>* of residue classes of polynomials modulo *f*. This field can also be written as *F[x]/<f>*, the field of residue classes of rational functions modulo *f*.

The parameter `x` may be omitted if `f` is a univariate polynomial or a polynomial expression that contains exactly one indeterminate; it is then taken to be the indeterminate occurring in `f`.

The field `F` must have normal representation.

`f` must not be a constant polynomial.

`f` must be irreducible; this is *not* checked.

`f` may be a polynomial over a coefficient ring different from `F`, or multivariate; however, it must be possible to convert it to a univariate polynomial over `F`. See "Example 2" on page 7-55.

Element Creation

`Dom::AlgebraicExtension(F, f)(g)` creates the residue class of `g` modulo `f`.

If `rat` has numerator and denominator `p` and `q`, respectively, then `Dom::AlgebraicExtension(F,f)(rat)` equals `Dom::AlgebraicExtension(F,f)(p)` divided by `Dom::AlgebraicExtension(F,f)(q)`.

Superdomain

`Dom::BaseDomain`

Axioms

If F has `Ax::canonicalRep`, then `Ax::canonicalRep`.

Categories

`Cat::Field`, `Cat::Algebra(F)`, `Cat::VectorSpace(F)`

If F is a `Cat::DifferentialRing`, then `Cat::DifferentialRing`.

If F is a `Cat::PartialDifferentialRing`, then `Cat::PartialDifferentialRing`.

Examples

Example 1

We adjoin a cubic root `alpha` of `2` to the rationals.

```
G := Dom::AlgebraicExtension(Dom::Rational, alpha^3 = 2)
```

$$\text{Dom::AlgebraicExtension}(\text{Dom::Rational}, \text{alpha}^3 - 2 = 0, \text{alpha})$$

The third power of a cubic root of `2` equals `2`, of course.

```
G(alpha)^3
```

$$2$$

The trace of α is zero:

```
G::conjTrace(G(alpha))
```

$$0$$

You can also create random elements:

```
G::random()
```

$$2 - 814\ \text{alpha}^2 - 65\ \text{alpha} + 824$$

Example 2

The ground field may be an algebraic extension itself. In this way, it is possible to construct a tower of fields. In the following example, an algebraic extension is defined using a primitive element `alpha`, and the primitive element `beta` of a further extension is defined in terms of `alpha`. In such cases, when a minimal equation contains more than one identifier, a third argument to `Dom::AlgebraicExtension` must be explicitly given.

```
F := Dom::AlgebraicExtension(Dom::Rational, alpha^2 = 2):
G := Dom::AlgebraicExtension(F, bet^2 + bet = alpha, bet)
```

$$\text{Dom::AlgebraicExtension}(\text{Dom::AlgebraicExtension}(\text{Dom::Rational}, \text{alpha}^2 - 2 = 0, \text{alpha}), -\text{alpha} + \text{bet} + \text{bet}^2 = 0, \text{bet})$$

Example 3

We want to define an extension of the field of fractions of the ring of bivariate polynomials over the rationals.

```
P:= Dom::DistributedPolynomial([x, y], Dom::Rational):
F:= Dom::Fraction(P):
K:= Dom::AlgebraicExtension(F, alpha^2 = x, alpha)
```

```
Dom::AlgebraicExtension(Dom::Fraction(Dom::DistributedPolynomial([x,
  y], Dom::Rational, LexOrder)), -x + alpha^2 = 0, alpha)
```

Now $K = Q[\sqrt{x}, y]$. Of course, the square root function has the usual derivative; note that $\frac{1}{\sqrt{x}}$ can be expressed as $\frac{\alpha}{x}$:

```
diff(K(alpha), x)
```

$$\frac{\text{alpha}}{2\,x}$$

On the other hand, the derivative of $\sqrt{x}$ with respect to y is zero, of course:

```
diff(K(alpha), y)
```

$$0$$

We must not use `D` here. This works only if we start our construction with a ring of univariate polynomials:

```
P:= Dom::DistributedPolynomial([x], Dom::Rational):
F:= Dom::Fraction(P):
K:= Dom::AlgebraicExtension(F, alpha^2 = x, alpha):
D(K(alpha))
```

$$\frac{\text{alpha}}{2\,x}$$

Parameters

`F`

The ground field: a domain of category `Cat::Field`

`f, f1, f2`

Polynomials or polynomial expressions

`x`

Identifier

`g`

Element of the residue class to be defined: polynomial over `F` in the variable `x`, or any object convertible to such.

`rat`

Rational function that belongs to the residue class to be defined: expression whose numerator and denominator can be converted to polynomials over `F` in the variable `x`. The denominator must not be a multiple of `f`.

Entries

"zero"	the zero element of the field extension
"one"	the unit element of the field extension
"groundField"	the ground field of the extension
"minpoly"	the minimal polynomial `f`
"deg"	the degree of the extension, i.e., of `f`
"variable"	the unknown of the minimal polynomial `f`
"characteristic"	the characteristic, which always equals the characteristic of the ground field. This entry only exists if the characteristic of the ground field is known.

"degreeOverPrimeField"	the dimension of the field when viewed as a vector space over the prime field. This entry only exists if the ground field is a prime field, or its degree over its prime field is known.

Methods

Mathematical Methods

`_plus` — Sum of field elements

```
_plus(a, …)
```

This method overloads the function `_plus` of the system kernel.

`_mult` — Product of field elements

```
_mult(a, …)
```

This method overloads the function `_mult` of the system kernel.

`_power` — Raise to the nth power

Inherited from `Cat::Monoid`.

`_negate` — Negate a field element

```
_negate(a)
```

This method overloads the function `_negate` of the system kernel.

`_subtract` — Difference of field elements

```
_subtract(a, b)
```

This method overloads the function `_subtract` of the system kernel.

`equal` — Test for mathematical equality

Inherited from `Dom::BaseDomain`.

`equiv` — Test for equivalence

Inherited from `Cat::BaseCategory`.

`iszero` — Test whether a field element is zero.

```
iszero(a)
```

This method overloads the function `iszero`.

`isone` — Test if element is one

Inherited from `Cat::Monoid`.

`isUnit` — Test if element is an unit

Inherited from `Cat::Field`.

`intmult` — Multiply a field element by an integer

```
intmult(a, b)
```

This method is more efficient than `"_mult"` in this special case.

`_invert` — Inverse of a field element

```
_invert(a)
```

This method overloads the function `_invert`.

`_divide` — Exact division

Inherited from `Cat::Field`.

`divide` — Division with remainder

Inherited from `Cat::Field`.

`quo` — Return Euclidean quotient

Inherited from `Cat::Field`.

`rem` — Return Euclidean remainder

Inherited from `Cat::Field`.

`euclideanDegree` — Return Euclidean degree

Inherited from `Cat::Field`.

`idealGenerator` — Generator of finitely generated ideal

Inherited from `Cat::EuclideanDomain`.

`divides` — Test if division is exact

Inherited from `Cat::Field`.

`gcd` — Gcd of field elements

`gcd(a, …)`

This method overloads the function `gcd`.

`gcdex` — Extended greatest common divisor

Inherited from `Cat::EuclideanDomain`.

`associates` — Test for associate elements

Inherited from `Cat::Field`.

`unitNormal` — Unit normal form

Inherited from `Cat::Field`.

`unitNormalRep` — Unit normal representation

Inherited from `Cat::Field`.

`lcm` — Least common multiple

Inherited from `Cat::GcdDomain`.

`sqrfree` — Square-free factorization

Inherited from `Cat::Field`.

`irreducible` — Test if element is irreducible

Inherited from `Cat::Field`.

`factor` — Unique factorization

Inherited from `Cat::Field`.

`conjNorm` — Norm of an element

```
conjNorm(a)
```

`conjTrace` — Trace of an element

```
conjTrace(a)
```

`minimalPolynomial` — Minimal polynomial of an element

```
minimalPolynomial(a)
```

`D` — Differential operator

```
D(a)
```

This method overloads the function `D`.

This method must not be called for inseparable extensions; note that MuPAD cannot check whether an extension is separable.

See “Example 3” on page 7-56.

`diff` — Partial differentiation

```
diff(a, x1, …)
```

Differentiation is defined to be the continuation of differentiation of the ground field; this method exists only if the ground field has a method `"diff"`, too.

Differentiation is not possible in inseparable extensions.

This method overloads the function `diff`.

This method must not be called for inseparable extensions; note that MuPAD cannot check whether an extension is separable.

See “Example 3” on page 7-56.

`random` — Random element of the field

```
random()
```

The `random` method of the ground field is used to generate coefficients of a random polynomial of the ground field; the residue class of that polynomial is the return value. Hence the probability distribution of the elements returned depends on that of the `random` method of the ground field.

Conversion Methods

`convert` — Convert into a field element

```
convert(x)
```

If the conversion fails, then `FAIL` is returned.

`convert_to` — Convert a field element into another type

```
convert_to(a, T)
```

Field elements can be converted to polynomials or expressions. Field elements represented by constant polynomials can also be converted to the same types as the elements of the ground field; in particular, they can be converted to elements of the ground field.

`coerce` — Coerce into this domain

Inherited from `Cat::BaseCategory`.

`new` — Create element of this domain

Inherited from `Cat::BaseCategory`.

`expr` — Convert an element of the field into an expression

```
expr(a)
```

This method overloads the function `expr`.

`subs` — Avoid substitution

Inherited from `Dom::BaseDomain`.

`subsex` — Avoid extended substitution

Inherited from `Dom::BaseDomain`.

`testtype` — Test type of object

Inherited from `Cat::BaseCategory`.

`print` — Return expression to print an element

Inherited from `Cat::BaseCategory`.

`printMethods` — Print out methods

Inherited from `Dom::BaseDomain`.

`TeX` — Generate TeX output

Inherited from `Dom::BaseDomain`.

`hasProp` — Test for a certain property

Inherited from `Dom::BaseDomain`.

`whichEntry` — Return the domain or category implementing an entry

Inherited from `Dom::BaseDomain`.

`allEntries` — Return the names of all entries

Inherited from `Dom::BaseDomain`.

`undefinedEntries` — Return missing entries

Inherited from `Dom::BaseDomain`.

`getAxioms` — Return axioms stated in the constructor

Inherited from `Dom::BaseDomain`.

`getSuperDomain` — Return super-domain stated in the constructor

Inherited from `Dom::BaseDomain`.

`allSuperDomains` — Return all super-domains

Inherited from `Dom::BaseDomain`.

`getCategories` — Return categories stated in the constructor

Inherited from `Dom::BaseDomain`.

`allAxioms` — Return all axioms

Inherited from `Dom::BaseDomain`.

`allCategories` — Return all categories

Inherited from `Dom::BaseDomain`.

See Also

MuPAD Domains
`Dom::GaloisField`

Dom::ArithmeticalExpression

Domains of arithmetical expressions

Syntax

```
Dom::ArithmeticalExpression(x)
```

Description

`Dom::ArithmeticalExpression` is a façade domain of arithmetical expressions built up by the system functions and operators like `+` and `*`.

This domain has almost no algebraic structure because unqualified expressions have no normal form. (For example, there are rational expressions for zero which are not normalized to `0`.) The main purpose of `Dom::ArithmeticalExpression` is to provide implementations for methods used by façade sub-domains like `Dom::Integer` which are represented by a subset of the arithmetical expressions.

Elements of `Dom::ArithmeticalExpression` are usually not created explicitly. However, if one creates elements using the usual syntax, the input is converted to an expression using `expr`, then it is checked whether the result is an arithmetical expression.

Superdomain

`Dom::Expression`

Axioms

`Ax::systemRep`

Categories

`Cat::BaseCategory`

Examples

Example 1

For brevity, we will use `AE` as a shorthand notation for `Dom::ArithmeticalExpression`:

```
AE := Dom::ArithmeticalExpression
```

Dom::ArithmeticalExpression

An element of this domain can *not* be created as follows:

```
e := AE(2*sin(x) + f(x)/y)
```

$$2\sin(x) + \frac{f(x)}{y}$$

Since `Dom::ArithmeticalExpression` is a façade domain, `e` is not a domain element, but an expression:

```
domtype(e)
```

DOM_EXPR

The fact that no error was returned yields the information that `e` is an arithmetical expression. This can also be checked as follows:

```
testtype(e,AE)
```

TRUE

In contrast to its super-domain `Dom::Expression`, this domain only allows elements which are valid arguments for the arithmetical functions, thus the following yields an error:

```
AE([a, b])
```

```
Error: The arguments are invalid. [Dom::ArithmeticalExpression::new]
```

Parameters

x

An arithmetical expression

Entries

"key"	The name of this domain.
"one"	The neutral element w.r.t. `"_mult"`: the constant `1`.
"zero"	The neutral element w.r.t. `"_plus"`: the constant `0`.

Methods

Mathematical Methods

`_divide` — Divide arithmetical expressions

```
_divide(f, g)
```

This method overloads the function `_divide`.

For details, please see `_divide`.

`_invert` — Invert an arithmetical expression

```
_invert(f)
```

This method overloads the function `_invert`.

For details, please see `_invert`.

`_mult` — Multiplie arithmetical expressions

```
_mult(<f, g, …>)
```

This method overloads the function `_mult`.

For details, please see `_mult`.

`_negate` — Negate an arithmetical expression

```
_negate(f)
```

This method overloads the function `_negate`.

For details, please see `_negate`.

`_plus` — Add arithmetical expressions

```
_plus(<f, g, …>)
```

This method overloads the function `_plus`.

For details, please see `_plus`.

`_power` — Power operator

```
_power(f, g)
```

This method overloads the function `_power`.

For details, please see `_power`.

`_subtract` — Subtract an arithmetical expression

```
_subtract(f, g)
```

For details, please see `_subtract`.

`D` — Differential operator for functions

```
D(f)
```

`D([c1, …], f)`

This method overloads the function `D`.

For details, please see `D`.

`diff` — Differentiate an arithmetical expression

`diff(f, <x, …>)`

This method overloads the function `diff`.

For details, please see `diff`.

`equiv` — Test for equivalence

Inherited from `Cat::BaseCategory`.

`intmult` — Multiply an arithmetical expression with an integer

`intmult(f, n)`

This method overloads the function `_mult`.

For details, please see `_mult`.

`iszero` — Test for zero

`iszero(f)`

This method overloads the function `iszero`.

For details, please see `iszero`.

`max` — Maximum of numbers

`max(x, <y, …>)`

All numerical arguments must be real.

This method overloads the function `max`.

For details, please see `max`.

`min` — Minimum of numbers

```
min(x, <y, …>)
```

All numerical arguments must be real.

This method overloads the function `min`.

For details, please see `min`.

`norm` — Norm of an arithmetical expression

```
norm(f)
```

This method overloads the function `abs`.

For details, please see `abs`.

`random` — Create random expression

Inherited from `Dom::Expression`.

Access Methods

`subs` — Substitution

Inherited from `Dom::Expression`.

`subsex` — Extended substitution

Inherited from `Dom::Expression`.

Conversion Methods

`coerce` — Coerce into this domain

Inherited from `Cat::BaseCategory`.

`convert` — Check for being an arithmetical expression

```
convert(x)
```

`convert_to` — Conversion to other domains

Inherited from `Dom::Expression`.

`expr` — Just return the argument

Inherited from `Dom::Expression`.

`float` — Convert numbers to floats

Inherited from `Dom::Expression`.

Technical Methods

`allAxioms` — Return all axioms

Inherited from `Dom::BaseDomain`.

`allCategories` — Return all categories

Inherited from `Dom::BaseDomain`.

`allEntries` — Return the names of all entries

Inherited from `Dom::BaseDomain`.

`allSuperDomains` — Return all super-domains

Inherited from `Dom::BaseDomain`.

`testtype` — Test type of object

Inherited from `Cat::BaseCategory`.

`undefinedEntries` — Return missing entries

Inherited from `Dom::BaseDomain`.

`whichEntry` — Return the domain or category implementing an entry

Inherited from `Dom::BaseDomain`.

`getAxioms` — Return axioms stated in the constructor

Inherited from `Dom::BaseDomain`.

`getCategories` — Return categories stated in the constructor

Inherited from `Dom::BaseDomain`.

`getSuperDomain` — Return super-domain stated in the constructor

Inherited from `Dom::BaseDomain`.

`hasProp` — Test for a certain property

Inherited from `Dom::BaseDomain`.

`info` — Print short information about this domain

Inherited from `Dom::BaseDomain`.

`new` — Create element of this domain

Inherited from `Cat::BaseCategory`.

`print` — Return expression to print an element

Inherited from `Cat::BaseCategory`.

`printMethods` — Print out methods

Inherited from `Dom::BaseDomain`.

`testtype` — Test whether its argument is an expression

Inherited from `Dom::Expression`.

`undefinedEntries` — Return missing entries

Inherited from `Dom::BaseDomain`.

`whichEntry` — Return the domain or category implementing an entry

Inherited from `Dom::BaseDomain`.

See Also

MuPAD Domains
`Dom::Expression`

Dom::BaseDomain

Root of the domain hierarchy

Description

`Dom::BaseDomain` is the root of the domain hierarchy as defined by the Dom package. Every domain of the package inherits from it.

The only purpose of `Dom::BaseDomain` is to supply all domains of the package with some basic methods like `"hasProp"`. Elements of `Dom::BaseDomain` cannot be created.

Unlike other super-domains this domain does not impose any restrictions on the representation of the elements of its sub-domains. Thus it may be a super-domain for any domain created by a domain constructor.

Categories

`Cat::BaseCategory`

Entries

"create_dom"	This domain entry is used to revive the domain when it is read from a binary MCode stream. If this entry is present it is written to the MCode stream instead of the contents of the domain. When the stream is read it is used to create the domain. If this entry does not exist all entries of the domain are written to the stream and read in later to create the domain. `Dom::BaseDomain` defines `"create_dom"` to have the same value as the key of the

domain, as stored in the entry `"key"`. All domains of the Dom package inherit this entry, thus they must be created by the reader of the MCode stream by evaluating the expression stored in the key.

Methods

Mathematical Methods

`equal` — Test for mathematical equality

```
equal(x, y)
```

If this domain has the axiom `Ax::canonicalRep`, which implies that two domain elements are mathematically equal if and only if they are structurally equal, the kernel function `_equal` is used to decide the equality. In this case `UNKNOWN` is never returned.

If the axiom `Ax::canonicalRep` does not hold the method will return `TRUE` if `x` and `y` are structurally equal (in the sense of the function `_equal`) and `UNKNOWN` otherwise.

Conversion Methods

`convert_to` — Convert element

```
convert_to(x, T)
```

The implementation provided here can convert `x` to an element of this domain (the trivial case) or to an element of `Dom::Expression` (by using the method `"expr"`, see `Cat::BaseCategory`).

`TeX` — Generate TeX output

```
TeX(x)
```

The default implementation provided here converts `x` into an expression using the method `"expr"` and then uses the function `generate::TeX` to convert the expression.

Access Methods

`allAxioms` — Return all axioms

```
allAxioms()
```

`allCategories` — Return all categories

```
allCategories()
```

`allEntries` — Return the names of all entries

```
allEntries()
```

`allSuperDomains` — Return all super-domains

```
allSuperDomains()
```

The last, most general, super-domain of all domains of the Dom package is `Dom::BaseDomain`.

`getAxioms` — Return axioms stated in the constructor

```
getAxioms()
```

`getCategories` — Return categories stated in the constructor

```
getCategories()
```

`getSuperDomain` — Return super-domain stated in the constructor

```
getSuperDomain()
```

`hasProp` — Test for a certain property

```
hasProp(d)
```

```
hasProp(dc)
```

```
hasProp(a)
```

```
hasProp(ac)
```

`hasProp(c)`

`hasProp(cc)`

`hasProp(dc)` tests if this domain or a super-domain of it was defined by the domain constructor `dc`.

`hasProp(a)` tests if this domain has the axiom `a`.

`hasProp(ac)` tests if an axiom of this domain was defined by the axiom constructor `ac`.

`hasProp(c)` tests if this domain has the category `c`.

`hasProp(cc)` tests if a category of this domain was defined by the category constructor `cc`.

`info` — Print short information about this domain

`info()`

It prints out the super-domains, categories, axioms and entry names of this domain.

If an entry `"info_str"`, which must be a string, is defined for this domain it is used to print the header line.

`printMethods` — Print out methods

`printMethods(<sort>, <Table>)`

`printMethods(<sort>, Tree)`

If no sorting function is given, `sort` is used as default.

Similar as above, using `Tree` provides only that the names of the entries are inserted into a tree, an element of the domain `adt::Tree`. The tree is both printed out and returned by the method.

Using neither `Table` nor `Tree` the function does the same as `dom::printMethods(sort, Table)`.

`subs` — Avoid substitution

`subs(x, , …)`

Sub-domains should provide a new implementation of this method with sensible semantics if possible.

`subsex` — Avoid extended substitution

```
subsex(x, , …)
```

Sub-domains should provide a new implementation of this method with sensible semantics if possible.

`undefinedEntries` — Return missing entries

```
undefinedEntries()
```

An entry is missing if it should have a definition according to a category of the domain, but the definition is not present.

`whichEntry` — Return the domain or category implementing an entry

```
whichEntry(e)
```

`FAIL` is returned if no entry with the given name is defined for this domain.

Dom::Complex

Field of complex numbers

Syntax

```
Dom::Complex(x)
```

Description

`Dom::Complex` is the domain of complex constants represented by expressions of type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT` or `DOM_COMPLEX`. An expression of type `DOM_EXPR` is considered a complex number if it is of type `Type::Arithmetical` and if it contains only indeterminates which are of type `Type::ConstantIdents` or if it contains no indeterminates, cf. “Example 2” on page 7-80.

`Dom::Complex` is of category `Cat::Field` due to pragmatism. This domain actually is not a field because `bool(1.0 = float(3) / float(3))` returns `FALSE`, for example.

Elements of `Dom::Complex` are usually not created explicitly. However, if one creates elements using the usual syntax, it is checked whether the input expression can be converted to a number. This means `Dom::Complex` is a facade domain which creates elements of domain type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT`, `DOM_COMPLEX` or `DOM_EXPR`.

`Dom::Complex` has no normal representation, because `0` and `0.0` both represent the zero.

Viewed as a differential ring, `Dom::Complex` is trivial. It only contains constants.

`Dom::Complex` has the domain `Dom::BaseDomain` as its super domain, i.e., it inherits each method which is defined by `Dom::BaseDomain` and not re-implemented by `Dom::Complex`. Methods described below are re-implemented by `Dom::Complex`.

Superdomain

`Dom::ArithmeticalExpression`

Axioms

`Ax::systemRep`, `Ax::efficientOperation("_divide")`, `Ax::efficientOperation("_mult")`, `Ax::efficientOperation("_invert")`

Categories

`Cat::DifferentialRing`, `Cat::Field`

Examples

Example 1

Creating some complex numbers using `Dom::Complex`:

```
Dom::Complex(2/3)
```

$$\frac{2}{3}$$

```
Dom::Complex(2/3 + 4*I)
```

$$\frac{2}{3} + 4\,i$$

Example 2

It's also possible to use expressions or constants for creating an element of `Dom::Complex`:

```
Dom::Complex(PI)
```

$$\pi$$

```
Dom::Complex(sin(2))
```

$$\sin(2)$$

```
Dom::Complex(sin(2/3*I) + 3)
```

$$3 + \sinh\left(\frac{2}{3}\right) i$$

If the expression cannot be converted to an element of `Dom::Complex` we will get an error message:

```
Dom::Complex(sin(x))
```

```
Error: The arguments are invalid. [Dom::Complex::new]
```

Parameters

x

An expression of type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT`, `DOM_COMPLEX`. An expression of type `DOM_EXPR` is also possible if it is of type `Type::Arithmetical` and if it contains only indeterminates which are of type `Type::ConstantIdents` or if it contains no indeterminates.

Entries

"characteristic"	the characteristic of this field is 0.
"one"	the unit element; it equals `1`.
"zero"	The zero element; it equals `0`.

Methods

Mathematical Methods

`_divide` — Divide numbers

```
_divide(x, y)
```

`_invert` — Invert numbers

```
_invert(x)
```

`_mult` — Multiplie numbers

```
_mult(x, y, …)
```

`_negate` — Negate numbers

```
_negate(x)
```

`_plus` — Add numbers

```
_plus(x, y, …)
```

`_power` — Power operator

```
_power(x, y)
```

`_unequal` — Inequalities

```
_unequal(x, y)
```

`conjugate` — Conversion to a basic type

```
conjugate(x)
```

`D` — Differential operator

```
D(x)
```

`diff` — Differentiates

```
diff(z, <x, …>)
```

`equal` — Equations

```
equal(x, y)
```

`expr` — Conversion to a basic type

```
expr(x)
```

`iszero` — Zero test

```
iszero(x)
```

`norm` — Absolute value of a number

```
norm(x)
```

`random` — Random number generation

```
random()
```

```
random(n)
```

```
random(m .. n)
```

`random(n)` returns a random number generator which creates complex random numbers where the real parts and the imaginary parts are positive integers between 0 and n - 1.

`random(m..n)` returns a random number generator which creates complex random numbers where the real parts and the imaginary parts are positive integers between m and n.

`unequal` — Inequalities

```
unequal(x, y)
```

Conversion Methods

`convert` — Conversion into this domain

```
convert(x)
```

An arithmetical expression can be converted if it only contains subexpression of the types just mentioned.

If the conversion fails, `FAIL` is returned.

`convert_to` — Conversion to other domains

```
convert_to(x, T)
```

If the conversion fails, FAIL is returned.

The following domains are allowed for T: DOM_INT, Dom::Integer, DOM_RAT, Dom::Rational, DOM_FLOAT, Dom::Float, Dom::Numerical, DOM_COMPLEX and DOM_EXPR.

normal — Normal form of objects

```
normal(x)
```

See Also

MuPAD Domains

Dom::Float | Dom::Integer | Dom::Numerical | Dom::Rational | Dom::Real

Dom::DenseMatrix

Matrices

Syntax

Domain Creation

```
Dom::DenseMatrix(<R>)
```

Element Creation

```
Dom::DenseMatrix(R)(Array)

Dom::DenseMatrix(R)(List)

Dom::DenseMatrix(R)(ListOfRows)

Dom::DenseMatrix(R)(Matrix)

Dom::DenseMatrix(R)(m, n)

Dom::DenseMatrix(R)(m, n, Array)

Dom::DenseMatrix(R)(m, n, List)

Dom::DenseMatrix(R)(m, n, ListOfRows)

Dom::DenseMatrix(R)(m, n, f)

Dom::DenseMatrix(R)(m, n, List, Diagonal)

Dom::DenseMatrix(R)(m, n, g, Diagonal)

Dom::DenseMatrix(R)(m, n, List, Banded)

Dom::DenseMatrix(R)(1, n, List)

Dom::DenseMatrix(R)(m, 1, List)
```

Description

Domain Creation

`Dom::DenseMatrix(R)` creates domains of matrices over a component domain `R` of category `Cat::Rng` (a ring, possibly without unit).

If the optional parameter `R` is not given, the domain `Dom::ExpressionField()` is used.

A vector with n entries is either an $n \times 1$ matrix (a column vector), or a $1 \times n$ matrix (a row vector).

Arithmetical operations with matrices can be performed by using the standard arithmetical operators of MuPAD.

E.g., if `A` and `B` are two matrices defined by `Dom::DenseMatrix(R)`, `A + B` computes the sum, and `A * B` computes the product of the two matrices, provided that the dimensions are correct.

Similarly, `A^(-1)` or `1/A` computes the inverse of a square matrix `A` if it exists, and returns `FAIL` otherwise. See “Example 1” on page 7-89.

Many system functions have been overloaded for matrices, such as `map`, `subs`, `has`, `zip`, `conjugate` to compute the complex conjugate of a matrix, `norm` to compute matrix norms, or `exp` to compute the exponential of a matrix.

Most of the functions in the MuPAD linear algebra package `linalg` work with matrices. For example, the command `linalg::gaussJordan(A)` performs Gauss-Jordan elimination on `A` to transform `A` to its reduced row echelon form.

The domain `Dom::DenseMatrix(R)` represents matrices over `R` of arbitrary size, and it therefore does not have any algebraic structure (other than being a *set* of matrices).

The domain `Dom::SquareMatrix(n, R)` represents the *ring* of $n \times n$ matrices over R. The domain `Dom::MatrixGroup(m, n, R)` represents the *Abelian group* of $m \times n$ matrices over R.

We use the following notations for a matrix A (an element of `Dom::DenseMatrix(R)`):

- *nrows*(A) denotes the number of rows of A.
- *ncols*(A) denotes the number of columns of A.

- A *row index* is an integer in the range from 1 to *nrows*(*A*).
- A *column index* is an integer in the range from 1 to *ncols*(*A*).

Note: The components of a matrix are no longer evaluated after the creation of the matrix, i.e., if they contain free identifiers they will not be replaced by their values.

Element Creation

`Dom::DenseMatrix(R)(Array)` and `Dom::DenseMatrix(R)(Matrix)` create a new matrix with the dimension and the components of `Array` and `Matrix`, respectively.

The components of `Array` or `Matrix` are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

`Dom::DenseMatrix(R)(List)` creates an $m \times 1$ column vector with components taken from the nonempty list, where m is the number of entries of `List`.

`Dom::DenseMatrix(R)(ListOfRows)` creates an $m \times n$ matrix with components taken from the nested list `ListOfRows`, where m is the number of inner lists of `ListOfRows`, and n is the maximal number of elements of an inner list. Each inner list corresponds to a row of the matrix. Both m and n must be non-zero.

If an inner list has less than n entries, then the remaining components in the corresponding row of the matrix are set to zero.

The entries of the inner lists are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

It might be a good idea first to create a two-dimensional array from that list before calling `Dom::DenseMatrix(R)`. This is due to the fact that creating a matrix from an array is the fastest way one can achieve. However, in this case the sublists must have the same number of elements.

The call `Dom::DenseMatrix(R)(m, n)` returns the $m \times n$ zero matrix.

Use the method `"identity"` to create the $n \times n$ identity matrix.

The call `Dom::DenseMatrix(R)(m, n, Array)` creates an $m \times n$ matrix with components taken from `Array`, which must be an array or an hfarray. `Array` must have $m\,n$ operands. The first m operands define the first row, the next m operands define the

second row, etc. The formatting of the array is irrelevant. E.g., any array with 6 elements can be used to create a matrix of dimension 1 ×6, or 2×3, or 3×2, or 6 ×1.

The call `Dom::DenseMatrix(R)(m, n, List)` creates an $m \times n$ matrix with components taken from the list `List` with $m\,n$ elements. The first m elements of the list define the first row, the next m elements of the list define the second row, etc.

The call `Dom::DenseMatrix(R)(m, n, ListOfRows)` creates an $m \times n$ matrix with components taken from the list`ListOfRows`.

If $m \geq 2$ and $n \geq 2$, then `ListOfRows` must consist of at most `m` inner lists, each having at most `n` entries. The inner lists correspond to the rows of the returned matrix.

If an inner list has less than `n` entries, then the remaining components of the corresponding row of the matrix are set to zero. If there are less than `m` inner lists, then the remaining lower rows of the matrix are filled with zeroes.

`Dom::DenseMatrix(R)(m, n, f)` returns the matrix whose (i, j)th component is the value of the function call `f(i,j)`. The row index i ranges from 1 to m and the column index j from 1 to n.

The function values are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

`Dom::DenseMatrix(R)(1, n, List)` returns the 1 ×n row vector with components taken from `List`. The list `List` must have at most `n` entries. If there are fewer entries, then the remaining vector components are set to zero.

The entries of the list are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

`Dom::DenseMatrix(R)(m, 1, List)` returns the $m \times 1$ column vector with components taken from `List`. The list `List` must have at most `m` entries. If there are fewer entries, then the remaining vector components are set to zero.

The entries of the list are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

Superdomain

`Dom::BaseDomain`

Axioms

If R has `Ax::canonicalRep`, then `Ax::canonicalRep`.

Categories

`Cat::Matrix(R)`.

Examples

Example 1

First we create the domain of matrices over the field of rational numbers:

```
MatQ := Dom::DenseMatrix(Dom::Rational)
```

```
Dom::DenseMatrix(Dom::Rational)
```

We assigned this domain to the identifier `MatQ`. Next we define the 2×2 matrix

$$\begin{pmatrix} 1 & 5 \\ 2 & 3 \end{pmatrix}$$

by a list of two rows, where each row is a list of two elements:

```
A := MatQ([[1, 5], [2, 3]])
```

$$\begin{pmatrix} 1 & 5 \\ 2 & 3 \end{pmatrix}$$

In the same way we define the following 2 ×3 matrix:

```
B := MatQ([[-1, 5/2, 3], [1/3, 0, 2/5]])
```

$$\begin{pmatrix} -1 & \frac{5}{2} & 3 \\ \frac{1}{3} & 0 & \frac{2}{5} \end{pmatrix}$$

and perform matrix arithmetic using the standard arithmetical operators of MuPAD, e.g., the matrix product $A\,B$, the 4th power of A as well as the scalar multiplication of A times $\frac{1}{3}$:

```
A * B, A ^ 4, 1/3 * A
```

$$\begin{pmatrix} \frac{2}{3} & \frac{5}{2} & 5 \\ -1 & 5 & \frac{36}{5} \end{pmatrix}, \begin{pmatrix} 281 & 600 \\ 240 & 521 \end{pmatrix}, \begin{pmatrix} \frac{1}{3} & \frac{5}{3} \\ \frac{2}{3} & 1 \end{pmatrix}$$

The matrices A and B have different dimensions, and therefore the sum of A and B is not defined. MuPAD issues an error message:

```
A + B
```

```
Error: The dimensions do not match. [(Dom::DenseMatrix(Dom::Rational))::_plus]
```

To compute the inverse of A, just enter:

```
1/A
```

$$\begin{pmatrix} -\frac{3}{7} & \frac{5}{7} \\ \frac{2}{7} & -\frac{1}{7} \end{pmatrix}$$

If a matrix is not invertible, `FAIL` is the result of this operation. For example, the matrix:

```
C := densematrix(2, 2, [[2]])
```

$$\begin{pmatrix} 2 & 0 \\ 0 & 0 \end{pmatrix}$$

is not invertible, hence:

```
C^(-1)
```

FAIL

Example 2

We create the domain of matrices over the reals:

```
MatR := Dom::DenseMatrix(Dom::Real)
```

$$\text{Dom::DenseMatrix}(\text{Dom::Real})$$

Beside standard matrix arithmetic, the library `linalg` offers a lot of functions dealing with matrices. For example, if one wants to compute the rank of a matrix, use `linalg::rank`:

```
A := MatR([[1, 2], [2, 4]])
```

$$\begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix}$$

```
linalg::rank(A)
```

$$1$$

Use `linalg::eigenvectors` to compute eigenvalues and eigenvectors of the matrix *A*:

```
linalg::eigenvectors(A)
```

$$\left[\left[0,\ 1,\ \left[\begin{pmatrix} -2 \\ 1 \end{pmatrix}\right]\right],\ \left[5,\ 1,\ \left[\begin{pmatrix} \frac{1}{2} \\ 1 \end{pmatrix}\right]\right]\right]$$

Try `info(linalg)` for a list of available functions, or enter `help(linalg)` for details about the library `linalg`.

Some of the functions in the `linalg` package simply serve as "interface" functions for methods of a matrix domain described above. For example, `linalg::transpose` uses the method `"transpose"` to get the transposed matrix. The function `linalg::gaussElim` applies Gaussian elimination to a matrix, such as:

```
linalg::gaussElim(A)
```

$$\begin{pmatrix} 1 & 2 \\ 0 & 0 \end{pmatrix}$$

The computation is performed by the method `"gaussElim"` as described above. Such functions of the `linalg` packages, in contrast to the corresponding methods of the domain `Dom::DenseMatrix(R)`, check their incoming parameters, and some of them offer extended functionalities.

Example 3

In this example, we use the default matrix domain which is created by `Dom::DenseMatrix()`. This domain represents matrices whose components can be arbitrary arithmetical expressions (i.e., the component ring is the domain `Dom::ExpressionField()`).

This domain is already known to MuPAD by the name `matrix`:

```
A := densematrix(
  [[1, 2, 3, 4], [2, 0, 4, 1], [-1, 0, 5, 2]]
)
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 0 & 4 & 1 \\ -1 & 0 & 5 & 2 \end{pmatrix}$$

```
domtype(A)
```

Dom::DenseMatrix()

Matrix components can be extracted by the index operator `[]`:

```
A[2, 1] * A[1, 2] - A[3, 1] * A[1, 3]
```

7

If one of the indices is not in its valid range, an error message is issued. Assignments to matrix components are performed similarly:

```
delete a:
```

```
A[1, 2] := a^2: A
```

$$\begin{pmatrix} 1 & a^2 & 3 & 4 \\ 2 & 0 & 4 & 1 \\ -1 & 0 & 5 & 2 \end{pmatrix}$$

Beside the usual indexing of matrix components, it is also possible to extract submatrices from a given matrix. The following call creates the submatrix of A which consists of the rows 2 to 3 and columns 1 to 3 of A:

```
A[2..3, 1..3]
```

$$\begin{pmatrix} 2 & 0 & 4 \\ -1 & 0 & 5 \end{pmatrix}$$

The index operator does not allow to insert submatrices into a given matrix. This is implemented by the function `linalg::substitute`.

Example 4

In the following examples, we demonstrate the different ways of creating matrices. We work with matrices defined over the field $\mathbb{Z}_{19}$, i.e., the field of integers modulo 19. This component ring can be created with the domain constructor `Dom::IntegerMod`.

We start by giving a list of rows, where each row is a list of row entries:

```
MatZ19 := Dom::DenseMatrix(Dom::IntegerMod(19)):
MatZ19([[1, 2], [2]])
```

$$\begin{pmatrix} 1 \bmod 19 & 2 \bmod 19 \\ 2 \bmod 19 & 0 \bmod 19 \end{pmatrix}$$

The elements of the two inner lists, the row entries, were converted into elements of the domain `Dom::IntegerMod(19)`.

The number of rows is the number of sublists of the argument, i.e., $m = 2$. The number of columns is determined by the length of the inner list with the most entries, which is

the first inner list with two entries. Missing entries in the other inner lists are treated as zero components. The call:

```
MatZ19(4, 4, [[1, 2], [2]])
```

$$\begin{pmatrix} 1 \bmod 19 & 2 \bmod 19 & 0 \bmod 19 & 0 \bmod 19 \\ 2 \bmod 19 & 0 \bmod 19 & 0 \bmod 19 & 0 \bmod 19 \\ 0 \bmod 19 & 0 \bmod 19 & 0 \bmod 19 & 0 \bmod 19 \\ 0 \bmod 19 & 0 \bmod 19 & 0 \bmod 19 & 0 \bmod 19 \end{pmatrix}$$

fixes the dimension of the matrix. Missing entries and inner lists are treated as zero components and zero rows, respectively.

An error message is issued if one of the given entries cannot be converted into an element over $\mathbb{Z}_{19}$:

```
MatZ19([[2, 3], [-1, I]])
```

```
Error: Cannot define a matrix over 'Dom::IntegerMod(19)'. [(Dom::DenseMatrix(Dom::Integ
```

Example 5

This example illustrates how to create a matrix with components given as values of an index function. First we create the 2 ×2 Hilbert matrix (see also the functions `linalg::hilbert` and `linalg::invhilbert`):

```
densematrix(2, 2, (i, j) -> 1/(i + j - 1))
```

$$\begin{pmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{3} \end{pmatrix}$$

Note the difference when working with expressions and functions. If you give an expression it is treated as a function in the row and column indices:

```
delete x:
densematrix(2, 2, x), densematrix(2, 2, (i, j) -> x)
```

$$\begin{pmatrix} x(1,1) & x(1,2) \\ x(2,1) & x(2,2) \end{pmatrix}, \begin{pmatrix} x & x \\ x & x \end{pmatrix}$$

Example 6

Diagonal matrices can be created with the option `Diagonal` and a list of diagonal components:

```
MatC := Dom::DenseMatrix(Dom::Complex):
MatC(3, 4, [1, 2, 3], Diagonal)
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \end{pmatrix}$$

Hence, to define the $n \times n$ identity matrix, you can enter:

```
MatC(3, 3, [1 $ 3], Diagonal)
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

or even call:

```
MatC(3, 3, x -> 1, Diagonal)
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The easiest way to create the identity matrix, however, is to use the method `"identity"`:

```
MatC::identity(3)
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Example 7

Toeplitz matrices can be defined with the option `Banded`. The following call defines a three-banded matrix with the component 2 on the main diagonal and the component - 1 on the first subdiagonals:

```
densematrix(4, 4, [-1, 2, -1], Banded)
```

$$\begin{pmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{pmatrix}$$

Example 8

Some system functions can be applied to matrices, such as `norm`, `expand`, `diff`, `conjugate`, or `exp`.

For example, to expand the components of the matrix:

```
delete a, b:
A := densematrix(
  [[(a - b)^2, a^2 + b^2], [a^2 + b^2, (a - b)*(a + b)]]
)
```

$$\begin{pmatrix} (a-b)^2 & a^2+b^2 \\ a^2+b^2 & (a+b)\,(a-b) \end{pmatrix}$$

enter:

```
expand(A)
```

$$\begin{pmatrix} a^2-2\,a\,b+b^2 & a^2+b^2 \\ a^2+b^2 & a^2-b^2 \end{pmatrix}$$

If you want to differentiate the matrix components, then call for example:

```
diff(A, a)
```

$$\begin{pmatrix} 2a-2b & 2a \\ 2a & 2a \end{pmatrix}$$

To substitute matrix components by some values, enter:

```
subs(A, a = 1, b = -1)
```

$$\begin{pmatrix} 4 & 2 \\ 2 & 0 \end{pmatrix}$$

The function `zip` can also be applied to matrices. The following call combines two matrices A and B by dividing each component of A by the corresponding component of B:

```
A := densematrix([[4, 2], [9, 3]]):
B := densematrix([[2, 1], [3,-1]]):
zip(A, B, `/`)
```

$$\begin{pmatrix} 2 & 2 \\ 3 & -3 \end{pmatrix}$$

The quoted character `` `/` `` is another notation for the function `_divide`, the functional form of the division operator `/`.

If one needs to apply a function to the components of a matrix, then use the function `map`. For example, to simplify the components of the matrix:

```
C := densematrix(
  [[sin(x)^2 + cos(x)^2, cos(x)*tan(x)],
  [(a^2 - b^2)/(a + b), 1]]
)
```

$$\begin{pmatrix} \cos(x)^2+\sin(x)^2 & \cos(x)\tan(x) \\ \frac{a^2-b^2}{a+b} & 1 \end{pmatrix}$$

call:

```
map(C, Simplify)
```

$$\begin{pmatrix} 1 & \sin(x) \\ a-b & 1 \end{pmatrix}$$

Example 9

However, there may appear some unexpected results using the function `diff` in the context of matrices. The derivative of the following unspecified function `f` of a matrix is computed due to the chain rule:

```
diff(f(densematrix([[a*x^2, b], [c, d]])), x)
```

$$\begin{pmatrix} 2\,a\,x\,f'\left(\begin{pmatrix} a\,x^2 & b \\ c & d \end{pmatrix}\right) & 0 \\ 0 & 0 \end{pmatrix}$$

Usually, the function `f` would implicitly be assumed to be scalar. Hence, the derivative of `f` should be scalar as well. In the above situation the chain rule is applied for differentiation: the inner function is the matrix containing the symbolic components `a*x^2`, `b`, `c` and `d`. Its derivative is computed by simply applying `diff` to each component of the matrix:

```
diff(densematrix([[a*x^2, b], [c, d]]), x)
```

$$\begin{pmatrix} 2\,a\,x & 0 \\ 0 & 0 \end{pmatrix}$$

Finally, the exterior unspecified function `f` is implicitly assumed to be scalar, such that each component of the derivative of the inner function is multiplied by the exterior differentiation.

Example 10

A column vector is represented as a 2×1 matrix:

```
MatR := Dom::DenseMatrix(Dom::Real):
v := MatR(2, 1, [1, 2])
```

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

The dimension of this vector is:

```
MatR::matdim(v)
```

$$[2, 1]$$

Use `linalg::vecdim`, or even call `nops(v)` to get the length of a vector:

```
linalg::vecdim(v)
```

$$2$$

The *i*th component of this vector can be extracted in two ways: either by `v[i,1]` or by `v[i]`:

```
v[1], v[2]
```

$$1, 2$$

We get the 2-norm of `v` by the following call:

```
norm(v, 2)
```

$$\sqrt{5}$$

Parameters

R

A ring, i.e., a domain of category `Cat::Rng`; default is `Dom::ExpressionField()`

`Array`

A one- or two-dimensional `array` or `hfarray`

`Matrix`

A matrix, i.e., an element of a domain of category `Cat::Matrix`

`m, n`

Matrix dimension (positive integers)

`List`

A list of matrix components

`ListOfRows`

A list of at most `m` rows; each row given as a list of at most `n` matrix components

`f`

A function or a functional expression with two parameters (the row and column index)

`g`

A function or a functional expression with one parameter (the row index)

Options

`Diagonal`

Create a diagonal matrix

With the option `Diagonal`, diagonal matrices can be created with diagonal elements taken from a list, or computed by a function or a functional expression.

`Dom::DenseMatrix(R)(m, n, List, Diagonal)` creates the $m \times n$ diagonal matrix whose diagonal elements are the entries of `List`.

`List` must have at most $min(m, n)$ entries. If it has fewer elements, the remaining diagonal elements are set to zero.

The entries of `List` are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

`Dom::DenseMatrix(R)(m, n, g, Diagonal)` returns the matrix whose ith diagonal element is `g(i, i)`, where the index i runs from 1 to $min(m, n)$.

The function values are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

`Banded`

Create a banded Toeplitz matrix

`Dom::DenseMatrix(R)(m, n, List, Banded)` creates an $m \times n$ banded Toeplitz matrix with the elements of `List` as entries. The number of entries of `List` must be odd, say $2h + 1$, and must not exceed `n`. The resulting matrix has bandwidth at most $2h + 1$.

A Toeplitz matrix is a matrix where the elements of each band are identical. See also "Example 7" on page 7-96.

All elements of the main diagonal of the created matrix are initialized with the middle element of `List`. All elements of the ith subdiagonal are initialized with the $(h + 1 - i)$th element of `List`. All elements of the ith superdiagonal are initialized with the $(h + 1 + i)$th element of `List`. All entries on the remaining sub- and superdiagonals are set to zero.

The entries of `List` are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

Entries

"isSparse"	is always `FALSE`, as elements of `Dom::DenseMatrix(R)` use a dense representation of their matrix components.
"randomDimen"	is set to `[10,10]`. See the method `"random"` below for details.

Methods

Mathematical Methods

_divide — Divide matrices

```
_divide(A, B)
```

An error message is issued if the dimensions of A and B do not match.

This method only exists if R is a commutative ring with a unit, i.e., a domain of category Cat::Ring.

This method overloads the function _divide for matrices, i.e., one may use it in the form A / B, or in functional notation: _divide(A, B).

_invert — Compute the inverse of a matrix

```
_invert(A, Normal = b)
```

If the component ring R is the domain Dom::Float, a floating-point approximation of the inverse matrix is computed by the function numeric::inverse.

This method only exists if R is a domain of category Cat::Ring.

This method overloads the function _invert for matrices, i.e., one may use it in the form 1/A or A^(-1), or in functional notation: _invert(A).

If Normal = TRUE, then the matrix inverse is always returned in a normalized form. For details about normalization, see normal. If Normal = FALSE, then the matrix inverse can appear in a normalized form, but normalization is not guaranteed. By default Normal = TRUE.

Normal affects the results only if a matrix contains variables or exact expressions, such as sqrt(5) or sin(PI/7).

_mod — Map the modulo operator to a matrix

```
_mod(A, n)
```

n must be non-zero, and a mod n must be defined for every entry a of A.

This method overloads the function `mod` for matrices; one may use it in the form `A mod n`, or in functional notation: `_mod(A, n)`.

`_mult` — Multiplie matrices by matrices, vectors and scalars

```
_mult(x, y)
```

```
_mult(x, y)
```

If `y` is of the domain type `R` or can be converted into such an element, the corresponding scalar multiplication is computed.

Otherwise, `y` is converted into a matrix of the domain type of `x`. If this conversion fails, then this method calls the method `"_mult"` of the domain of `y` giving all arguments in the same order.

If `x` is a matrix of the same domain type as `y`, then the matrix product $x\,y$ is computed. An error message is issued if the dimensions of the matrices do not match.

If `x` is of the domain type `R` or can be converted into such an element, the corresponding scalar multiplication is computed.

Otherwise, `x` is converted into a matrix of the domain type of `y`. If this conversion fails, then `FAIL` is returned.

This method handles more than two arguments by calling itself recursively with the first half of all arguments and the last half of all arguments. Then the product of these two results is computed with the system function `_mult`.

This method overloads the function `_mult` for matrices, i.e., one may use it in the form `x * y`, or in functional notation: `_mult(x, y)`.

`_negate` — Negate a matrix

```
_negate(A)
```

This method overloads the function `_negate` for matrices, i.e., one may use it in the form `-A`, or in functional notation: `_negate(A)`.

`_plus` — Add matrices

```
_plus(A, B, …)
```

The arguments `A, B , ...` are converted into matrices of the domain type `Dom::DenseMatrix(R)`. `FAIL` is returned if one of these conversions fails.

This method overloads the function `_plus` for matrices, i.e., one may use it in the form `A + B`, or in functional notation: `_plus(A, B)`.

`_power` — Integer power of a matrix

```
_power(A, n)
```

If the power `n` is a negative integer then `A` must be nonsingular and `R` must be a domain of category `Cat::IntegralDomain`. Otherwise `FAIL` is returned.

If `n` is zero and the component ring `R` is a ring with no unit (i.e., of category `Cat::Rng`, but not of category `Cat::Ring`), `FAIL` is returned.

This method overloads the function `_power` for matrices, i.e., one may use it in the form `A^n`, or in functional notation: `_power(A, n)`.

`conjugate` — Complex conjugate of a matrix

```
conjugate(A)
```

This method only exists if `R` implements the method `"conjugate"`, which computes the complex conjugate of an element of the domain `R`.

This method overloads the function `conjugate` for matrices, i.e., one may use it in the form `conjugate(A)`.

`cos` — Cosine of a matrix

```
cos(A)
```

If `A` is not square, an error message is issued. If the component domain of `A` does not allow the computation of `cos(elem)` for an arbitrary element `elem` of the component ring, `FAIL` is returned.

This method uses the function `numeric::expMatrix` for a floating-point approximation of the exponential of `A` if `A` is defined over the domain `Dom::Float`.

If some eigenvalues of `A` do not exist in `R` or cannot be computed, then `FAIL` is returned.

In the symbolic case the functions `exp` and `linalg::jordanForm` are called. The latter may not be able to compute the Jordan form of `A`. In this case `FAIL` is returned.

This method only exists if `R` is a domain of category `Cat::Field`.

This method overloads the function `cos` for matrices, i.e., one may use it in the form `cos(A)`.

`diff` — Differentiation of matrix components

```
diff(A, …)
```

This method only exists if `R` implements the method `"diff"`.

This method overloads the function `diff` for matrices, i.e., one may use it in the form `diff(A, ...)`. See “Example 8” on page 7-96 and “Example 9” on page 7-98.

`equal` — Equality test of matrices

```
equal(A, B)
```

Note that if `R` has the axiom `Ax::systemRep` then `normal` is used to simplify the components of `A` and `B` before testing their equality.

`exp` — Exponential of a matrix

```
exp(A, <t>)
```

If `A` is not square, an error message is issued. If the component domain of `A` does not allow the computation of `exp(elem)` for an arbitrary element `elem` of the component ring, `FAIL` is returned.

This method uses the function `numeric::expMatrix` for a floating-point approximation of the exponential of `A` if `A` is defined over the domain `Dom::Float` and if $t = 1$.

If some eigenvalues of `A` do not exist in `R` or cannot be computed, then `FAIL` is returned.

In the symbolic case the function `linalg::jordanForm` is called, which may not be able to compute the Jordan form of `A`. In this case `FAIL` is returned.

This method only exists if `R` is a domain of category `Cat::Field`.

This method overloads the function `exp` for matrices, i.e., one may use it in the form `exp(A, ...)`.

expand — Expand matrix components

```
expand(A)
```

This method only exists if R implements the method "expand", or if R has the axiom Ax::systemRep (in this case, the system function expand is used).

This method overloads the function expand for matrices, i.e., one may use it in the form expand(A).

factor — Scalar-matrix factorization

```
factor(A)
```

The factor s is the gcd of all components of the matrix A. Hence, this method only exists if R is of category Cat::GcdDomain.

This method overloads the function factor for matrices, i.e., one may use it in the form factor(A).

float — Floating-point approximation of the matrix components

```
float(A)
```

This method only exists if R implements the method "float".

Note: Usually the floating-point approximations are not elements of R! For example, Dom::Integer implements such a method, but the floating-point approximation of an integer cannot be re-converted into an integer.

This method checks whether the resulting matrix can be converted into the domain type of A only if testargs returns TRUE (e.g., if one calls this method from the interactive level of MuPAD).

Otherwise, one has to take care that the matrix returned is compatible to its component ring.

---

gaussElim — Gaussian elimination

```
gaussElim(A)
```

If the matrix is not square, i.e., the determinant of A is not defined, then the third entry of the list returned is the value FAIL.

This method only exists if the component ring R is an integral domain, i.e., a domain of category Cat::IntegralDomain.

If R has the method "pivotSize", then the pivot element of smallest size is chosen at every pivoting step, whereby pivotSize must return a positive integer representing the "size" of an element.

If no such method is defined, Gaussian elimination without a pivot strategy is applied to A.

If R has the axiom Ax::efficientOperation("_invert") and is of category Cat::Field, then ordinary Gaussian elimination is used. Otherwise, fraction-free elimination is performed on A.

If R implements the method "normal", it is used to simplify subsequent computations of the Gaussian elimination process.

Note that if R does not implement the method "normal", but the elements of R are represented by kernel domains, i.e., R has the axiom Ax::systemRep, the system function normal is used instead.

identity — Identity matrix

```
identity(n)
```

This method only exists if the component ring R is of category Cat::Ring, i.e., a ring with unit.

int — Integration of matrix components

```
int(A, …)
```

This method only exists if R implements the method "int".

This method overloads the system function int for matrices, i.e., one may use it in the form int(A, ...).

iszero — Test for zero matrices

```
iszero(A)
```

Note that there may exist more than one representation of the zero matrix of a given dimension if R does not have Ax::canonicalRep.

If R implements the method "normal", it is used to simplify the components of A for the zero-test.

Note that if R does not implement such a method, but the elements of R are represented by kernel domains, i.e., R has the axiom Ax::systemRep, the system function normal is used instead.

This method overloads the function iszero for matrices, i.e., one may use it in the form iszero(A).

matdim — Matrix dimension

```
matdim(A)
```

norm — Norm of matrices and vectors

```
norm(A, Infinity)

norm(A, Maximum)

norm(v, Infinity)

norm(v, Maximum)

norm(A, Frobenius)

norm(A, 1)

norm(v, Euclidean)

norm(v, k)
```

norm(A, Maximum) computes the maximum norm of the matrix A, which is the maximum row sum (the row sum is the sum of norms of each component in a row).

If the domain R does not implement the methods "max" and "norm", FAIL is returned.

Using norm(v, Infinity) for a vector v the maximum norm of all elements is returned.

If the domain R does not implement the methods "max" and "norm", FAIL is returned.

Using `norm(v, Maximum)` for a vector `v` the maximum norm of all elements is returned.

If the domain `R` does not implement the methods `"max"` and `"norm"`, `FAIL` is returned.

`norm(A, Frobenius)` computes the Frobenius norm of `A`, which is the square root of the sum of the squares of the norms of each component.

If the result is no longer an element of the domain `R`, or if `R` does not implement the method `"norm"`, `FAIL` is returned.

`norm(A, 1)` computes the 1-norm of the matrix `A`, which is the maximum sum of the norms of the elements of each column. If `R` does not implement the methods `"max"` and `"norm"`, `FAIL` is returned.

`norm(v, Euclidean)` computes the Euclidean norm (2-norm) of the vector `v`, which is defined to be the square root of the sum of the norms of the elements of `v` raised to the square.

`FAIL` is returned if the result is no longer an element of the domain `R`. The function `linalg::scalarProduct` is used to compute the Euclidean norm of the vector `v`.

If `R` does not implement the method `"norm"`, `FAIL` is returned.

`norm(v, k)` computes the k-norm of the vector `v`, which is defined to be the kth root of the sum of the norms of the elements of `v` raised to the kth power.

`FAIL` is returned if the result is no longer an element of the domain `R`. For $k = 2$, the function `linalg::scalarProduct` is used to compute the 2-norm of `v`.

If `R` does not implement the method `"norm"`, `FAIL` is returned.

The method `norm` overloads the function `norm` for matrices, i.e., one may use it in the form `norm(A k )`, where `k` is either `Infinity`, `Frobenius`, or a positive integer. The default value of `k` is `Infinity`.

`normal` — Simplification of matrix components

`normal(A)`

If `R` does not implement the method `"normal"`, but the elements of `R` are represented by kernel domains, i.e., `R` has the axiom `Ax::systemRep`, then the system function `normal` is applied to the components of `A`. Otherwise `normal(A)` returns `A` without any changes.

This method overloads the function `normal` for matrices, i.e., one may use it in the form `normal(A)`.

`nonZeros` — Number of non-zero components of a matrix

```
nonZeros(A)
```

`nonZeroes` — Number of non-zero components of a matrix

```
nonZeroes(A)
```

`nonZeroOperands` — Return a sequence of all non-zero operands

```
nonZeroOperands(A)
```

This method is useful for retrieving information on the non-zero entries. For example, to find out the types of the entries in the matrix, one should not consider all operands `op(A)`, because this would also involve the zero entries. For large matrices with few entries, it is much more efficient to use this method to extract the entries.

`random` — Random matrix generation

```
random()
```

This method only exists if `R` implements the method `"random"`.

The dimension of the matrix is also chosen randomly, but it is limited by the values given in `"randomDimen"` (see “Entries” above).

To change the value of the entry `"randomDimen"` for a domain `MatR` created with `Dom::DenseMatrix`, one must first unprotect the domain `Dom` (see `unprotect` for details).

`sin` — Sine of a matrix

```
sin(A)
```

If `A` is not square, an error message is issued. If the component domain of `A` does not allow the computation of `sin(elem)` for an arbitrary element `elem` of the component ring, `FAIL` is returned.

This method uses the function `numeric::expMatrix` for a floating-point approximation of the exponential of `A` if `A` is defined over the domain `Dom::Float`.

If some eigenvalues of A do not exist in R or cannot be computed, then FAIL is returned.

In the symbolic case the functions exp and linalg::jordanForm are called. The latter may not be able to compute the Jordan form of A. In this case FAIL is returned.

This method only exists if R is a domain of category Cat::Field.

This method overloads the function sin for matrices, i.e., one may use it in the form sin(A).

sqrt — Square root of a matrix

```
sqrt(A, <sqrtfunc>)
```

Returned is a matrix B with $B^2 = A$ such that the eigenvalues of B are the square roots of the eigenvalues of A or FAIL if the square root of the matrix does not exist. For computing the square roots of the eigenvalues a function satisfying $sqrtfunc(a)^2 = a$ for every element a of the coefficient ring of A can be given as optional second argument.

For details we refer to the help page of the function linalg::sqrtMatrix.

testeq — Testing for equality of two matrices

```
testeq(A, B)
```

tr — Trace of a square matrix

```
tr(A)
```

If A is not square, then an error message is issued.

transpose — Transpose of a matrix

```
transpose(A)
```

Access Methods

_concat — Horizontal concatenation of matrices

```
_concat(A, B, …)
```

An error message is issued if the given matrices do not have the same number of rows.

This method overloads the function `_concat` for matrices, i.e., one may use it in the form `A . B . ...`, or in functional notation: `_concat(A, B, ...)`.

`_index` — Matrix indexing

```
_index(A, i, j)

_index(A, r1 .. r2, c1 .. c2)

_index(v, i)

_index(v, i1 .. i2)
```

If `i` and `j` are not integers, then the call of this method returns in its symbolic form (of type `"_index"`) with evaluated arguments.

Otherwise an error message is given, if `i` and `j` are not valid row and column indices, respectively.

Note: Note that this method uses the system function `context` to evaluate the entry in the context of the calling environment.

`_index(A,r1..r2,c1..c2)` returns the submatrix of `A` created by the rows of `A` with indices from `r1` to `r2` and the columns of `A` with indices from `c1` to `c2`.

`_index(v,i)` returns the *i*th entry of the vector `v`.

An error message is issued if `v` is not a vector.

If `i` is not an integer, then the call of this method returns in its symbolic form (of type `"_index"`) with evaluated arguments.

Otherwise an error message is given, if `i` is less than one or greater than the dimension of `v`.

Note: Note that this method uses the system function `context` to evaluate the entry in the context of the calling environment.

`_index(v,i1..i2)` returns the subvector of `v`, formed by the entries with index `i1` to `i2`. See also the method `"op"`.

An error message is issued if `v` is not a vector.

This method overloads the function `_index` for matrices, i.e., one may use it in the form `A[i, j]`, `A[r1..r2, c1..c2]`, `v[i]` and `v[i1..i2]`, respectively, or in functional notation: `_index(A, ...)`.

`concatMatrix` — Horizontal concatenation of matrices

```
concatMatrix(A, B, …)
```

`col` — Extracting a column

```
col(A, c)
```

An error message is issued if `c` is less than one or greater than the number of columns of `A`.

`delCol` — Deleting a column

```
delCol(A, c)
```

`NIL` is returned if `A` consists of only one column.

An error message is issued if `c` is less than one or greater than the number of columns of `A`.

`delRow` — Deleting a row

```
delRow(A, r)
```

`NIL` is returned if `A` consists of only one row.

An error message is issued if `r` is less than one or greater than the number of rows of `A`.

`evalp` — Evaluating matrices of polynomials at a certain point

```
evalp(A, x = a, …)
```

This method is only defined if `R` is a polynomial ring of category `Cat::Polynomial`.

This method overloads the function `evalp` for matrices, i.e., one may use it in the form `evalp(A, x = a)`.

`length` — Length of a matrix

```
length(A)
```

This method overloads the function `length` for matrices, i.e., one may use it in the form `length(A)`.

`map` — Apply a function to matrix components

```
map(A, func, <expr, …>)
```

Note: Note that the function values are converted into elements of the domain `R` only if `testargs` returns `TRUE` (e.g., if one calls this method from the interactive level of MuPAD).

If `testargs` returns `FALSE`, then one must guarantee that the function calls return elements of the domain type `R`, otherwise the resulting matrix, which is of domain type `Dom::DenseMatrix(R)`, would have components which are not elements of the domain `R`!

This method overloads the function `map` for matrices, i.e., one may use it in the form `map(A, func, ...)`.

`mapNonZeroes` — Apply a function to the non-zero components of a matrix

```
mapNonZeroes(A, f, <p1, p2, …>)
```

`nops` — Number of components of a matrix

```
nops(A)
```

This method overloads the function `nops` for matrices, i.e., one may use it in the form `nops(A)`.

`op` — Component of a matrix

```
op(A, i)
```

```
op(A)
```

This method returns an expression sequence of all components of A.

See also the method "_index".

This method overloads the function op for matrices, i.e., one may use it in the form op(A, i) and op(A), respectively.

row — Extracting a row

```
row(A, r)
```

An error message is issued if r is less than one or greater than the number of rows of A.

setCol — Replacing a column

```
setCol(A, c, v)
```

An error message is issued if c is less than one or greater than the number of rows of A.

setRow — Replacing a row

```
setRow(A, r, v)
```

An error message is issued if r is less than one or greater than the number of rows of A.

stackMatrix — Vertical concatenation of matrices

```
stackMatrix(A, B, …)
```

An error message is issued if the given matrices do not have the same number of columns.

subs — Substitution of matrix components

```
subs(A, …)
```

Note: Note that the function values are converted into elements of the domain R only if testargs returns TRUE (e.g., if one calls this method from the interactive level of MuPAD).

If testargs returns FALSE, then one must guarantee that the function calls return elements of the domain type R, otherwise the resulting matrix, which is of domain type

`Dom::DenseMatrix(R)`, would have components which are not elements of the domain `R`!

This method overloads the function `subs` for matrices, i.e., one may use it in the form `subs(A, ...)`.

subsex — Extended substitution of matrix components

`subsex(A, …)`

Note: Note that the results of the substitutions are converted into elements of the domain `R` only if `testargs` returns `TRUE` (e.g., if one calls this method from the interactive level of MuPAD).

If `testargs` returns `FALSE`, then one must guarantee that the results of the substitutions are of the domain type `R`, otherwise the resulting matrix, which is of domain type `Dom::DenseMatrix(R)`, would have components which are not elements of the domain `R`!

This method overloads the function `subsex` for matrices, i.e., one may use it in the form `subsex(A, ...)`.

subsop — Operand substitution of matrix components

`subsop(A, i = x, …)`

Note: Note that `x` is converted into the domain `R` only if `testargs` returns `TRUE` (e.g., if one calls this method from the interactive level of MuPAD).

If `testargs` returns `FALSE`, then `x` must be an element of `R`, otherwise the resulting matrix, which is of domain type `Dom::DenseMatrix(R)`, would have components which are not elements of the domain `R`!

See also the method `"set_index"`.

This method overloads the function `subsop` for matrices, i.e., one may use it in the form `subsop(A, ...)`.

swapCol — Swapping matrix columns

```
swapCol(A, c1, c2)
```

```
swapCol(A, c1, c2, r1 .. r2)
```

An error message is issued if one of the column indices is less than one or greater than the number of columns of `A`.

`swapCol(A,c1,c2,r1..r2)` swaps the column with index `c1` and the column with index `c2` of `A`, but by taking only those column components which lie in the rows with indices `r1` to `r2`.

An error message is issued if one of the column indices is less than one or greater than the number of columns of `A`, or if one of the row indices is less than one or greater than the number of rows of `A`.

swapRow — Swapping matrix rows

```
swapRow(A, r1, r2)
```

```
swapRow(A, r1, r2, c1 .. c2)
```

An error message is issued if one of the row indices is less than one or greater than the number of rows of `A`.

`swapCol(A,r1,r2,c1..c2)` swaps the row with index `r1` and the row with index `r2` of `A`, but by taking only those row components which lie in the columns with indices `c1` to `c2`.

An error message is issued if one of the row indices is less than one or greater than the number of rows of `A`, or if one of the column indices is less than one or greater than the number of columns of `A`.

set_index — Setting matrix components

```
set_index(A, i, j, x)
```

```
set_index(v, i, x)
```

Note: Note that `x` is converted into an element of the domain `R` only if `testargs` returns `TRUE` and `i` and `j` are integers (e.g., if one calls this method from the interactive level of

MuPAD). If x is a matrix of the same type as A or can be converted into a matrix of the same type as A and the indices i or j are ranges corresponding to a submatrix of A, then x replaces the corresponding submatrix in A.

Otherwise one has to take care that x is of domain type R.

See also the method "subsop".

set_index(v,i,x) replaces the *i*th entry of the vector v by x.

set_index on vectors overloads the function set_index for matrices, i.e., one may use it in the form A[i, j] := x and v[i] := x, respectively, or in functional notation: A := set_index(A, i, j, x) or v := set_index(v, i, x).

zip — Combine matrices component-wise

zip(A, B, func, <expr, …>)

The row number of the matrix returned is the minimum of the row numbers of A and B, and its column number is the minimum of the column numbers of A and B.

Note: Note that the function values are converted into elements of the domain R only if testargs returns TRUE (e.g., if one calls this method from the interactive level of MuPAD).

If testargs returns FALSE, then one must guarantee that the function calls return elements of the domain type R, otherwise the resulting matrix, which is of domain type Dom::DenseMatrix(R), would have components which are not elements of the domain R!

This method overloads the function zip for matrices, i.e., one may use it in the form zip(A, B, ...).

Conversion Methods

convert — Conversion to a matrix

convert(x)

FAIL is returned if the conversion fails.

x may either be an array, a matrix, or a list (of sublists, see the parameter ListOfRows in "Creating Elements" above). Their entries must then be convertible into elements of the domain R.

convert_to — Matrix conversion

```
convert_to(A, T)
```

T may either be DOM_ARRAY, DOM_LIST, or a domain constructed by Dom::DenseMatrix or Dom::SquareMatrix. The elements of A must be convertible into elements of the domain R.

Use the function expr to convert A into an object of a kernel domain (see below).

create — Defining matrices without component conversions

```
create(x, …)
```

This method works more efficient than if one creates matrices by calling the method "new" of the domain, because it avoids any conversion of the components. One must guarantee that the components have the correct domain type, otherwise run-time errors can be caused.

If x is a list of sublists, it might be a good idea first to create a two-dimensional array from that list before calling this method. This is due to the fact that creating a matrix from an array is the fastest way one can achieve.

Please note that when creating a two-dimensional array from a list of sublists, the sublists must have the same number of elements.

expr — Matrix conversion into an object of a kernel domain

```
expr(A)
```

The result is an array representing the matrix A where each entry is an object of a kernel domain.

This method overloads the function expr for matrices, i.e., one may use it in the form expr(A).

`expr2text` — Matrix conversion to a string

```
expr2text(A)
```

This method overloads the function `expr2text` for matrices, i.e., one may use it in the form `expr2text(A)`.

`TeX` — TeX formatting of a matrix

```
TeX(A)
```

Note that in the case of very large matrices the output will not be useful. For printing large matrices use the function `"doprint"`.

The method `"TeX"` of the component ring `R` is used to get the TeX-representation of each component of `A`.

This method is used by the function `generate::TeX`.

Technical Methods

`assignElements` — Multiple assignment to matrices

```
assignElements(A, …)
```

The assigned components must have the domain type `R`, an implicit conversion of the components into elements of domain type `R` is not performed.

This method overloads the function `assignElements` for matrices, i.e., one may use it in the form `assignElements(A, ...)`.

`mkDense` — Conversion of a matrix to an array

```
mkDense(Array)
```

```
mkDense(List)
```

```
mkDense(r, c, List)
```

`mkDense(List)` tries to convert the list `List` into an array `a`. The result is either `FAIL` if this is not possible, or the list `[r, c, a]`, where the positive integers `r` and `c` give the

dimension of `a`. See the parameters `List` and `ListOfRows` in "Creating Elements" above for admissible formats of `List`.

The array `a` has dimension one if `r` or `c` is equal to one. The entries of `a` have been converted into elements of the domain `R`.

`mkDense(r,c,List)` tries to convert the list `List` into an array `a` of the dimension `r` times `c`.

The result is either `FAIL` if this is not possible, or the list `[r, c, a]`.

The array `a` has dimension one if `r` or `c` is equal to one. The entries of `a` have been converted into elements of the domain `R`.

`print` — Printing matrices

```
print(A)
```

Note: Note that in general it is not useful to print very large matrices. Hence, a warning message is be displayed if the size of the matrix oversteps a certain dimension – printing such matrices can be done by using the function `"doprint"`.

`doprint` — Printing very large matrices

```
doprint(A)
```

`unapply` — Create a procedure from a matrix

```
unapply(A, <x, …>)
```

This method overloads the function `fp::unapply` for matrices, i.e., one may use it in the form `fp::unapply(A)`.

See Also

MuPAD Domains
`Dom::Matrix` | `Dom::MatrixGroup` | `Dom::SquareMatrix`

Dom::DihedralGroup

Dihedral groups

Syntax

Domain Creation

```
Dom::DihedralGroup(n)
```

Element Creation

```
Dom::DihedralGroupn(l)
```

Description

Domain Creation

`Dom::DihedralGroup(n)` creates the dihedral group of size n, i.e., the group of symmetries of a regular polygon with n edges.

`Dom::DihedralGroup(n)` creates the group of all congruent mappings of the plane that induce a bijective mapping of the set of corners of a regular n-angle to itself.

Element Creation

`Dom::DihedralGroup(n)([a,b])` represents the group element "t^a carried out after r^b", where r is a rotation that maps each corner to its left neighbor, and t is a reflection w.r.t. some fixed central diagonal.

Superdomain

`Dom::BaseDomain`

Axioms

`Ax::canonicalRep`

Categories

`Cat::Group`

Examples

Example 1

Define the group D_6, i.e., the group of congruence mappings of the hexagon:

```
G := Dom::DihedralGroup(6)
```

```
Dom::DihedralGroup(6)
```

Then elements may be created as follows:

```
a := G([7, 19]);
```

```
[1, 1]
```

This means that 19 rotations—mapping each corner to its left neighbor—and 7 reflections have the same effect as one operation of either type.

Parameters

n

Positive integer

`l`

List or array of two integers

Entries

"size"	the number of elements, which equals $2n$.
"one"	the mapping leaving each point fixed.

Methods

Mathematical Methods

`_mult` — Functional composition of elements

```
_mult(a, …)
```

This method overloads the kernel function `_mult`.

`_invert` — Inverse of an element

```
_invert(a)
```

This method overloads the kernel function `_invert`.

`_power` — Power of an element

```
_power(a, n)
```

It overloads the kernel function `_power`.

`order` — Order of a group element

```
order(a)
```

`random` — Random element

```
random()
```

Conversion Methods

`expr` — Convert group element to list

```
expr(a)
```

`TeX` — TeX output of a group element

```
TeX(a)
```

`equiv` — Test for equivalence

Inherited from `Cat::BaseCategory`.

`new` — Create element of this domain

Inherited from `Cat::BaseCategory`.

`coerce` — Coerce into this domain

Inherited from `Cat::BaseCategory`.

`hasProp` — Test for a certain property

Inherited from `Dom::BaseDomain`.

`whichEntry` — Return the domain or category implementing an entry

Inherited from `Dom::BaseDomain`.

`isone` — Test if element is one

Inherited from `Cat::Monoid`.

`printMethods` — Print out methods

Inherited from `Dom::BaseDomain`.

`info` — Print short information about this domain

Inherited from `Dom::BaseDomain`.

`_divide` — Return quotient

Inherited from `Cat::Group`.

`getAxioms` — Return axioms stated in the constructor

Inherited from `Dom::BaseDomain`.

`getCategories` — Return categories stated in the constructor

Inherited from `Dom::BaseDomain`.

`equal` — Test for mathematical equality

Inherited from `Dom::BaseDomain`.

`allAxioms` — Return all axioms

Inherited from `Dom::BaseDomain`.

`undefinedEntries` — Return missing entries

Inherited from `Dom::BaseDomain`.

`allCategories` — Return all categories

Inherited from `Dom::BaseDomain`.

`testtype` — Test type of object

Inherited from `Cat::BaseCategory`.

`allEntries` — Return the names of all entries

Inherited from `Dom::BaseDomain`.

`getSuperDomain` — Return super-domain stated in the constructor

Inherited from `Dom::BaseDomain`.

`subs` — Avoid substitution

Inherited from `Dom::BaseDomain`.

`allSuperDomains` — Return all super-domains

Inherited from `Dom::BaseDomain`.

subsex — Avoid extended substitution

Inherited from `Dom::BaseDomain`.

Dom::DistributedPolynomial

Domains of distributed polynomials

Syntax

Domain Creation

```
Dom::DistributedPolynomial(<Vars, <R, <Order>>>)
```

Element Creation

```
Dom::DistributedPolynomial(Vars, R, Order)(p)
```

```
Dom::DistributedPolynomial(Vars, R, Order)(lm)
```

```
Dom::DistributedPolynomial(Vars, R, Order)(lm, v)
```

Description

`Dom::DistributedPolynomial(Vars, R, ..)` creates the domain of polynomials in the variables of the list `Vars` over the commutative ring `R` in distributed representation.

`Dom::DistributedPolynomial(Vars, R, Order)` creates a domain of polynomials in the variables of the list `Vars` over a domain of category `Cat::CommutativeRing` in sparse distributed representation with respect to the monomial ordering `Order`.

If `Dom::DistributedPolynomial` is called without any argument, a polynomial domain in arbitrarily many indeterminates over the domain `Dom::ExpressionField(normal)` with respect to the lexicographic monomial ordering is created.

If `Dom::DistributedPolynomial` is called only with the variable list `Vars` as argument, the polynomial domain in the variable list `Vars` over the domain `Dom::ExpressionField(normal)` with respect to the lexicographic monomial ordering is created.

Note: Only commutative coefficient rings of type `DOM_DOMAIN` are allowed which inherit from `Dom::BaseDomain`. If `R` is of type `DOM_DOMAIN` but does not inherit from `Dom::BaseDomain`, the domain `Dom::ExpressionField(normal)` will be used instead.

`Dom::DistributedPolynomial` accepts expressions as indeterminates, similar to the kernel domain `DOM_POLY`. Hence, for example, `[x,cos(x)]` is a valid variable list.

If the variable list `Vars` is the empty list (`[]`), a polynomial domain in arbitrarily many indeterminates is created. In this case, when creating new elements from polynomials or polynomial expressions, the system function `indets` is first called to get the variables and then the polynomial is created with respect to these variables. Hence, in this case only identifiers can be valid indeterminates, because `indets` returns only identifiers.

It is not allowed to create polynomial domains in arbitrarily many indeterminates over another polynomial domain of category `Cat::Polynomial`, but it is possible to create multivariate polynomial domains with a given list of variables over any polynomial domain.

`Dom::DistributedPolynomial` represents polynomials over arbitrary commutative rings. It is intended as a basic domain for distributed polynomials from which it is easy to create new distributed polynomial domains.

All usual algebraic and arithmetical polynomial operations are implemented, including Gröbner basis computation.

Note: It is highly recommended to use only coefficient rings with unique zero representation. Otherwise it can happen that, e.g., a polynomial division will not terminate or a wrong degree will be returned.

Please note that for reasons of efficiency not all methods check their arguments, not even at the interactive level. In particular this is true for many access methods, converting methods and technical methods.

Superdomain

`Dom::BaseDomain`

Axioms

If R has `Ax::normalRep`, then `Ax::normalRep`.

If R has `Ax::canonicalRep`, then `Ax::canonicalRep`.

Categories

If Vars has exactly one variable, then `Cat::UnivariatePolynomial(R)`, else `Cat::Polynomial(R)`.

Examples

Example 1

The following call creates a polynomial domain in x, y and z.

```
DP := Dom::DistributedPolynomial([x, y, z])
```

Dom::DistributedPolynomial([x, y, z], Dom::ExpressionField(normal, iszero ∘ normal), LexOrder)

Since neither the coefficient ring nor the monomial ordering was specified, this domain is created with the default values for these parameters.

It is rather easy to create elements of this domain, as e.g.

```
a := DP(x + 2*y*z + 3)
```

$$x + 2\,y\,z + 3$$

```
b := DP(z^4 - 2*y^2*x^2)
```

$$-2\,x^2\,y^2 + z^4$$

In contrast to expressions all elements of this domain have a representation which is fixed by the chosen `Order`, the representation of the coefficient ring `R` and the way of representing monomials.

With these elements one can now perform usual arithmetic operations as, e.g., (scalar) multiplication, multiplication with integers and adding polynomials and ring elements:

```
4*b^2 + a/3 + 1/2
```

$$16\,x^4\,y^4 - 16\,x^2\,y^2\,z^4 + \frac{x}{3} + \frac{2\,y\,z}{3} + 4\,z^8 + \frac{3}{2}$$

There are a lot of methods for manipulating polynomials and to get access to all parts of a polynomial. For example one has access to the leading monomial of a as follows:

```
lmonomial(a)
```

$$x$$

The leading monomial of a polynomial depends on the monomial ordering, so with respect to the degree order one gets a different result:

```
lmonomial(a, DegreeOrder)
```

$$2\,y\,z$$

To get a minus its leading monomial one may call:

```
DP::reductum(a)
```

$$2\,y\,z + 3$$

Obviously the following identity holds:

```
a - lmonomial(a) - DP::reductum(a)
```

$$0$$

There are also methods for converting elements of this domain into other domains, like a basic polynomial domain or the domain of arbitrary expressions:

```
poly(a), domtype(poly(a))
```

$$\text{poly}(x + 2\,y\,z + 3,\ [x,\ y,\ z],\ \text{Dom::ExpressionField}(\text{normal},\ \text{iszero} \circ \text{normal})),\ \text{DOM\_POLY}$$

```
expr(b), domtype(expr(b))
```

$$z^4 - 2\,x^2\,y^2,\ \text{DOM\_EXPR}$$

Parameters

`Vars`

A list of indeterminates. Default is `[]` (the empty list, indicating "arbitrary indeterminates").

`R`

A commutative ring, i.e., a domain of category `Cat::CommutativeRing`. Default is `Dom::ExpressionField(normal)`.

`Order`

A monomial ordering, i.e., one of the predefined orderings `LexOrder`, `DegreeOrder` or `DegInvLexOrder` or any object of type `Dom::MonomOrdering`. Default is `LexOrder`.

`p`

A polynomial or a polynomial expression.

`lm`

List of monomials, which are represented as lists containing the coefficients together with the exponents or exponent vectors.

`v`

List of indeterminates. This parameter is only valid for `Vars = []`.

Entries

"characteristic"	The characteristic of this domain.
"coeffRing"	The coefficient ring of this domain as defined by the parameter `R`.
"key"	The name of the created domain.
"one"	The neutral element w.r.t. `"_mult"`.

"ordering"	The monomial order as defined by the parameter `Order`.
"variables"	The list of variables as defined by the parameter `Vars`.
"zero"	The neutral element w.r.t. `"_plus"`.

Methods

Mathematical Methods

`_divide` — Exact polynomial division

```
_divide(a, b)
```

```
_divide(a, b)
```

```
_divide(a, b)
```

It overloads the function `_divide` for polynomials, i.e., one may use it either in the form `a / b`, or in functional form `_divide(a, b)`.

Note: This method only exists if `R` is an integral domain, i.e., a domain of category `Cat::IntegralDomain`.

`_invert` — Inverse of an element

```
_invert(a)
```

`_mult` — Multiplie polynomials and coefficient ring elements

```
_mult(<a, b, …>)
```

This method overloads the function `_mult` for polynomials, i.e., one may use it either in the form `a * b * ...` or in functional notation `_mult(a, b, ...)`.

`_negate` — Negate a polynomial

```
_negate(a)
```

This method overloads the function `_negate` for polynomials, i.e., one may use it either in the form `-a` or in functional notation `_negate(a)`.

`_plus` — Add polynomials and coefficent ring elements

```
_plus(<a, b, …>)
```

This method overloads the function `_plus` for polynomials, i.e., one may use it either in the form `a + b + ...` or in functional notation `_plus(a, b, ...)`.

`_power` — Nth power of a polynomial

```
_power(a, n)
```

This method overloads the function `_power` for polynomials, i.e., one may use it either in the form `a^n` or in functional notation `_power(a,n)`.

`_subtract` — Subtract a polynomial or a coefficient ring element

```
_subtract(a, b)
```

This method overloads the function `_subtract` for polynomials, i.e., one may use it either in the form `a - b` or in functional notation `_subtract(a, b)`.

`associates` — Test if elements are associates

Inherited from `Cat::IntegralDomain`.

`content` — Content of a polynomial

```
content(a)
```

Note: This method only exists if `R` is a domain of category `Cat::GcdDomain`.

`D` — Differential operator for polynomials

```
D(a)
```

```
D(l, a)
```

`Dpoly` — Differential operator for polynomials

```
Dpoly(a)
```

```
Dpoly(l, a)
```

`Dpoly(l,a)` computes the partial derivative of `a` with respect to `l`. For details see `polylib::Dpoly`.

This method overloads the function `polylib::Dpoly` for polynomials.

`decompose` — Functional decomposition of a polynomial

```
decompose(a, <var>)
```

If `a` is a polynomial in only one variable, the second argument is not necessary.

This method overloads the function `polylib::decompose` for polynomials.

`diff` — Differentiate a polynomial

```
diff(a, varseq)
```

If `varseq` is an empty sequence, `a` is returned unchanged.

If in `varseq` an expression occurs which is not a variable of `a`, the zero polynomial is returned.

This method overloads the function `diff` for polynomials.

`dimension` — Dimension of affine variety

```
dimension(ais, <ord>)
```

```
dimension(ais, <ord>)
```

This method is merely an interface for the function `groebner::dimension`.

Note: This method only exists if `R` is a field, i.e., a domain of category `Cat::Field` and `Vars` is not the empty list.

`divide` — Divide polynomials

```
divide(a, b, <Quo | Rem | Exact>)
```

```
divide(a, b, var, <Quo | Rem | Exact>)
```

If no option is given, the quotient `s` and the remainder `r` are computed such that `a = s*b + r` and the degree of `r` in the relevant indeterminate is smaller than that of `b`. The sequence consisting of `s, r` is returned, otherwise `FAIL`.

If the option `Quo` is given, only the quotient `s` is returned.

If the option `Rem` is given, only the remainder `r` is returned.

If the option `Exact` is given, only the quotient `s` is returned, in case the remainder is zero, otherwise `FAIL`.

`divide(a,b,Exact)` divides the multivariate polynomial `a` by `b`. If `a` cannot be divided by `b`, the method returns `FAIL`.

This method overloads the function `divide` for polynomials.

Note: This method only exists if `R` is a field, i.e., a domain of category `Cat::Field` and either this domain is of category `Cat::UnivariatePolynomial(R)` or `R` has characteristic zero (`R::characteristic = 0`). If the first pair of conditions is true then the first call is valid otherwise the second one.

`divides` — Test if elements divides another

Inherited from `Cat::IntegralDomain`.

`equal` — Test for mathematical equality

Inherited from `Dom::BaseDomain`.

`equiv` — Test for equivalence

Inherited from `Cat::BaseCategory`.

`evalp` — Evaluate a polynomial

```
evalp(a, var = e)
```

This method overloads the function `evalp` for polynomials.

`factor` — Factor a polynomial

```
factor(a)
```

This method overloads the function `factor` for polynomials.

Note: This method only exists if `R` is a domain of category `Cat::Field` or if `R` is the domain `Dom::Integer`.

`func_call` — Applie expressions to a polynomial

```
func_call(a, e1, …, en, <Expr>)
```

```
func_call(a, e1, …, en, <Expr>)
```

```
func_call(a, e1, …, en, <Expr>)
```

`a(e1,...,en)` applies the sequence `e1,...,en` of either elements of this domain or elements of `R` with respect to `Vars` (where `n` is the number of variables) to the polynomial `a`. An element of this domain or an element of the coefficient ring respectively is returned.

`a(e1,...,en, Expr)` applies the sequence of expressions or of elements of this domain or of elements of `R` to the polynomial `a`. With this call `a` is first converted into an expression. Afterwards `e1,...,en` is substituted into this expression with respect to `Vars`. The return value may be any object.

The number of variables must be equal to the number of applied expressions.

Note: This method only exists if `Vars` has at least one indeterminate.

`gcd` — Greatest common divisor of polynomials

```
gcd(a, b, …)
```

This method overloads the function `gcd` for polynomials.

Note: This method only exists if `R` is a domain of category `Cat::GcdDomain`.

---

gcdex — Extended Euclidean algorithm for polynomials

```
gcdex(a, b)
```

This method overloads the function `gcdex` for polynomials. Especially, it only works for coefficient rings described there.

Note: This method only exists if `R` is a domain of category `Cat::GcdDomain`.

---

groebner — Reduced Gröbner basis

```
groebner(ais, <ord>, <Reorder>)
```

```
groebner(ais, <ord>, <Reorder>)
```

If the option `Reorder` is given, the lexicographical order of variables may change to another one that is likely to decrease the running time.

Note: Note that this may also cause a change of the returned list, which may now have polynomials over the same coefficient ring `R` but with a possibly re-ordered variable list. Thus, it may contain elements *not* belonging to this domain.

This method is merely an interface for the function `groebner::gbasis`.

Note: This method only exists if `R` is a field, i.e., a domain of category `Cat::Field`, and `Vars` is not the empty list.

---

idealGenerator — Generator of finitely generated ideal

Inherited from `Cat::EuclideanDomain`.

int — Definite and indefinite integration of a polynomial

```
int(a, <x>)
```

```
int(a, <x = x0 .. x1>)
```

`int(a,x=x0..x1)` returns the definite integral $\int_{x0}^{x1} a \, dx$ or `FAIL`, if the result is not an element of this domain or an element of a polynomial domain over `Dom::Fraction(R)`.

This method overloads the function `int` for polynomials.

`intmult` — Multiplie a polynomial with an integer

```
intmult(a, z)
```

This method is more efficient than using polynomial multiplication and is, e.g., necessary for the method `"Dpoly"`.

`irreducible` — Test if element is irreducible

Inherited from `Cat::FactorialDomain`.

`isUnit` — Test if element is a unit

Inherited from `Cat::Polynomial`.

`isone` — Test for one

```
isone(a)
```

Note: The result can only be valid if the coefficients of `a` are in normal form (i.e., if zero has a unique representation in `R`). Thus, `R` should have at least `Ax::normalRep`.

`iszero` — Test for zero

```
iszero(a)
```

Note: The result can only be valid, if the coefficients of `a` are in normal form (i.e., if zero has a unique representation in `R`). Thus, the coefficient ring `R` should have at least `Ax::normalRep`.

`lcm` — Least common multiple of polynomials

```
lcm(a, b, …)
```

This method overloads the function `lcm` for polynomials.

Note: This method only exists if R is a domain of category Cat::GcdDomain.

makeIntegral — Make the coefficients fraction free

```
makeIntegral(a)
```

Note: This method only exists if R is a domain of category Cat::GcdDomain and R has the method "denom".

monic — Normalize a polynomial

```
monic(a)
```

The zero polynomial returns itself.

Note: This method only exists if R is a field, i.e., a domain of category Cat::Field.

normalForm — Complete reduction modulo an ideal

```
normalForm(a, ais, <ord>)
```

```
normalForm(a, ais, <ord>)
```

This method is merely an interface for the function groebner::normalf.

Note: This method only exists if R is a field, i.e., a domain of category Cat::Field, and Vars is not the empty list.

pdioe — Solve polynomial Diophantine equations

```
pdioe(a, b, c)
```

This method overloads the function solvelib::pdioe.

Note: This method only exists if R is a field, i.e., a domain of category Cat::Field and Vars consists of a single variable.

pdivide — Pseudo-division of polynomials

```
pdivide(a, b, <Quo | Rem>)
```

If the option `Quo` is given, only the pseudo-quotient `q` is returned.

If the option `Rem` is given, only the pseudo-remainder `r` is returned.

This method overloads the function `pdivide` for polynomials.

Note: This method only exists if `Vars` consists of a single variable.

---

pquo — Pseudo-quotient of polynomials

```
pquo(a, b)
```

Note: This method only exists if `Vars` consists of a single variable.

---

prem — Pseudo-remainder of polynomials

```
prem(a, b)
```

Note: This method only exists if `Vars` consists of a single variable.

---

primpart — Return primitive part

Inherited from `Cat::Polynomial`.

quo — Euclidean quotient

Inherited from `Cat::EuclideanDomain`.

random — Create a random polynomial

```
random()
```

With every call the global variable `SEED` is changed by a call of `random()`. Thus it is hard to create the same random sequence twice, see `random`.

If the parameter `Vars` is the empty list, first a list of 1 to 4 variables is generated randomly and the random polynomial is generated in these indeterminates afterwards.

This method overloads the function `polylib::randpoly` for polynomials.

`rem` — Euclidean remainder

Inherited from `Cat::EuclideanDomain`.

`resultant` — Resultant of two polynomials

```
resultant(a, b, <var>)
```

`resultant(a, b, var)` returns the resultant of `a` and `b` with respect to the variable `var`.

The value returned is a polynomial of this domain or `FAIL`.

This method overloads the function `polylib::resultant` for polynomials.

Note: This method only exists if `R` has the method `"_divide"`.

`ringmult` — Multiplie a polynomial with a coefficient ring element

```
ringmult(a, c)
```

`solve` — Zero of polynomials

```
solve(a, <var>, <options>)
```

```
solve(a, <vars>, <options>)
```

```
solve(ais, <var>, <options>)
```

```
solve(ais, <vars>, <options>)
```

`solve(ais, ..)` tries to find the zeros of the polynomial system `ais`. The exact behavior depends on further arguments.

For a detailed description of possible return values and options see function `solve`.

This method overloads the function `solve`.

`SPolynomial` — Compute the S-polynomial of two polynomials

```
SPolynomial(a, b, <ord>)
```

This method is merely an interface for the function `groebner::spoly`.

Note: This method only exists if `R` is a field, i.e., a domain of category `Cat::Field`, and `Vars` is not the empty list.

`sqrfree` — Square-free factorization of polynomials

```
sqrfree(a)
```

The `ai` are primitive and pairwise different square-free divisors of `a` and represented as elements of this domain. `u` is a unit of the coefficient ring and represented as an element of this domain. The `ei` are integers.

This method overloads the function `polylib::sqrfree` for polynomials.

Note: This method only exists if `R` is a field, i.e., a domain of category `Cat::Field`, or if `R` is `Dom::Integer`.

`unitNormal` — Return unit normal

Inherited from `Cat::Polynomial`.

`unitNormalRep` — Return unit normal representation

Inherited from `Cat::Polynomial`.

Access Methods

`coeff` — Coefficient of a polynomial

```
coeff(a)
```

```
coeff(a, var, n)
```

```
coeff(a, n)
```

`coeff(a,var,n)` returns the coefficient of the term `var^n`—as an element of this domain if it is of category `Cat::Polynomial(R)`, or as an element of the coefficient ring `R` if it is of `Cat::UnivariatePolynomial(R)`, where `a` is considered as a univariate polynomial in a valid variable `var`.

`coeff(a,n)` returns the coefficient of the term `var^n`—as an element of this domain if it is of category `Cat::Polynomial(R)`, or as an element of the coefficient ring `R` if it is of `Cat::UnivariatePolynomial(R)`, where `a` is considered as a univariate polynomial in `var` and `var` is the main variable of `a`, i.e., the variable returned by `dom::mainvar(a)`.

This method overloads the function `coeff` for polynomials.

`degree` — Degree of a polynomial

```
degree(a)
```

```
degree(a, var)
```

`degree(a, var)` returns the degree of `a` with respect to `var`.

The degree of the zero polynomial is defined as zero.

This method overloads the function `degree` for polynomials.

`degreevec` — Vector of exponents of the leading term of a polynomial

```
degreevec(a, <ord>)
```

The degree vector of the zero polynomial is defined as a list of zeros.

This method overloads the function `degreevec` for polynomials.

`euclideanDegree` — Euclidean degree function

```
euclideanDegree(a)
```

Note: This method only exists if `Vars` consists of a single variable.

`ground` — Ground term of a polynomial

```
ground(a)
```

This method overloads the function `ground` for polynomials.

`has` — Existence of an object in a polynomial

```
has(a, obj)
```

This method overloads the function `has`.

`indets` — Indeterminate of a polynomial

```
indets(<a>)
```

In case `Vars` is not the empty list, `indets` can be called without argument.

Since this domain allows expressions as indeterminates, the returned set may contain expressions, too.

This method overloads the function `indets` for polynomials.

`lcoeff` — Leading coefficient of a polynomial

```
lcoeff(a)
```

```
lcoeff(a, <vars>, <ord>)
```

`lcoeff(a, ord)` returns the leading coefficient of `a` with respect to the monomial ordering `ord` as an element of the coefficient ring `R`.

`lcoeff(a, vars, ord)` returns the leading coefficient of `a` with respect to the variable list `vars` and the monomial ordering `ord` as an element of this domain if it is of category `Cat::Polynomial(R)`, or as an element of the coefficient ring `R` if it is of `Cat::UnivariatePolynomial(R)`.

- If `ord` is not explicitly given, the lexicographical order `LexOrder` will be used instead.
- It tries to convert `a` into a polynomial in the specified list of indeterminates `vars` over the coefficient ring `R` and returns `FAIL` if this conversions fails.

This method overloads the function `lcoeff` for polynomials.

ldegree — Lowest degree of a polynomial

```
ldegree(a)
```

```
ldegree(a, x)
```

`ldegree(a, x)` returns the lowest degree of the variable `x` in `a`.

This method overloads the function `ldegree` for polynomials.

lmonomial — Leading monomial of a polynomial

```
lmonomial(a, <ord>)
```

```
lmonomial(a, <vars>, <ord>, <Rem>)
```

`lmonomial(a, vars, ord)` returns the leading monomial of `a` with respect to the variable list `vars` and the monomial ordering `ord` as an element of this domain.

- If `ord` is not explicitly given, the lexicographical order `LexOrder` will be used instead.
- It tries to convert `a` into a polynomial in the specified list of indeterminates `vars` over the coefficient ring `R` and returns `FAIL` if this conversions fails.

`lmonomial(a, vars, ord, Rem)` returns the list consisting of the leading monomial and the reductum of `a` with respect to the variable list `vars` and the monomial ordering `ord` as a list of elements of this domain.

- If `ord` is not explicitly given, the lexicographical order `LexOrder` will be used instead.
- It tries to convert `a` into a polynomial in the specified list of indeterminates `vars` over the coefficient ring `R` and returns `FAIL` if this conversions fails.

Note: In MuPAD a monomial denotes a coefficient together with a power product as, e.g., $3\,x^2$.

This method overloads the function `lmonomial` for polynomials.

lterm — Leading term of a polynomial

```
lterm(a)
```

```
lterm(a, <vars>, <ord>)
```

`lterm(a, ord)` returns the leading coefficient of `a` with respect to the monomial ordering `ord` as an element of this domain.

`lterm(a, vars, ord)` returns the leading term of `a` with respect to the variable list `vars` and the monomial ordering `ord` as an element of this domain.

- If `ord` is not explicitly given, the lexicographical order `LexOrder` will be used instead.
- It tries to convert `a` into a polynomial in the specified list of indeterminates `vars` over the coefficient ring `R` and returns `FAIL` if this conversions fails.

Note: In MuPAD a term denotes a power product without a coefficient as, e.g., $x^2 y^3 z$.

This method overloads the function `lterm` for polynomials.

`mainvar` — Main variable of a polynomial

```
mainvar(<a>)
```

If `Vars` is not the empty list, `mainvar` can be called without argument.

`mapcoeffs` — Applie a function to the coefficients of a polynomial

```
mapcoeffs(a, f, <e1, …>)
```

This method overloads the function `mapcoeffs` for polynomials.

`multcoeffs` — Multiplie the coefficients of a polynomial with a factor

```
multcoeffs(a, c)
```

This method overloads the function `multcoeffs` for polynomials.

`nterms` — Number of terms of a polynomial

```
nterms(a)
```

This method overloads the function `nterms` for polynomials.

`nthcoeff` — N-th coefficient of a polynomial

```
nthcoeff(a, n, <ord>)
```

If n is larger than the number of monomials of the polynomial then the function returns FAIL.

The zero polynomial has no monomials. nthcoeff returns FAIL when invoked on the zero polynomial.

This method overloads the function nthcoeff for polynomials.

nthmonomial — N-th monomial of a polynomial

```
nthmonomial(a, n, <ord>)
```

If n is larger than the number of monomials of the polynomial then the function returns FAIL.

The zero polynomial has no monomials. nthmonomial returns FAIL for the zero polynomial.

This method overloads the function nthmonomial for polynomials.

nthterm — N-th term of a polynomial

```
nthterm(a, n, <ord>)
```

If n is larger than the number of monomials of the polynomial then the function returns FAIL.

The zero polynomial has no monomials. nthterm returns FAIL when called with the zero polynomial.

This method overloads the function nthterm for polynomials.

orderedVariableList — Ordered list of indeterminates of a polynomial

```
orderedVariableList(<a>)
```

In case Vars is not the empty list, orderedVariableList can be called without an argument.

pivotSize — Size of a pivot element

```
pivotSize(a)
```

This method is called if this domain is used as the component ring of a matrix domain to perform Gaussian elimination.

`reductum` — Reductum of a polynomial

```
reductum(a, <ord>)
```

`subs` — Avoid substitution

Inherited from `Dom::BaseDomain`.

`subsex` — Avoid extended substitution

Inherited from `Dom::BaseDomain`.

`tcoeff` — Lowest coefficient of a polynomial

```
tcoeff(a, <ord>)
```

This method overloads the function `tcoeff` for polynomials.

Conversion Methods

`coerce` — Coerce into this domain

Inherited from `Cat::BaseCategory`.

`convert` — Conversion to a polynomial

```
convert(p)
```

`convert_to` — Convert element

Inherited from `Dom::BaseDomain`.

`expr` — Conversion to a basic type

```
expr(a)
```

This method overloads the function `expr`.

poly — Convert to a basic polynomial domain

```
poly(a)
```

This method overloads the function `poly`.

TeX — TeX formatting of a polynomial

```
TeX(a)
```

TeXCoeff — TeX formatting of a polynomial coefficient

```
TeXCoeff(c)
```

TeXident — TeX formatting of a polynomial indeterminate

```
TeXident(var)
```

TeXTerm — TeX formatting of a polynomial term

```
TeXTerm(t)
```

Technical Methods

adaptIndets — Convert polynomials to common indeterminates

```
adaptIndets(<a, b, …>)
```

Note: This method only exists if the parameter `Vars` is the empty list (`[]`).

---

allAxioms — Return all axioms

Inherited from `Dom::BaseDomain`.

allCategories — Return all categories

Inherited from `Dom::BaseDomain`.

allEntries — Return the names of all entries

Inherited from `Dom::BaseDomain`.

`allSuperDomains` — Return all super-domains

Inherited from `Dom::BaseDomain`.

`getAxioms` — Return axioms stated in the constructor

Inherited from `Dom::BaseDomain`.

`getCategories` — Return categories stated in the constructor

Inherited from `Dom::BaseDomain`.

`getSuperDomain` — Return super-domain stated in the constructor

Inherited from `Dom::BaseDomain`.

`hasProp` — Test for a certain property

Inherited from `Dom::BaseDomain`.

`info` — Print short information about this domain

Inherited from `Dom::BaseDomain`.

`isNeg` — Test on leading output token

```
isNeg(a)
```

`mult` — Multiplie polynomials

```
mult(a, b, …)
```

`new` — Create a new element

```
new(p)
```

```
new(lm)
```

```
new(lm, v)
```

`dom(p)` creates an element of this domain from a polynomial or a polynomial expression `p` and returns that element. If this is not possible, an error message is given.

If `Vars` is chosen as the empty list (`[]`) then in creating new elements from a polynomial or polynomial expression the function `indets` is first called to get the identifiers.

Afterwards the element is created with this list of identifiers. For creating an element from a constant the dummy variable `_dummy` is introduced. The drawback of this approach is that two mathematically equal polynomials may have variable lists which differ by the dummy variable.

`dom(lm)` creates, if `Vars` is not the empty list `[]`, a polynomial from the list `lm` of the form `[[c1, [e11,... e1n]],... [cm,[em1,... emn]]]` where the `ci` are coefficients and the `eij` are the exponents with respect to `Vars`. For a univariate polynomial this list can be simplified to `[[c1,e1],... [cm,em]]`.

`dom(lm,v)` creates, if `Vars = []`, a polynomial from the list `lm` of the form `[[c1, [e11,... e1n]],... [cm,[em1,... emn]]]` where the `ci` are coefficients and the `eij` are the exponents with respect to `v`. For a univariate polynomial this list can be simplified to `[[c1,e1],... [cm,em]]`. The list of indeterminates `v` must contain valid indeterminates.

`plus` — Add polynomials

```
plus(a, b, …)
```

`print` — Print polynomials

```
print(a)
```

This method overloads the function `print`.

`printMethods` — Print out methods

Inherited from `Dom::BaseDomain`.

`printMonomial` — Print a monomial in defined order

```
printMonomial(c, d, v)
```

`printTerm` — Print a term in defined order

```
printTerm(d)
```

```
printTerm(d, v)
```

`printTerm(d,v)` returns an ordered sequence of the indeterminates together with their powers as given in the variable list `v` and the degree vector `d` respectively.

Note that this call is only valid if `nops(v)=nops(d)`.

`Rep` — Data representation of a polynomial

```
Rep(a)
```

`sign` — Leading sign of a polynomial

```
sign(a)
```

Note: this method does not have the meaning of a mathematical sign function!

`testtype` — Test type of object

Inherited from `Cat::BaseCategory`.

`undefinedEntries` — Return missing entries

Inherited from `Dom::BaseDomain`.

`whichEntry` — Return the domain or category implementing an entry

Inherited from `Dom::BaseDomain`.

See Also

MuPAD Domains

`Dom::MultivariatePolynomial` | `Dom::Polynomial` | `Dom::UnivariatePolynomial`

Dom::Expression

Domain of all objects of basic type

Syntax

```
Dom::Expression(x)
```

Description

`Dom::Expression` comprises all objects only consisting of operands of built-in types.

`Dom::Expression` is a façade domain: it has no domain elements, but uses system representation.

Unlike `Dom::ExpressionField`, `Dom::Expression` does not belong to any arithmetical category, and its elements need not be arithmetical expressions.

`Dom::Expression` mainly serves as a super-domain to `Dom::ArithmeticalExpression`; it rarely makes sense to use it directly.

Superdomain

`Dom::BaseDomain`

Axioms

`Ax::systemRep`, `Ax::efficientOperation("_divide")`, `Ax::efficientOperation("_mult")`, `Ax::efficientOperation("_invert")`

Categories

`Cat::BaseCategory`

Examples

Example 1

Almost every MuPAD object can be converted to an expression. Objects of basic type *are* expressions.

```
Dom::Expression([3, array(1..2), rectform(exp(I))])
```

$$[3,\ (\ \mathrm{NIL}\ \ \mathrm{NIL}\ ),\ \cos(1)+\sin(1)\,i]$$

The `convert` method flattens its argument: hence expression sequences are *not* allowed.

```
Dom::Expression((3, x))
```

```
Error: The number of arguments is incorrect. [expr]
  Evaluating: Dom::Expression::new
```

Parameters

x

An object of basic type consisting only of operands of built-in types, or any other object convertible to such using `expr`.

Entries

"randomIdent"	an identifier used for creating random elements

Methods

Conversion Methods

`convert` — Conversion of objects

```
convert(x)
```

`convert_to` — Conversion to other domains

```
convert_to(x, T)
```

`expr` — Just return the argument

```
expr(x)
```

`testtype` — Test whether its argument is an expression

```
testtype(x, Dom::Expression)
```

This method overloads `testtype`; since `Dom::Expression` has no domain elements, the overloading can only be caused by the second argument.

`float` — Convert numbers to floats

```
float(x)
```

Technical Methods

`subs` — Substitution

```
subs(x, s, …)
```

`subsex` — Extended substitution

```
subsex(x, s, …)
```

`random` — Create random expression

```
random()
```

See Also

MuPAD Domains
`Dom::ExpressionField`

Dom::ExpressionField

Domains of expressions forming a field

Syntax

Domain Creation

```
Dom::ExpressionField(<Normal, <IsZero>>)
```

Element Creation

```
Dom::ExpressionField(Normal, IsZero)(e)
```

Description

Domain Creation

`Dom::ExpressionField(Normal, IsZero)` creates a domain of expressions forming a field, where the functions `Normal` and `IsZero` are used to normalize expressions and test for zero.

The function `Normal` is used to normalize the expressions representing the elements, the function `IsZero` is used to test the expressions for zero. It is assumed that the field has characteristic 0.

The domain cannot decide if the element expressions—given the normalizing function and zero test—actually form a field. It is up to the user to choose correct functions for normalizing and zero test and to enter only valid expressions as domains elements.

One should view this domain constructor as a pragmatic way to create a field of characteristic 0 in an ad-hoc fashion. Note that the default of using `id` and `iszero` does not yield a field really, but it is often convenient and sensible to use the resulting structure as a field.

`Normal` must be a function which takes an expression representing a domain element and returns the normalized expression. `Normal` should return `FAIL` if the expression is not valid.

If `Normal` is not given, then the system function `id` is used, i.e., only the kernel simplifier is used to normalize expressions.

If a normalizing function other than `id` is given, it is assumed that this functions returns a normal form where the zero element is uniquely represented by the constant `0`.

`IsZero` must be a function which takes an expression representing a domain element and returns `TRUE` if the expression represents zero and `FALSE` otherwise.

If `IsZero` is not given, then `iszero @ Normal` is used for zero testing. If `Normal` is equal to `id` this functional expression is simplified to `iszero`.

If `Normal` is equal to `id` and `IsZero` is equal to `iszero`, a façade domain is created, i.e., the domain elements are simply expressions and are not explicitly created by `new`.

Otherwise the elements of the domain are explicitly created by `new`. Each such element has one operand, which is the expression representing the domain element. The element expressions are normalized after each operation using the function `Normal`.

Element Creation

`Dom::ExpressionField(Normal, IsZero)(e)` creates a field element represented by the expression `e`. The expression is normalized using the function `Normal`.

If `Normal` returns `FAIL`, it is assumed that the expression does not represent a valid field element. If this test is not fully implemented the domain cannot decide if the expression represents a valid field element. In this case it is up to the user to enter only valid expressions as field elements.

If `Normal` is equal to `id` and `IsZero` is equal to `iszero`, the domain is only a façade domain. In this case the expression `e` is returned after being simplified by the built-in kernel simplifier.

Superdomain

If `Normal = id` and `IsZero = iszero`, then `Dom::ArithmeticalExpression`, else `Dom::BaseDomain`.

Axioms

`Ax::indetElements`

If `Normal = id` and `IsZero = iszero`, then `Ax::efficientOperation ("_divide"), Ax::efficientOperation ("_mult"), Ax::efficientOperation("_invert")`, else `Ax::normalRep`.

If `Normal = id` and `IsZero = iszero` and `IsZero = iszero`, then `Ax::systemRep`.

Categories

`Cat::Field`, `Cat::DifferentialRing`

Examples

Example 1

`Dom::ExpressionField(normal)` creates a field of rational expressions over the rationals. The expressions representing the field elements are allways normalized by `normal`:

```
Fn := Dom::ExpressionField(normal):
a := Fn((x^2 - 1)/(x - 1))
```

$$x + 1$$

The field elements are explicit elements of the domain:

```
domtype(a)
```

Dom::ExpressionField(normal, iszero ∘ normal)

Example 2

In the domain `Dom::ExpressionField(id, iszero@normal)` the expressions representing the elements are normalized by the kernel simplifier only:

```
Fi := Dom::ExpressionField(id, iszero@normal):
a := Fi((x^2 - 1)/(x - 1))
```

$$\frac{x^2-1}{x-1}$$

The elements of this domain are not normalized (when viewed as rational expressions over the rationals), thus the domain does not have the axiom `Ax::normalRep`:

```
b := a/Fi(x + 1) - Fi(1)
```

$$\frac{x^2-1}{(x-1)\,(x+1)}-1$$

But nevertheless this domain also represents the field of rational expressions over the rationals, because zero is detected correctly by the function `iszero @ normal`:

```
iszero(b)
```

TRUE

Parameters

`Normal`

A function used to normalize the expressions of the domain; default is `id`.

`IsZero`

A function used to test the expressions of the domain for zero; default is `iszero @ Normal`.

`e`

An expression representing a field element.

Entries

"characteristic"	The characteristic of the fields created by this constructor is assumed to be `0`.
"one"	The element represented by the expression `1` is assumed to be a neutral element w.r.t. `"_mult"`.
"zero"	The element represented by the expression `0` is assumed to be a neutral element w.r.t. `"_plus"`.

Methods

Mathematical Methods

`abs` — Absolute value

```
abs(x)
```

Overloads the function `abs`, thus may be called via `abs(x)`.

`combine` — Combine terms of the same algebraic structure

```
combine(x, <a>)
```

Overloads the function `combine`, thus may be called via `combine(x,...)`.

`conjugate` — Complex conjugate

```
conjugate(x)
```

Overloads the function `conjugate`, thus may be called via `conjugate(x)`.

`D` — Differential operator

```
D(<l>, x)
```

Overloads the function `D`, thus may be called via `D(x)` or `D(l, x)`.

`denom` — Denominator

```
denom(x)
```

Overloads the function `denom`, thus may be called via `denom(x)`.

`diff` — Differentiate an element

```
diff(x, <v, , …>)
```

Overloads the function `diff`, thus may be called via `diff(x,...)`.

`_divide` — Divide elements

```
_divide(x, y)
```

Overloads the function `_divide`, thus may be called via `x/y` or `_divide(x, y)`.

`equal` — Test for mathematical equality

```
equal(x, y)
```

`expand` — Expand an element

```
expand(x)
```

Overloads the function `expand`, thus may be called via `expand(x)`.

`factor` — Factorize an element

```
factor(x)
```

Overloads the function `factor`, thus may be called via `factor(x)`.

`float` — Floating-point approximation

```
float(x)
```

Overloads the function `float`, thus may be called via `float(x)`.

`gcd` — Greatest common divisor

```
gcd(x, …)
```

Overloads the function `gcd`, thus may be called via `gcd(x,...)`.

`Im` — Imaginary part of an element

```
Im(x)
```

Overloads the function `Im`, thus may be called via `Im(x)`.

`int` — Definite and indefinite integration

```
int(x, <v>)
```

Overloads the function `int`, thus may be called via `int(x,...)`.

`intmult` — Integer mutiple

```
intmult(x, n)
```

`_invert` — Invert an element

```
_invert(x)
```

Overloads the function `_invert`, thus may be called via `1/x` or `_invert(x)`.

`iszero` — Test for zero

```
iszero(x)
```

Overloads the function `iszero`, thus may be called via `iszero(x)`.

`lcm` — Least common multiple

```
lcm(x, …)
```

Overloads the function `lcm`, thus may be called via `lcm(x, ...)`.

`_leequal` — Test if less or equal

```
_leequal(x, y)
```

Please note that the function `_leequal` can only test numbers (in a syntactical sense), but not constant expressions like PI or `sqrt(2)`.

Overloads the function `_leequal`, thus may be called via `x <= y`, `y >= x` or `_leequal(x, y)`.

`_less` — Test if element is less

```
_less(x, y)
```

Please note that the function `_less` can only test numbers (in a syntactical sense), but not constant expressions like PI or `sqrt(2)`.

Overloads the function `_less`, thus may be called via `x < y`, `y > x` or `_less(x, y)`.

`limit` — Limit computation

```
limit(x, <v, …>)
```

Overloads the function `limit`, thus may be called via `limit(x,...)`.

`max` — Maximum of arguments

```
max(x, …)
```

Overloads the function `max`, thus may be called via `max(x,...)`.

`min` — Minimum of arguments

```
min(x, …)
```

Overloads the function `min`, thus may be called via `min(x,...)`.

`_mult` — Multiplie elements

```
_mult(x, …)
```

If all arguments are of this domain or can be coerced to this domain (using the method `coerce`), the product of the expressions representing the arguments is calculated using the function `_mult`.

If one of the arguments cannot be coerced, the arguments up to the offending one are multiplied and then the method `"_mult"` of the domain of the offending argument is called to multiply the remaining arguments.

Overloads the function `_mult`, thus may be called via `x*...` or `_mult(x,...)`.

`_negate` — Negate an element

```
_negate(x)
```

Overloads the function `_negate`, thus may be called via `-x` or `_negate(x)`.

`norm` — Norm of an element

```
norm(x)
```

Overloads the function `norm`, thus may be called via `norm(x)`.

Please note that the system function `norm`, applied to an expression, computes the norm of that expression interpreted as a polynomial expression and *not* the absolute value of the expression. This may be regarded as an inconsistency.

`normal` — Normal form

```
normal(x)
```

Overloads the function `normal`, thus may be called via `normal(x)`.

`numer` — Numerator

```
numer(x)
```

Overloads the function `numer`, thus may be called via `numer(x)`.

`_plus` — Add elements

```
_plus(x, …)
```

If all arguments are of this domain or can be coerced to this domain (using the method `coerce`) the sum of the expressions representing the arguments is calculated using the function `_plus`.

If one of the arguments cannot be coerced the arguments up to the offending one are added and then the method `"_plus"` of the domain of the offending argument is called to add the remaining arguments.

Overloads the function `_plus`, thus may be called via `x+...` or `_plus(x,...)`.

`_power` — Exponentiate arguments

```
_power(x, y)
```

```
_power(x, y)
```

If both arguments are of this domain the power is calculated by mapping the function `_power` to the expressions representing the arguments.

If one of the arguments is not of this domain it is coerced to this domain, then the power is computed. If the coercion fails an error is raised.

Note that it is assumed that at least one of the arguments is of this domain.

Overloads the function `_power`, thus may be called via `x^y` or `_power(x, y)`.

`radsimp` — Simplifie radicals

```
radsimp(x)
```

Overloads the function `radsimp`, thus may be called via `radsimp(x)`.

`random` — Create a random element

```
random()
```

See `polylib::randpoly` for details about creating random polynomials.

`Re` — Real part of an element

```
Re(x)
```

Overloads the function `Re`, thus may be called via `Re(x)`.

`sign` — Sign of an element

```
sign(x)
```

Overloads the function `sign`, thus may be called via `sign(x)`.

`simplify` — General simplification of an element

```
simplify(x, <a>)
```

Overloads the function `simplify`, thus may be called via `simplify(x,...)`.

`solve` — Solve an equation

```
solve(x, <a, …>)
```

Note that this method will never return an element of this domain. See `solve` for details about results and optional additional arguments.

Overloads the function `solve`, thus may be called via `solve(x,...)`.

`sqrfree` — Square-free factorization

```
sqrfree(x)
```

Overloads the function `polylib::sqrfree`, thus may be called via `polylib::sqrfree(x)`.

`_subtract` — Subtract elements

```
_subtract(x, y)
```

Overloads the function `_subtract`, thus may be called via `x-y` or `_subtract(x, y)`.

Conversion Methods

`convert` — Convert to this domain

```
convert(x)
```

`convert_to` — Convert to other domain

```
convert_to(x, T)
```

`expr` — Convert to basic type

```
expr(x)
```

This method is called by the function `expr` if a subexpression of the argument is an element of this domain.

`new` — Creating an element

```
new(x)
```

Overloads the function call operator for this domain, thus may be called via `F(x)` where `F` is this domain.

Access Methods

nops — Number of operands

```
nops(x)
```

Overloads the function `nops`, thus may be called via `nops(x)`.

op — Get operands

```
op(x)
```

```
op(x, i)
```

Returns the operand with index `i` of the expression representing `x`. If `i` is `0` then the operator of the expression is returned, which usually is not an element of this domain. The other operands are converted to elements of this domain.

This method is called by the function `op` when an element of this domain is contained, as a subexpression, in the first argument of `op`. Operand ranges and paths are handled by `op` and need not be handled by this method. See `op` for details.

subs — Substitute subexpressions

```
subs(x, e, …)
```

Maps `subs` to the expression representing `x`. The resulting expression is converted to an element of this domain.

This method is called by the function `subs` when an element of this domain is contained, as a subexpression, in the first argument of `subs`. See `subs` for details.

subsex — Extended substitution

```
subsex(x, e, , …)
```

Maps `subsex` to the expression representing `x`. The resulting expression is converted to an element of this domain.

This method is called by the function `subsex` when an element of this domain is contained, as a subexpression, in the first argument of `subsex`. See `subsex` for details.

`subsop` — Substitute operand

```
subsop(x, e, , …)
```

This method is called by the function `subsop` when an element of this domain is contained, as a subexpression, in the first argument of `subsop`. Operand ranges and paths are handled by `subsop` and need not be handled by this method. See `subsop` for details.

Technical Methods

`indets` — Identifier of an element

```
indets(x, <optionName>)
```

Overloads the function `indets`, thus may be called via `indets(x)` and `indets(x, optionName)`, respectively.

`length` — Size of an element

```
length(x)
```

Overloads the function `length`, thus may be called via `length(x)`.

`map` — Applie function to operands

```
map(x, f, <a, …>)
```

Overloads the function `map`, thus may be called via `map(x, f,...)`.

`rationalize` — Approximate floating-point numbers by rationals

```
rationalize(x, <a, …>)
```

Note that this method does *not* overload the function `rationalize` from the standard library package, but the function `numeric::rationalize` from the `numeric` package instead. Thus the method may be called via `numeric::rationalize(x,...)`.

`pivotSize` — Pivot size

```
pivotSize(x)
```

Dom::Float

Real floating-point numbers

Syntax

```
Dom::Float(x)
```

Description

`Dom::Float` is the set of real floating-point numbers represented by elements of the domain `DOM_FLOAT`.

`Dom::Float` is the domain of real floating point numbers represented by expressions of type `DOM_FLOAT`.

`Dom::Float` has category `Cat::Field` out of pragmatism. This domain actually is not a field because `bool(1.0 = float(3) / float(3))` returns `FALSE` for example.

Elements of `Dom::Float` are usually not created explicitly. However, if one creates elements using the usual syntax, it is checked whether the input expression may be converted to a floating point number. This means `Dom::Float` is a facade domain which creates elements of domain type `DOM_FLOAT`.

Viewed as a differential ring `Dom::Float` is trivial, it contains constants only.

`Dom::Float` has the domain `Dom::Numerical` as its super domain, i.e., it inherits each method which is defined by `Dom::Numerical` and not implemented by `Dom::Float`. Methods described below are re-implemented by `Dom::Float`.

Superdomain

`Dom::Numerical`

Axioms

`Ax::canonicalRep`, `Ax::systemRep`, `Ax::canonicalOrder`, `Ax::efficientOperation("_divide")`, `Ax::efficientOperation("_mult")`, `Ax::efficientOperation("_invert")`

Categories

`Cat::DifferentialRing`, `Cat::Field`, `Cat::OrderedSet`

Examples

Example 1

Creating some floating-point numbers using `Dom::Float`. This example also shows that `Dom::Float` is a facade domain.

```
Dom::Float(2.3); domtype(%)
```

```
    2.3

    DOM_FLOAT
```

```
Dom::Float(sin(2/3*PI) + 3)
```

```
    3.866025404
```

```
Dom::Float(sin(x))
```

```
Error: The arguments are invalid. [Dom::Float::new]
```

Example 2

By tracing the method `Dom::Float::testtypeDom` we can see the interaction between `testtype` and `Dom::Float::testtypeDom`.

```
prog::trace(Dom::Float::testtypeDom):
delete x:
testtype(x, Dom::Float);
testtype(3.4, Dom::Float);
prog::untrace(Dom::Float::testtypeDom):

enter Dom::Float::testtypeDom(x, Dom::Float)
computed FAIL

    FALSE


enter Dom::Float::testtypeDom(3.4, Dom::Float)
computed TRUE

    TRUE
```

Parameters

x

An expression which can be converted to a `DOM_FLOAT` by the function `float`.

Entries

"one"	the unit element; it equals `1.0`.
"zero"	The zero element; it equals `0.0`.

Methods

Mathematical Methods

`pivotSize` — Size of a pivot element

`pivotSize(x)`

This method is called if this domain is used as the component ring of a matrix domain to perform Gaussian elimination.

`random` — Random number generation

```
random()
```

Conversion Methods

`convert` — Conversion of objects

```
convert(x)
```

In general, if `float(x)` evaluates to a real floating-point number of type `DOM_FLOAT`, this number is the result of the conversion.

`convert_to` — Conversion to other domains

```
convert_to(x, T)
```

The following domains are allowed for `T`: `DOM_FLOAT`, `Dom::Float` and `Dom::Numerical`.

`testtype` — Type checking

```
testtype(x, T)
```

In general this method is called from the function `testtype` and not directly by the user. “Example 2” on page 7-171 demonstrates this behavior.

See Also

MuPAD Domains

`Dom::Complex` | `Dom::Integer` | `Dom::Numerical` | `Dom::Rational` | `Dom::Real`

Dom::FloatIV

The “field” of Floating Point Intervals

Syntax

```
Dom::FloatIV(x, …)
```

Description

`Dom::FloatIV` is the inclusion algebra of (finite unions of) rectangular intervals in the complex plane.

`Dom::FloatIV` is the domain of kernel intervals of type `DOM_INTERVAL`.

`Dom::FloatIV` has category `Cat::Field` out of pragmatism. This domain actually is not a field because, for example, there is no additive inverse of `1...2`.

Elements of `Dom::FloatIV` are usually not created explicitly. The syntax given above is equivalent to an `interval` call, with no check to ensure that the result is in fact an interval, it could, for example, also be an expression with all numerical coefficients replaced by intervals. Apart from this behavior of the constructor and the `"convert"` slot, `Dom::FloatIV` is a façade domain for elements of domain type `DOM_INTERVAL`.

Viewed as a differential ring `Dom::FloatIV` is trivial, it contains constants only.

`Dom::FloatIV` has the domain `Dom::Numerical` as its super domain, i.e., it inherits each method which is defined by `Dom::Numerical` and not re-implemented by `Dom::FloatIV`. Methods described below are those implemented by `Dom::FloatIV`.

Superdomain

`Dom::Numerical`

Axioms

`Ax::canonicalRep`, `Ax::systemRep`, `Ax::efficientOperation("_divide")`, `Ax::efficientOperation("_invert")`, `Ax::efficientOperation("_mult")`, `Ax::efficientOperation("_plus")`

Categories

`Cat::Field`, `Cat::DifferentialRing`

Parameters

`x, …`

MuPAD expressions

Methods

Mathematical Methods

`Im` — Imaginary Part

`Im(iv)`

`Re` — Real Part

`Re(iv)`

`abs` — Absolute Value

`abs(iv)`

`arccos` — Inverse Cosine

`arccos(iv)`

arccosh — Inverse Hyperbolic Cosine

```
arccosh(iv)
```

arccot — Inverse Cotangent

```
arccot(iv)
```

arccoth — Inverse Hyperbolic Cotangent

```
arccoth(iv)
```

arccsc — Inverse Cosecant

```
arccsc(iv)
```

arccsch — Inverse Hyperbolic Cosecant

```
arccsch(iv)
```

arcsec — Inverse Secant

```
arcsec(iv)
```

arcsech — Inverse Hyperbolic Secant

```
arcsech(iv)
```

arcsin — Inverse Sine

```
arcsin(iv)
```

arcsinh — Inverse Hyperbolic Sine

```
arcsinh(iv)
```

arctan — Inverse Tangent

```
arctan(iv)
```

arctanh — Inverse Hyperbolic Tangent

```
arctanh(iv)
```

arg — Argument ('Polar Angle')

```
arg(iv)
```

beta — Beta Function

```
beta(iv)
```

ceil — Rounding Up

```
ceil(iv)
```

center — Geometric Center

```
center(iv)
```

cos — Cosine

```
cos(iv)
```

cosh — Hyperbolic Cosine

```
cosh(iv)
```

cot — Cotangent

```
cot(iv)
```

coth — Hyperbolic Cotangent

```
coth(iv)
```

csc — Cosecant

```
csc(iv)
```

dirac — Dirac delta distribution

```
dirac(iv)
```

exp — Exponential Function

```
exp(iv)
```

`floor` — Rounding Down

```
floor(iv)
```

`gamma` — Gamma Function

```
gamma(iv)
```

`ln` — Logarithm

```
ln(iv)
```

`mag` — Interval Magnitude

```
mag(iv)
```

`mig` — Interval Mignitude

```
mig(iv)
```

`random` — Random Element

```
random()
```

`round` — Round

```
round(iv)
```

`sec` — Secans

```
sec(iv)
```

`sign` — Sign

```
sign(iv)
```

`sin` — Sine

```
sin(iv)
```

`sinh` — Hyperbolic Sine

```
sinh(iv)
```

`sqrt` — Square Root

```
sqrt(iv)
```

`tan` — Tangent

```
tan(iv)
```

`tanh` — Hyperbolic Tangent

```
tanh(iv)
```

`trunc` — Round to Zero

```
trunc(iv)
```

`width` — Width of an Interval

```
width(x)
```

Conversion Methods

`convert` — Conversion of Objects

```
convert(x)
```

`testtype` — Type checking

```
testtype(x, T)
```

Usually, this method is called from the function `testtype` and not directly by the user.

See Also

MuPAD Domains
`Dom::Complex` | `Dom::Float` | `Dom::Integer` | `Dom::Interval` | `Dom::Numerical` | `Dom::Rational` | `Dom::Real`

Dom::Fraction

Field of fractions of an integral domain

Syntax

Domain Creation

```
Dom::Fraction(R)
```

Element Creation

```
Dom::Fraction(R)(r)
```

Description

Domain Creation

`Dom::Fraction(R)` creates a domain which represents the field of fractions $F = \left\{ \frac{x}{y} \middle| x \in \mathbb{R}, y \in \mathbb{R} \setminus \{0\} \right\}$ of the integral domain R.

An element of the domain `Dom::Fraction(R)` has two operands, the numerator and denominator.

If `Dom::Fraction(R)` has the axiom `Ax::canonicalRep` (see below), the denominators have unit normal form and the gcds of numerators and denominators cancel.

The domain `Dom::Fraction(Dom::Integer)` represents the field of rational numbers. But the created domain is not the domain `Dom::Rational`, because it uses a different representation of its elements. Arithmetic in `Dom::Rational` is much more efficient than it is in `Dom::Fraction(Dom::Integer)`.

Element Creation

If `r` is a rational expression, then an element of the field of fractions `Dom::Fraction(R)` is created by going through the operands of `r` and converting each operand into an

element of `R`. The result of this process is `r` in the form $\frac{x}{y}$, where x and y are elements of `R`. If `R` has `Cat::GcdDomain`, then x and y are coprime.

If one of the operands can not be converted into the domain `R`, an error message is issued.

Superdomain

`Dom::BaseDomain`

Axioms

`Ax::normalRep`

Categories

`Cat::QuotientField(R)`

Examples

Example 1

We define the field of rational functions over the rationals:

```
F := Dom::Fraction(Dom::Polynomial(Dom::Rational))
```

$$\text{Dom::Fraction(Dom::Polynomial(Dom::Rational, LexOrder))}$$

and create an element of `F`:

```
a := F(y/(x - 1) + 1/(x + 1))
```

$$\frac{x + y + x\,y - 1}{x^2 - 1}$$

To calculate with such elements use the standard arithmetical operators:

```
2*a, 1/a, a*a
```

$$\frac{2x+2y+2xy-2}{x^2-1}, \frac{x^2-1}{x+y+xy-1}, \frac{x^2y^2+2x^2y+x^2+2xy^2-2x+y^2-2y+1}{x^4-2x^2+1}$$

Some system functions are overloaded for elements of domains generated by `Dom::Fraction`, such as `diff`, `numer` or `denom` (see the description of the corresponding methods `"diff"`, `"numer"` and `"denom"` above).

For example, to differentiate the fraction `a` with respect to `x` enter:

```
diff(a, x)
```

$$-\frac{y-2x+2xy+x^2y+x^2+1}{x^4-2x^2+1}$$

If one knows the variables in advance, then using the domain `Dom::DistributedPolynomial` yields a more efficient arithmetic of rational functions:

```
Fxy := Dom::Fraction(
  Dom::DistributedPolynomial([x, y], Dom::Rational)
)
```

Dom::Fraction(Dom::DistributedPolynomial([x, y], Dom::Rational, LexOrder))

```
b := Fxy(y/(x - 1) + 1/(x + 1)):
b^3
```

$$(x^3y^3+3x^3y^2+3x^3y+x^3+3x^2y^3+3x^2y^2-3x^2y-3x^2+3xy^3-3xy^2-3xy+3x+y^3-3y^2+3y-1)/(x^6-3x^4+3x^2-1)$$

Example 2

We create the field of rational numbers as the field of fractions of the integers, i.e., $\mathbb{Q} = \left\{ \frac{x}{y} \;\middle|\; x \in \mathbb{Z}, y \in \mathbb{Z} \setminus \{0\} \right\}$:

```
Q := Dom::Fraction(Dom::Integer):
Q(1/3)
```

$$\frac{1}{3}$$

```
domtype(%)
```

Dom::Fraction(Dom::Integer)

Another representation of $\mathbb{Q}$ in MuPAD is the domain `Dom::Rational` where the rationals are of the kernel domains `DOM_INT` and `DOM_RAT`. Therefore it is much more efficient to work with `Dom::Rational` than with `Dom::Fraction(Dom::Integer)`.

Parameters

`R`

An integral domain, i.e., a domain of category `Cat::IntegralDomain`

`r`

A rational expression, or an element of `R`

Entries

"characteristic"	is the characteristic of `R`.
"coeffRing"	is the integral domain `R`.
"one"	is the one of the field of fractions of `R`, i.e., the fraction 1.
"zero"	is the zero of the field of fractions of `R`, i.e., the fraction 0.

Methods

Mathematical Methods

_divide — Divide two fractions

```
_divide(x, y)
```

This method overloads the function `_divide` for fractions, i.e., one may use it in the form `x / y` or in functional notation: `_divide(x, y)`.

_invert — Invert a fraction

```
_invert(r)
```

This method overloads the function `_invert` for fractions, i.e., one may use it in the form `1/r` or `r^(-1)`, or in functional notation: `_invert(r)`.

_less — Less-than relation

```
_less(q, r)
```

An implementation is provided only if `R` is an ordered set, i.e., a domain of category `Cat::OrderedSet`.

This method overloads the function `_less` for fractions, i.e., one may use it in the form `q < r`, or in functional notation: `_less(q, r)`.

_mult — Multiplie fractions by fractions or rational expressions

```
_mult(q, r)
```

If `q` is not of the domain type `Dom::Fraction(R)`, it is considered as a rational expression which is converted into a fraction over `R` and multiplied with `q`. If the conversion fails, `FAIL` is returned.

The same applies to `r`.

This method also handles more than two arguments. In this case, the argument list is splitted into two parts of the same length which both are multiplied with the function `_mult`. The two results are multiplied again with `_mult` whose result then is returned.

This method overloads the function `_mult` for fractions, i.e., one may use it in the form `q * r` or in functional notation: `_mult(q, r)`.

`_negate` — Negate a fraction

```
_negate(r)
```

This method overloads the function `_negate` for fractions, i.e., one may use it in the form `-r` or in functional notation: `_negate(r)`.

`_power` — Integer power of a fraction

```
_power(r, n)
```

This method overloads the function `_power` for fractions, i.e., one may use it in the form `r^n` or in functional notation: `_power(r, n)`.

`_plus` — Add fractions

```
_plus(q, r, …)
```

If one of the arguments is not of the domain type `Dom::Fraction(R)`, then `FAIL` is returned.

This method overloads the function `_plus` for fractions, i.e., one may use it in the form `q + r` or in functional notation: `_plus(q, r)`.

`D` — Differential operator

```
D(r)
```

An implementation is provided only if `R` is a partial differential ring, i.e., a domain of category `Cat::PartialDifferentialRing`.

This method overloads the operator `D` for fractions, i.e., one may use it in the form `D(r)`.

`denom` — Denominator of a fraction

```
denom(r)
```

This method overloads the function `denom` for fractions, i.e., one may use it in the form `denom(r)`.

`diff` — Differentiation of fractions

```
diff(r, u)
```

This method overloads the function `diff` for fractions, i.e., one may use it in the form `diff(r, u)`.

An implementation is provided only if `R` is a partial differential ring, i.e., a domain of category `Cat::PartialDifferentialRing`.

`equal` — Test on equality of fractions

```
equal(q, r)
```

`factor` — Factorize the numerator and denominator of a fraction

```
factor(r)
```

The factors u, r_1, ..., r_n are fractions of type `Dom::Fraction(R)`, the exponents e_1, ..., e_n are integers.

The system function `factor` is used to perform the factorization of the numerator and denominator of `r`.

This method overloads the function `factor` for fractions, i.e., one may use it in the form `factor(r)`.

`intmult` — Integer multiple of a fraction

```
intmult(r, n)
```

`iszero` — Test for zero

```
iszero(r)
```

An element of the field `Dom::Fraction(R)` is zero if its numerator is the zero element of `R`. Note that there may be more than one representation of the zero element if `R` does not have `Ax::canonicalRep`.

This method overloads the function `iszero` for fractions, i.e., one may use it in the form `iszero(r)`.

`numer` — Numerator of a fraction

```
numer(r)
```

This method overloads the function `numer` for fractions, i.e., one may use it in the form `numer(r)`.

`random` — Random fraction generation

```
random()
```

The returning fraction is normalized (see the methods `"normalize"` and `"normalizePrime"`.

Conversion Methods

`convert_to` — Fraction conversion

```
convert_to(r, T)
```

If the conversion fails, `FAIL` is returned.

The conversion succeeds if `T` is one of the following domains: `Dom::Expression` or `Dom::ArithmeticalExpression`.

Use the function `expr` to convert `r` into an object of a kernel domain (see below).

`expr` — Convert a fraction into an object of a kernel domain

```
expr(r)
```

The result is an object of a kernel domain (e.g., `DOM_RAT` or `DOM_EXPR`).

This method overloads the function `expr` for fractions, i.e., one may use it in the form `expr(r)`.

`TeX` — TeX formatting of a fraction

```
TeX(r)
```

The method `TeX` of the component ring `R` is used to get the TeX-representations of the numerator and denominator of `r`, respectively.

`retract` — Retraction to base domain

```
retract(r)
```

Technical Methods

`normalize` — Normalizing fractions

```
normalize(x, y)
```

Normalization means to remove the gcd of `x` and `y`. Hence, `R` needs to be of category `Cat::GcdDomain`. Otherwise, normalization cannot be performed and the result of this method is the fraction $\frac{x}{y}$.

`normalizePrime` — Normalizing fractions over integral domains with a gcd

```
normalizePrime(x, y)
```

In rings of category `Cat::GcdDomain`, elements are assumed to be relatively prime. Hence, there is no need to normalize the fraction $\frac{x}{y}$.

In rings not of category `Cat::GcdDomain`, normalization of elements can not be performed and the result of this method is the fraction $\frac{x}{y}$.

See Also

MuPAD Domains
`Dom::Rational`

Dom::GaloisField

Finite fields

Syntax

Domain Creation

```
Dom::GaloisField(q)
Dom::GaloisField(p, n)
Dom::GaloisField(p, n, f)
Dom::GaloisField(F, n)
Dom::GaloisField(F, n, f)
```

Element Creation

```
Dom::GaloisField(p, n, f)(g)
```

Description

Domain Creation

`Dom::GaloisField(p, n, f)` creates the residue class field Z_p[X]/<f>, a finite field with p^n elements. If `f` is not given, it is chosen at random among all irreducible polynomials of degree n.

`Dom::GaloisField(q)` (where $q = p^n$) is equivalent to `Dom::GaloisField(p,n)`.

`Dom::GaloisField(F, n, f)` creates the residue class field *F[X]/<f>*, a finite field with $|F|^n$ elements.

If `f` is not given, a random irreducible polynomial of appropriate degree is used; some free identifier is chosen as its variable, and this one must also be used when creating domain elements.

Although $n = 1$ is allowed, `Dom::IntegerMod` should be used for representing prime fields.

If `F` is of type `Dom::GaloisField`, consisting of residue classes of polynomials, the variable of these polynomials must be distinct from the variable of `f`. If a tower several of Galois fields is constructed, the variable used in the uppermost Galois field must not equal any of those used in the tower. A special entry `"VariablesInUse"` serves to keep track of all variables appearing somewhere in the tower.

Element Creation

`Dom::GaloisField(p,n,f)(g)` (or, respectively, `Dom::GaloisField(F,n,f)(g)`) creates the residue class of `g` modulo `f`. It is represented by the unique polynomial in that class that has smaller degree than `f`.

Superdomain

`Dom::AlgebraicExtension(Dom::IntegerMod(p), f)`

Axioms

`Ax::canonicalRep`

Categories

`Cat::Field`, `Cat::Algebra(F)`, `Cat::VectorSpace(F)`

Examples

Example 1

We define `L` to be the field with 4 elements. Then $a^4 = a$ for every $a \in L$, by a well-known theorem.

```
L:=Dom::GaloisField(2, 2, u^2+u+1): L(u+1)^4
```

$$u+1$$

Parameters

q

Prime power

p

Prime

n

Positive integer

f

Univariate irreducible polynomial over `Dom::IntegerMod(p)` or `F`, or polynomial expression convertible to such

F

Finite field of type `Dom::IntegerMod` or `Dom::GaloisField`.

g

Univariate polynomial over the ground field in the same variable as `f`, or polynomial expression convertible to such

Entries

"zero"	the zero element of the field
"one"	the unit element of the field
"characteristic"	the characteristic of the field

"size"	the number of elements of the field
"PrimeField"	the prime field, which equals `Dom::IntegerMod(p)`.
"Variable"	the variable of the polynomial `f`.
"VariablesInUse"	a list consisting of `"Variable"` and the variables used by the ground field.

Methods

Mathematical Methods

`iszero` — Test for zero

```
iszero(a)
```

It overloads the function `iszero`.

`_power` — Integer power of an element

```
_power(a, n)
```

It overloads `_power`.

`frobenius` — Frobeniu map

```
frobenius(a)
```

`conjugates` — Conjugate of an element

```
conjugates(a)
```

`order` — Order of an element

```
order(a)
```

`isSquare` — Test whether an element is a square

```
isSquare(a)
```

`ln` — Discrete logarithm

```
ln(a, b)
```

`elementNumber` — Enumerate field elements

```
elementNumber(a)
```

The inverse of this mapping has not been implemented.

`companionMatrix` — Companion matrix of the Galois field over its ground field

```
companionMatrix()
```

`companionPowers` — Power of the companion matrix

```
companionPowers()
```

`matrixRepresentation` — Isomorphism to the algebra generated by the companion matrix

```
matrixRepresentation(a)
```

If A is the companion matrix, the image of $\sum_i a_i X^i$ is $\sum_i a_i A^i$.

`randomPrimitive` — Choose a primitive element at random

```
randomPrimitive()
```

`isBasis` — Test elements for being a basis over the ground field

```
isBasis(l)
```

`isNormal` — Test whether a given field element is normal

```
isNormal(a)
```

`randomNormal` — Choose normal element at random

```
randomNormal()
```

`isPrimitivePolynomial` — Test whether a polynomial over the field is primitive

```
isPrimitivePolynomial(h)
```

Conversion Methods

`convert` — Conversion from other types

```
convert(a)
```

`convert_to` — Conversion to other types

```
convert_to(a, T)
```

See Also

MuPAD Domains
`Dom::AlgebraicExtension` | `Dom::IntegerMod`

Dom::ImageSet

Domain of images of sets under mappings

Syntax

Domain Creation

```
Dom::ImageSet()
```

Element Creation

```
Dom::ImageSet(f, x, S)
```

```
Dom::ImageSet(f, [x1, …], [S1, …])
```

Description

Domain Creation

`Dom::ImageSet` is the domain of all sets of complex numbers that can be written as the set of all values taken on by some mapping, i.e., sets of the form $\{f(x_1, \ldots, x_n) \mid x_i \in S_i\}$ for some function f and some sets S_1, …, S_n.

Image sets are mainly used by `solve` to express sets like $\{k\,\pi \mid k \in \mathbb{Z}\}$.

`Dom::ImageSet` belongs to the category `Cat::Set`—arithmetical and set-theoretic operations are inherited from there.

Element Creation

`Dom::ImageSet(f, x, S)` represents the set of all values that can be obtained by substituting some element of `S` for `x` in the expression `f`.

`Dom::ImageSet(f, [x1, ...], [S1, ...])` represents the set of all values that can be obtained by substituting, for each i, the identifier x_i by some element of S_i in the expression `f`.

`Dom::ImageSet(f, x, S)` represents the set $\{f \mid x \in S\}$. `Dom::ImageSet(f, [x1, ..., xn], [S1, ..., Sn])` represents the set $\{f \mid x_i \in S_i\}$, $i = 1 \ldots n$.

`f` need not contain `x`; on the other hand, it may contain other identifiers (free variables).

If a list of several identifiers is given, the identifiers must be distinct.

`S` must be a set; see `solve` for an overview of the different kinds of sets in MuPAD.

`Dom::ImageSet` carries out some automatical simplifications that may produce a result of a type different from `Dom::ImageSet`.

`Dom::ImageSet` renames the variables `x1, ..., xn`, in order to avoid naming conflicts as well as producing a nicer output.

Superdomain

`Dom::BaseDomain`

Categories

`Cat::Set`

Examples

Example 1

We define S to be the set of all integer multiples of π.

```
S:= Dom::ImageSet(ugly*PI, ugly, Z_)
```

$$\{\pi\, k \mid k \in \mathbb{Z}\}$$

Our ugly variable name has been replaced by a nicer one which suggests that it represents an integer.

We may now apply the usual set-theoretic operations.

```
S intersect Dom::Interval(3..7)
```

$$\{\pi, 2\,\pi\}$$

Example 2

An element of an image set may be obtained by substituting all parameters by some values:

```
S:= Dom::ImageSet(a^7 + b^3 + C, [a, b], [Z_, Z_])
```

$$\{k^7 + l^3 + C \mid k \in \mathbb{Z}, l \in \mathbb{Z}\}$$

On calling the `evalParam` method, we have to take care that the variable names have been replaced.

```
Dom::ImageSet::evalParam(S, k = 3, l = 5)
```

$$C + 2312$$

The same may be achieved using the index operator:

```
S[3, 5]
```

$$C + 2312$$

Substituting only for one parameter, we obtain an image set in the other parameter:

```
Dom::ImageSet::evalParam(S, k = 3)
```

$$\{l^3 + C + 2187 \mid l \in \mathbb{Z}\}$$

A parameter may be substituted by itself, meaning that it becomes a free variable:

```
Dom::ImageSet::evalParam(S, k = k)
```

$$\{k^7 + l^3 + C \mid l \in \mathbb{Z}\}$$

The `evalParam` method cannot be used to substitute a free variable:

```
Dom::ImageSet::evalParam(S, C = 3)
```

$$\{k^7 + l^3 + C \mid k \in \mathbb{Z}, l \in \mathbb{Z}\}$$

```
delete S:
```

Parameters

`f`

Arithmetical expression

`x`

Identifier or indexed identifier

`S`

Set of any type

Methods

Mathematical Methods

`changevar` — Change the name of a variable

```
changevar(A, oldvar, newvar)
```

The new variable `newvar` must not equal any element of the list of variables; this is not checked!

`setvar` — Set the name of the variable

```
setvar(A, newvar)
```

```
setvar(A, newvar)
```

For an argument A that is not an image set, the method "setvar" is applied to all image sets contained in the expression A. A might be, for example, a union, intersection, etc. of image sets and other sets.

homogpointwise — Define an n-ary pointwise operator for image sets

```
homogpointwise(Op)
```

Op must accept arithmetical expressions as arguments.

isEmpty — Test whether a set is empty

```
isEmpty(A)
```

substituteBySet — Substitute an ImageSet for a variable

```
substituteBySet(a, x, A)
```

freeIndets — Free parameters of a set

```
freeIndets(A)
```

If $A = \{f(x_1, \ldots, x_n, y_1, \ldots y_k) \mid x_i \in S_i\}$, the x_i are called bound and the y_i are called free parameters.

Use the slot "variables" to obtain the bound parameters.

evalParam — Insert values for bound parameters

```
evalParam(A(x = value, …))
```

If x is not a parameter, but a free variable of A, it is not substituted by value.

value may be an identifier or contain identifiers; in particular, it may contain x and/or some of the remaining parameters. This may be used to convert parameters into free variables.

Several parameters may be replaced in a single call.

See “Example 2” on page 7-197.

`_index` — Extract element by inserting values for bound parameters

```
_index(A, value1, …)
```

The number of values passed must match the number of variables of `A`.

It is not checked whether for each i, the value for the ith parameter belongs to the ith set.

See "Example 2" on page 7-197.

Access Methods

`expr` — Defining mapping as an expression

```
expr(A)
```

This method overloads the function `expr`.

`variables` — List of variables

```
variables(A)
```

The free parameters (identifiers appearing in f other than the x_i) can be obtained using the `"freeIndets"` slot.

`nvars` — Number of variables

```
nvars(A)
```

`sets` — List of sets

```
sets(A)
```

Technical Methods

`print` — Print image set

```
print(A)
```

Dom::Integer

Ring of integer numbers

Syntax

```
Dom::Integer(x)
```

Description

`Dom::Integer` is the ring of integer numbers represented by elements of the domain `DOM_INT`.

Elements of `Dom::Integer` are usually not created explicitly. However, if one creates elements using the usual syntax, it is checked whether the input is an integer number. This means that `Dom::Integer` is a façade domain which creates elements of domain type `DOM_INT`.

Viewed as a differential ring `Dom::Integer` is trivial, it contains constants only.

`Dom::Integer` has the domain `Dom::Numerical` as its super domain, i.e., it inherits each method which is defined by `Dom::Numerical` and not re-implemented by `Dom::Integer`. Methods described below are those implemented by `Dom::Integer`.

Superdomain

`Dom::Numerical`

Axioms

`Ax::canonicalRep`, `Ax::systemRep`, `Ax::canonicalOrder`, `Ax::canonicalUnitNormal`, `Ax::closedUnitNormals`, `Ax::efficientOperation("_divide")`, `Ax::efficientOperation("_mult")`

Categories

`Cat::EuclideanDomain`, `Cat::FactorialDomain`, `Cat::DifferentialRing`, `Cat::OrderedSet`

Examples

Example 1

Creating some integer numbers using `Dom::Integer`. This example also shows that `Dom::Integer` is a façade domain.

```
Dom::Integer(2); domtype(%)
```

```
    2

    DOM_INT
```

```
Dom::Integer(2/3)
```

```
Error: The arguments are invalid. [Dom::Integer::new]
```

Example 2

By tracing the method `Dom::Integer::testtypeDom` we can see the interaction between `testtype` and `Dom::Integer::testtypeDom`.

```
prog::trace(Dom::Integer::testtypeDom):
delete x:
testtype(x, Dom::Integer);
testtype(3, Dom::Integer);
prog::untrace(Dom::Integer::testtypeDom):
```

```
enter Dom::Integer::testtypeDom(x, Dom::Integer)
computed FALSE

    FALSE
```

```
enter Dom::Integer::testtypeDom(3, Dom::Integer)
computed TRUE

    TRUE
```

Parameters

x

An integer

Methods

Mathematical Methods

`associates` — Associate elements

```
associates(x, y)
```

`_divide` — Division of two objects

```
_divide(x, y)
```

`_divides` — Decide if a number divides another one

```
_divides(x, y)
```

`euclideanDegree` — Euclidean degree

```
euclideanDegree(x)
```

`factor` — Factorization

```
factor(x)
```

gcd — Gcd computation

```
gcd(x1, x2, …)
```

gcdex — Applie the extended Euclidean algorithm

```
gcdex(x, y)
```

_invert — Inverse of an element

```
_invert(x)
```

irreducible — Prime number test

```
irreducible(x)
```

isUnit — Test if an element is a unit

```
isUnit(x)
```

lcm — Compute the lcm

```
lcm(x1, x2, …)
```

quo — Compute the euclidean quotient

```
quo(x, y)
```

random — Random number generation

```
random()
```

```
random(n)
```

```
random(m .. n)
```

This methods returns a random number between 0 and n - 1.

This methods returns a random number between `m` and `n`.

rem — Compute the Euclidean reminder

```
rem(x, y)
```

`unitNormal` — Unit normal part

```
unitNormal(x)
```

`unitNormalRep` — Unit normal representation

```
unitNormalRep(x)
```

Conversion Methods

`convert` — Conversion of objects

```
convert(x)
```

`convert_to` — Conversion to other domains

```
convert_to(x, T)
```

The following domains are allowed for `T`: `DOM_INT`, `Dom::Integer`, `Dom::Rational`, `DOM_FLOAT`, `Dom::Float` and `Dom::Numerical`.

`testtype` — Type checking

```
testtype(x, T)
```

Usually, this method is called from the function `testtype` and not directly by the user. “Example 2” on page 7-202 demonstrates this behavior.

See Also

MuPAD Domains
`Dom::Complex` | `Dom::Float` | `Dom::Numerical` | `Dom::Rational` | `Dom::Real`

Dom::IntegerMod

Residue class rings modulo integers

Syntax

Domain Creation

```
Dom::IntegerMod(n)
```

Element Creation

```
Dom::IntegerMod(n)(a)
```

Description

Domain Creation

`Dom::IntegerMod(n)` creates the residue class ring of integers modulo `n`.

`Dom::IntegerMod(n)` creates the integer residue class rings $\mathbb{Z}/n\mathbb{Z}$.

Element Creation

`Dom::IntegerMod(n)(a)` creates the residue class of `a` modulo `n`.

Superdomain

`Dom::BaseDomain`

Axioms

`Ax::normalRep`, `Ax::canonicalRep`, `Ax::noZeroDivisors`, `Ax::closedUnitNormals`, `Ax::canonicalUnitNormal`,

`Ax::efficientOperation("_invert")`, `Ax::efficientOperation("_divide")`, `Ax::efficientOperation("_mult")`

Categories

If n is prime, then `Cat::Field`, else `Cat::CommutativeRing`.

Examples

Example 1

We define the residue class ring of the integers `mod 7`:

```
Z7:= Dom::IntegerMod(7)
```

```
Dom::IntegerMod(7)
```

Next, we create some elements:

```
a:= Z7(1); b:= Z7(2); c:= Z7(3)
```

```
1 mod 7
```

```
2 mod 7
```

```
3 mod 7
```

We may use infix notation for arithmetical operations since the operators have been overloaded:

```
a + b, a*b*c, 1/c, b/c/a/c
```

```
3 mod 7, 6 mod 7, 5 mod 7, 1 mod 7
```

`a` and `b` are squares while `c` is not:

```
Z7::isSquare(a), Z7::isSquare(b), Z7::isSquare(c)
```

```
TRUE, TRUE, FALSE
```

Indeed, `c` is a generator of the group of units:

```
Z7::order(a), Z7::order(b), Z7::order(c)
```

```
1, 3, 6
```

Parameters

`n`

Positive integer greater than 1

`a`

Any integer or a rational number whose denominator is coprime to `n`

Entries

"characteristic"	the characteristic of the residue class ring, `n`
"one"	the unit element, `1 mod n`
"zero"	the zero element, `0 mod n`

Methods

Mathematical Methods

`_divide` — Division of two elements

```
_divide(element1, element2)
```

`_invert` — Invert elements

```
_invert(element)
```

`_mult` — Multiply elements

```
_mult(element, …)
```

`_negate` — Negate elements

```
_negate(element)
```

`_plus` — Add elements

```
_plus(element, …)
```

`_power` — Power of elements

```
_power(element, power)
```

`_subtract` — Subtraction of two elements

```
_subtract(element1, element2)
```

`D` — Return derivative

Inherited from `Cat::CommutativeRing`.

`associates` — Test for associate elements

Inherited from `Cat::Field`.

`coerce` — Coerce into this domain

Inherited from `Cat::BaseCategory`.

`diff` — Differentiate element

Inherited from `Cat::CommutativeRing`.

`divide` — Division with remainder

Inherited from `Cat::Field`.

`divides` — Test if division is exact

Inherited from `Cat::Field`.

`equal` — Test for mathematical equality

Inherited from `Dom::BaseDomain`.

`equiv` — Test for equivalence

Inherited from `Cat::BaseCategory`.

`euclideanDegree` — Return Euclidean degree

Inherited from `Cat::Field`.

`factor` — Unique factorization

Inherited from `Cat::Field`.

`gcd` — Greatest common divisor

Inherited from `Cat::Field`.

`gcdex` — Extended greatest common divisor

Inherited from `Cat::EuclideanDomain`.

`idealGenerator` — Generator of finitely generated ideal

Inherited from `Cat::EuclideanDomain`.

`irreducible` — Test if element is irreducible

Inherited from `Cat::Field`.

`isUnit` — Test if element is an unit

Inherited from `Cat::Field`.

`isone` — Test if element is one

Inherited from `Cat::Monoid`.

`lcm` — Least common multiple

Inherited from `Cat::GcdDomain`.

quo — Return Euclidean quotient

Inherited from `Cat::Field`.

rem — Return Euclidean remainder

Inherited from `Cat::Field`.

sqrfree — Square-free factorization

Inherited from `Cat::Field`.

testtype — Test type of object

Inherited from `Cat::BaseCategory`.

isSquare — Test for being a square

```
isSquare(element)
```

iszero — Zero test

```
iszero(element)
```

ln — Discrete logarithm

```
ln(element, base)
```

The result is `infinity` if `element` is not in the subgroup generated by `base`.

The result is `FAIL` if `base` is not a unit.

order — Order

```
order(element)
```

The result is `FAIL` if `element` is not a unit.

Access Methods

subs — Avoid substitution

Inherited from `Dom::BaseDomain`.

subsex — Avoid extended substitution

Inherited from `Dom::BaseDomain`.

Conversion Methods

TeX — TeX output

```
TeX(element)
```

convert — Conversion

```
convert(number)
```

The conversion fails if the denominator of `number` and the modulus `n` are not relatively prime.

convert_to — Conversion

```
convert_to(element, d)
```

expr — Convert an element to an expression

```
expr(element)
```

Technical Methods

allAxioms — Return all axioms

Inherited from `Dom::BaseDomain`.

allCategories — Return all categories

Inherited from `Dom::BaseDomain`.

allEntries — Return the names of all entries

Inherited from `Dom::BaseDomain`.

allSuperDomains — Return all super-domains

Inherited from `Dom::BaseDomain`.

`getAxioms` — Return axioms stated in the constructor

Inherited from `Dom::BaseDomain`.

`getCategories` — Return categories stated in the constructor

Inherited from `Dom::BaseDomain`.

`getSuperDomain` — Return super-domain stated in the constructor

Inherited from `Dom::BaseDomain`.

`hasProp` — Test for a certain property

Inherited from `Dom::BaseDomain`.

`info` — Print short information about this domain

Inherited from `Dom::BaseDomain`.

`new` — Create element of this domain

Inherited from `Cat::BaseCategory`.

`print` — Printing elements

```
print(element)
```

`printMethods` — Print out methods

Inherited from `Dom::BaseDomain`.

`random` — Random element

```
random()
```

`undefinedEntries` — Return missing entries

Inherited from `Dom::BaseDomain`.

`unitNormal` — Unit normal form

Inherited from `Cat::Field`.

`unitNormalRep` — Unit normal representation

Inherited from `Cat::Field`.

`whichEntry` — Return the domain or category implementing an entry

Inherited from `Dom::BaseDomain`.

See Also

MuPAD Domains
`Dom::GaloisField` | `Dom::Integer`

Dom::Interval

Intervals of real numbers

Syntax

```
Dom::Interval(l, r)

Dom::Interval([l], r)

Dom::Interval(l, [r])

Dom::Interval([l], [r])

Dom::Interval([l, r])
```

Description

`Dom::Interval` represents the set of all intervals of real numbers.

`Dom::Interval(l, r)` creates the interval of all real numbers between `l` and `r`. If a border is given as a list with `l` or `r` as the sole element, this border will be regarded as a closed border, otherwise the interval does not contain `l` and `r`.

A border can be any arithmetical expression that could represent a real number, e.g., `sqrt(2*x)` and `a + I`. Properties are ignored.

The domain `Dom::Interval` provides fundamental operations to combine intervals with intervals and other mathematical objects.

The return value can be either an interval of type `Dom::Interval` or the empty set of type `DOM_SET`, if the interval is empty.

Most mathematical operations are overloaded to work with intervals (such as `sin`). If f is a function of n real variables, its extension to intervals is defined to be $f(J_1, \ldots, J_n) = \{f(j_1, \ldots, j_n) \mid j_i \in J_i\}$. The return value of such an operation is in most cases an interval, a union of intervals, a `Dom::ImageSet` or a set. For example,

the sine of an interval `[a, b]` is the interval `{sin(x), x in [a, b]}` that contains all sine values of the given interval. In general, you should expect the return value to be an interval larger than strictly necessary. Also note that, when using the same interval twice in one formula, the uses are regarded as independent, so `interval1/interval1` does not return the interval `[1, 1]` as you might expect.

The functions overloaded in this way are:

- `_mult`, `_divide`, `_invert`, `_power`
- `_plus`, `_negate`, `_subtract`
- `abs`
- `cos`, `arccos`, `cosh`, `arccosh`, `cot`, `arccot`, `coth`, `arccoth`, `csc`, `arccsc`, `csch`, `arccsch`, `sec`, `arcsec`, `sech`, `arcsech`, `sin`, `arcsin`, `sinh`, `arcsinh`, `tan`, `arctan`, `tanh`, `arctanh`
- `dirac`, `heaviside`
- `exp`, `ln`
- `sign`

Furthermore, an interval is a special type of set. This is reflected by `Dom::Interval` having the category `Cat::Set`. Among the methods inherited from `Cat::Set`, the following are especially important: `intersect`, `minus` and `union`.

An interval can be open or closed. If one border is given as a list with one element `[x]`, then this element `x` is taken as border and the interval will be created as closed at this side. If the interval should be closed at both sides, one list with the both borders as arguments can be given.

Superdomain

`Dom::BaseDomain`

Categories

`Cat::Set`, `Cat::AbelianMonoid`

Examples

Example 1

First create a closed interval between 0 and 1.

```
A:= Dom::Interval([0], [1])
```

$$[0, 1]$$

Now another open interval between -1 and 1.

```
B:= Dom::Interval(-1, 1)
```

$$(-1, 1)$$

Intervals can be handled like other objects.

```
A + B, A - B, A*B, A/B
```

$$(-1, 2), (-1, 2), (-1, 1), \mathbb{R}$$

```
2*A, 1 - A, (A - 1)^2
```

$$[0, 2], [0, 1], [0, 1]$$

Example 2

Standard functions are overloaded to work with intervals.

```
sin(B), float(sin(B))
```

$$(-\sin(1), \sin(1)), (-0.8414709848, 0.8414709848)$$

Example 3

The next examples shows some technical methods to access and manipulate intervals.

Get the borders and open/closed information about intervals.

```
A:= Dom::Interval([0], [1]):
Dom::Interval::left(A), Dom::Interval::leftB(A)
```

$$0, [0]$$

```
Dom::Interval::isleftopen(A), Dom::Interval::subsleft(A, -1)
```

$$\mathrm{FALSE}, [-1, 1]$$

Parameters

`l`

The left border. If given as a list of one element (the left border), the interval will be created as left closed.

`r`

The right border. If given as a list of one element (the right border), the interval will be created as right closed.

Entries

"one"	the unit element; it equals the one-point interval [1, 1].
"zero"	the zero element; it equals the one-point interval [0, 0].

Methods

Mathematical Methods

`Im` — Imaginary part of an interval (this always equals zero)

```
Im(interval)
```

`Re` — Real part of an interval (this is the interval)

```
Re(interval)
```

`_divide` — Divide intervals

```
_divide(interval1, interval2)
```

`_intersect` — Intersection of sets

Inherited from `Cat::Set`.

`_invert` — Invert intervals

```
_invert(interval)
```

`_minus` — Set of subtractions

Inherited from `Cat::Set`.

`_mult` — Set of product of set elements

Inherited from `Cat::Set`.

`_negate` — Negate intervals

```
_negate(interval)
```

`_plus` — Set of sums of set elements

Inherited from `Cat::Set`.

`_power` — Pointwise power

Inherited from `Cat::Set`.

`_union` — Union of sets

Inherited from `Cat::Set`.

`abs` — Absolute value of intervals

```
abs(interval)
```

arccos — Inverse cosine of intervals

```
arccos(interval)
```

arccosh — Area cosine of intervals

```
arccosh(interval)
```

arccot — Inverse cotangent of intervals

```
arccot(interval)
```

arccoth — Area cotangent of intervals

```
arccoth(interval)
```

arcsin — Inverse sine of intervals

```
arcsin(interval)
```

arcsinh — Area sine of intervals

```
arcsinh(interval)
```

arctan — Inverse tangent of intervals

```
arctan(interval)
```

arctanh — Area tangent of intervals

```
arctanh(interval)
```

coerce — Coerce into this domain

Inherited from `Cat::BaseCategory`.

contains — Containing an element

```
contains(interval, element)
```

cos — Cosinu of intervals

```
cos(interval)
```

`cosh` — Hyperbolic cosinus of intervals

```
cosh(interval)
```

`cot` — Cotangent of intervals

```
cot(interval)
```

`coth` — Hyperbolic cotangent of intervals

```
coth(interval)
```

`dirac` — Dirac distribution of an interval

```
dirac(interval)
```

`equiv` — Test for equivalence

Inherited from `Cat::BaseCategory`.

`exp` — Exponential function of an interval

```
exp(interval)
```

`heaviside` — Heaviside function

```
heaviside(interval)
```

`intmult` — Return integer multiple

Inherited from `Cat::AbelianMonoid`.

`ln` — Natural logarithm of an interval

```
ln(interval)
```

`max` — Maximum of an interval

```
max(interval, …)
```

The maximum of intervals is the set of all possible results of the function `max` when applied to a sequence of arguments consisting of exactly one element of each interval.

min — Minimum of an interval

```
min(interval, …)
```

The minimum of intervals is defined analogously to their maximum.

new — Create an interval

```
new(left, right)
```

```
new([left], right)
```

```
new(left, [right])
```

```
new([left], [right])
```

sign — Signum of an interval

```
sign(interval)
```

sin — Sine of intervals

```
sin(interval)
```

sinh — Hyperbolic sine of intervals

```
sinh(interval)
```

tan — Tangent of intervals

```
tan(interval)
```

tanh — Hyperbolic tangent of intervals

```
tanh(interval)
```

Access Methods

borders — Border of an interval

```
borders(interval)
```

`left` — Left border of an interval

```
left(interval)
```

`leftB` — Left border of an interval

```
leftB(interval)
```

`isleftopen` — Left open interval

```
isleftopen(interval)
```

`isrightopen` — Right open interval

```
isrightopen(interval)
```

`iszero` — Null interval

```
iszero(interval)
```

`op` — Operand (borders) of an interval

```
op(interval)
```

`subs` — Substitution in intervals

```
subs(Interval, equation, …)
```

`subsex` — Avoid extended substitution

Inherited from `Dom::BaseDomain`.

`subsleft` — Substitute left border

```
subsleft(interval, left)
```

`subsright` — Substitute right border

```
subsright(interval, right)
```

`subsvals` — Substitute both borders

```
subsvals(interval, left, right)
```

Conversion Methods

`convert` — Converting objects to intervals

```
convert(object)
```

If the conversion fails, `FAIL` is returned.

`convert_to` — Convert element

Inherited from `Dom::BaseDomain`.

`float` — Convert to floating-point interval

```
float(interval)
```

`getElement` — One element of an interval

```
getElement(interval)
```

`simplify` — Simplify intervals

```
simplify(interval)
```

`testtype` — Test type of object

Inherited from `Cat::BaseCategory`.

`TeX` — Generate TeX output

Inherited from `Dom::BaseDomain`.

Technical Methods

`allAxioms` — Return all axioms

Inherited from `Dom::BaseDomain`.

`allCategories` — Return all categories

Inherited from `Dom::BaseDomain`.

`allEntries` — Return the names of all entries

Inherited from `Dom::BaseDomain`.

`allSuperDomains` — Return all super-domains

Inherited from `Dom::BaseDomain`.

`emptycheck` — Check intervals

```
emptycheck(interval)
```

`equal` — Comparison of intervals

```
equal(interval, interval)
```

`getAxioms` — Return axioms stated in the constructor

Inherited from `Dom::BaseDomain`.

`getCategories` — Return categories stated in the constructor

Inherited from `Dom::BaseDomain`.

`getSuperDomain` — Return super-domain stated in the constructor

Inherited from `Dom::BaseDomain`.

`hasProp` — Test for a certain property

Inherited from `Dom::BaseDomain`.

`info` — Print short information about this domain

Inherited from `Dom::BaseDomain`.

`map` — Apply functions to intervals

```
map(interval, function, <argument, …>)
```

`mapBorders` — Apply functions to the borders of an interval

```
mapBorders(interval, function, <argument, …>)
```

`print` — Printing intervals

```
print(interval)
```

`printMethods` — Print out methods

Inherited from `Dom::BaseDomain`.

`random` — Random interval

```
random()
```

`undefinedEntries` — Return missing entries

Inherited from `Dom::BaseDomain`.

`whichEntry` — Return the domain or category implementing an entry

Inherited from `Dom::BaseDomain`.

`zip` — Combine intervals

```
zip(interval, interval, function)
```

Algorithms

The operand of an object of `Dom::Interval` is an object of the domain `property::IVnat`, which realizes the basic interval arithmetic. This domain is not documented.

See Also

MuPAD Functions
`Type::Interval`

Dom::LinearOrdinaryDifferentialOperator

Domain of linear ordinary differential operators

Syntax

Domain Creation

```
Dom::LinearOrdinaryDifferentialOperator(<Var, <DVar, <Ring>>>)
```

Element Creation

```
Dom::LinearOrdinaryDifferentialOperator(Var, DVar, Ring)(p)
```

```
Dom::LinearOrdinaryDifferentialOperator(Var, DVar, Ring)(l)
```

```
Dom::LinearOrdinaryDifferentialOperator(Var, DVar, Ring)(eq, yx)
```

Description

`Dom::LinearOrdinaryDifferentialOperator(Var, DVar, Ring)` creates the domain of linear ordinary differential operators with coefficients in the differential ring `Ring` and with derivation `Var` where `DVar` is the differential indeterminate. Elements of this domain are also called Ore polynomials and the multiplication of two elements is completely determined by the prescribed rule $\mathrm{Var}\ r = r\ \mathrm{Var} + \frac{\partial}{\partial \mathrm{DVar}}\ r$ for every element `r` in `Ring`. And so `Dom::LinearOrdinaryDifferentialOperator` is a noncommutative ring.

Note: Nevertheless, for some reasons, for every element `r` in `Ring`, `Var*r` is automatically rewritten as `r*Var`. See “Example 1” on page 7-228.

If `Dom::LinearOrdinaryDifferentialOperator` is called without any argument, a domain with coefficients in `Dom::ExpressionField(normal)` with derivation `Df` and differential indeterminate `x` is created.

Note: Only commutative differential rings of type `DOM_DOMAIN` are allowed which inherit from `Dom::BaseDomain`. If `Ring` is of type `DOM_DOMAIN` but does not inherit from `Dom::BaseDomain`, the domain `Dom::ExpressionField(normal)` will be used instead.

Note: It is highly recommend to use only coefficient rings with unique zero representation. Otherwise it can happen that, e.g., a polynomial division will not terminate or a wrong degree will be returned.

Examples

Example 1

First we create the domain of linear ordinary differential operators:

```
lodo := Dom::LinearOrdinaryDifferentialOperator()
```

Dom::LinearOrdinaryDifferentialOperator(Df, x, Dom::ExpressionField(normal, iszero ∘ normal))

by default the above domain has coefficients in the field `Dom::ExpressionField(normal)` with derivation `Df` and differential indeterminate `x`.

We can create elements of `lodo` in 3 ways: polynomials in `Df`, list of elements of `Dom::ExpressionField` and with a linear ordinary homogeneous differential equation:

```
lodo(Df^2 + (x + 1)*Df + 2*x), lodo([2*x, x + 1, 1]),
lodo(diff(y(x),x,x) + (x + 1)*diff(y(x),x) + 2*x*y(x), y(x))
```

$$\mathrm{Df}^2 + (x+1)\,\mathrm{Df} + 2\,x,\ \mathrm{Df}^2 + (x+1)\,\mathrm{Df} + 2\,x,\ \mathrm{Df}^2 + (x+1)\,\mathrm{Df} + 2\,x$$

It's easy to obtain the linear differential equation associated to a linear differential operator:

```
L := lodo((x + x^3)*Df^3 + (6*x^2 + 3)*Df^2 - 12):
L(y(x))
```

$$6\,x^2\,\frac{\partial^2}{\partial x^2}\,y(x) - 12\,y(x) + x\,\frac{\partial^3}{\partial x^3}\,y(x) + x^3\,\frac{\partial^3}{\partial x^3}\,y(x) + 3\,\frac{\partial^2}{\partial x^2}\,y(x)$$

and one can also evaluates a differential operator at an expression:

```
L(2*x^2 + 1), L(ln(x)), L(ln(x), Unsimplified)
```

$$0,\ -\frac{12\,x^2\ln(x)+4\,x^2+1}{x^2},\ \frac{2\,(x^3+x)}{x^3}-\frac{6\,x^2+3}{x^2}-12\ln(x)$$

Multiplication of elements of `lodo` is noncommutative but for every element `r` of the coefficients ring one has `Df*r = r*Df`:

```
lodo(x^2*Df), lodo(Df*x^2), lodo(Df)*lodo(x^2)
```

$$x^2\,\mathrm{Df},\ x^2\,\mathrm{Df},\ x^2\,\mathrm{Df}+2\,x$$

Example 2

`Dom::LinearOrdinaryDifferentialOperator` is a domain where the Euclidean division exists but one has to precise if the multiplication of 2 elements of this domain is made on the right or on the left side:

```
L1 := lodo(x*Df^3 + (x^2 - 3)*Df^2 + 4*x*Df + 2):
lodo::leftDivide(L1,lodo(x*Df + 1))
```

quotient	$\mathrm{Df}^2+\frac{\mathrm{Df}\,\left(x^2-4\right)}{x}+2$
remainder	0

```
lodo(x*Df + 1) * %[quotient] = L1
```

$$x\,\mathrm{Df}^3+(x^2-3)\,\mathrm{Df}^2+(4\,x)\,\mathrm{Df}+2=x\,\mathrm{Df}^3+(x^2-3)\,\mathrm{Df}^2+(4\,x)\,\mathrm{Df}+2$$

Hence one has the notions of greatest common divisor, least common multiple on the right and on the left, and a modified version of the extended Euclidean algorithm:

```
L2 := lodo(x*Df + 1):
```

```
ree := lodo::rightExtendedEuclid(L1,L2)
```

$$\left[\left[-\frac{12}{x^2},\ 1,\ -\mathrm{Df}^2-\frac{x^2-6}{x}\,\mathrm{Df}-\frac{2\,(x^2+6)}{x^2}\right],\right.$$
$$\left.\left[\frac{x^3}{12}\,\mathrm{Df}+\frac{x^2}{4},\ -\frac{x^3}{12}\,\mathrm{Df}^3+\left(\frac{x^2}{4}-\frac{x^4}{12}\right)\mathrm{Df}^2-\frac{x^3}{2}\,\mathrm{Df}-\frac{x^2}{2}\right]\right]$$

The right greatest common divisor and the left least common multiple can be read from the above list:

```
iszero(lodo::rightGcd(L1,L2) - ree[1][1]),
iszero(ree[1][1] - (ree[1][2]*L1 + ree[1][3]*L2)),
iszero(lodo::leftLcm(L1,L2) - (-ree[2][1]*L1)),
iszero(-ree[2][1]*L1 - ree[2][2]*L2)
```

TRUE, TRUE, TRUE, TRUE

Example 3

One can compute polynomial, rational and exponential zeros of linear differential operators of any degree provided the ring `Ring` is the field of rational functions of `x`

```
L3 := lodo((x^2 + 1)*x*Df^3 + 3*(2*x^2 + 1)*Df^2 - 12):
lodo::rationalZeros(L3), lodo::exponentialZeros(L3)
```

$$\left\{x^2+\frac{1}{2}\right\},\ \left\{x^2+\frac{1}{2},\ x\sqrt{x^2+1}\right\}$$

even when the operator contains some parameters rationally:

```
lodo::exponentialZeros(
lodo(Df^4 + (b*l - 2*a^2 - a*l*x)*Df^2 + a^4-a^2*b*l + a^3*l*x))
```

$$\left\{e^{a\,x},\ e^{-a\,x}\right\}$$

Example 4

One can factorize linear differential operators into irreducible factors when the ring `Ring` is the field of rational functions of `x`. Nevertheless, the algorithm is complete only for operators of degree at most 3; for higher degree only left and right factors of degree 1 are found:

```
factor(lodo((x^2 + 1)*x*Df^3 + 3*(2*x^2 + 1)*Df^2 - 12)),
factor(lodo(Df^3 + a*x*Df + a + b^3 + a*b*x))
```

$$((x\,(x^2+1))\,\mathrm{Df}+5\,x^2+3)\left(\mathrm{Df}+\frac{x\,(6\,x^2+5)}{(x^2+1)\,(2\,x^2+1)}\right)\left(\mathrm{Df}-\frac{4\,x}{2\,x^2+1}\right),$$

$$(\mathrm{Df}+b)\,(\mathrm{Df}^2-b\,\mathrm{Df}+a\,x+b^2)$$

Here the operator factors into two factors of degree 2 which cannot be found by MuPAD:

```
factor(lodo(Df^2 + x^3 + 1/x^3) * lodo(Df^2 + x^2 - 1/x^3))
```

$$\mathrm{Df}^4+(x^2+x^3)\,\mathrm{Df}^2+\frac{2\,(2\,x^5+3)}{x^4}\,\mathrm{Df}+\frac{-12\,x+x^5+x^6+x^{11}-1}{x^6}$$

Example 5

Solving linear differential operators using the command `solve` is also possible:

```
solve(lodo(Df^2 + (3 - x)/(16*x^2)))
```

$$\left\{x^{1/4}\,e^{-\frac{\sqrt{x}}{2}},\ x^{1/4}\,e^{\frac{\sqrt{x}}{2}}\right\}$$

For certain cases, where the groups associated to the dfferential operators are finite primitive groups of degree 2, a polynomial is returned corresponding to the minimal polynomial of all zeros of the differential operator (they are algebraic over the base field):

```
solve(lodo(Df^2 +
```

```
(-27*x + 32*x^2 + 27)/(144*x^2 - 288*x^3 + 144*x^4)))
```

$$\left\{\text{RootOf}\left({}_Y1^{24} + \left(-4320\,x^2\,(x-1)^3\right)\,{}_Y1^{16} + \left(\left(51840\,\sqrt{3}\,i\right)x^3\,(x-2)\,(x-1)^4\right)\,{}_Y1^{12} + \left(-2799360\,x^4\,(x-1)^6\right)\,{}_Y1^{8} + \left(\left(4478976\,\sqrt{3}\,i\right)x^5\,(x-2)\,(x-1)^7\right)\,{}_Y1^{4} + 2985984\,x^8\,(x-1)^8,\ {}_Y1\right)\right\}$$

For linear differential operators of degree greater than 3 only exponential zeros will be found:

```
solve(lodo(x*Df^4 + (-x + 4)*Df^3 - 3*Df^2 - x^2*Df - x + x^2))
```

$$\left\{\frac{e^x}{x}\right\}$$

Certain second degree linear differential operator can be solved in terms of some special functions (nonliouvillian functions) such as `airyAi`, `besselI` and `whittakerM`:

```
solve(lodo(Df^2 - (x + 1)/(x - 1)^5))
```

$$\left\{\text{airyAi}\left(\sigma_1,\ 0\right)\,(x-1),\ \text{airyBi}\left(\sigma_1,\ 0\right)\,(x-1)\right\}$$

where

$$\sigma_1 = -\frac{2\,2^{1/3}\,\sqrt{3}\,i + 2\,2^{1/3}\,x + 2\,2^{1/3} + 2\,2^{1/3}\,\sqrt{3}\,x\,i}{8\,x - 8}$$

```
solve(lodo(Df^2 - (243 + 4*x^8 + 162*x^2 + 19*x^4)/
                  36/x^2/(x^2 + 3)^2))
```

$$\left\{\frac{\text{M}_{\frac{1}{2},\,-\frac{2}{3}\left(\frac{x^2}{3}+1\right)}}{\sqrt{x}},\ \frac{\text{W}_{\frac{1}{2},\,-\frac{2}{3}\left(\frac{x^2}{3}+1\right)}}{\sqrt{x}}\right\}$$

Parameters

`Var`

An indeterminate of type `DOM_IDENT`. Default is `Df`.

`DVar`

A differential indeterminate of type `DOM_IDENT`. Default is `x`.

`Ring`

An arbitrary commutative differential ring of characteristic zero. Default is `Dom::ExpressionField(normal)`.

`p`

A polynomial expression in `Var`.

`l`

A list corresponding to the coefficients of the differential operator. If n is the length of `l` then the result returned is `l[1] + l[2]*Var + ... + l[n]*Var^(n-1)`.

`eq`

A linear homogeneous differential equation.

`yx`

A function of `DVar` representing the dependent variable of the above linear differential equation.

Methods

Mathematical Methods

`_mult` — Multiplie linear differential operators

```
_mult(<a, b, …>)
```

This method overloads the function `_mult` of the system kernel, i.e. one may use it either in the form `a * b * ...` or in functional notation `_mult(a, b, ...)`.

`_negate` — Negate a linear differential operator

```
_negate(a)
```

This method overloads the function `_negate` of the system kernel, i.e. one may use it either in the form `-a` or in functional notation `_negate(a)`.

`_plus` — Add linear differential operators and coefficient ring elements

```
_plus(<a, b, …>)
```

This method overloads the function `_plus` of the system kernel, i.e. one may use it either in the form `a + b + ...` or in functional notation `_plus(a, b, ...)`.

`_power` — Nth power of a linear differential operator

```
_power(a, n)
```

This method overloads the function `_power` of the system kernel, i.e., one may use it either in the form `a^n` or in functional notation `_power(a,n)`.

`_subtract` — Subtract a linear differential operator

```
_subtract(a, b)
```

This method overloads the function `_subtract` of the system kernel, i.e. one may use it either in the form `a - b` or in functional notation `_subtract(a, b)`.

`adjoint` — Adjoint of a linear differential operator

```
adjoint(a)
```

`companionSystem` — Companion matrix of a linear differential operator

```
companionSystem(a)
```

If `a` is not of positive degree, an error message is issued.

`D` — Derivative of a linear differential operator

```
D(<l>, a)
```

`Dpoly` — Derivative of a linear differential operator

```
Dpoly(<l>, a)
```

`Dpoly(l,a)` computes the partial derivative of `a` with respect to `l`. If `l = [1,...,1]` with `length(l) = n` then the method computes the `n`-th derivative `a`. If `l = []` then the result returned is `a`.

`evalLODO` — Applie an expression to a linear differential operator

```
evalLODO(a, f)
```

This method may be used either in the form `a(f)` or in functional notation `evalLODO(a,f)`.

`exponentialZeros` — Exponential zeros of a linear differential operator

```
exponentialZeros(a)
```

Note: This method only works when `Ring` is the field of rational functions in `DVar`.

`factor` — Factor a linear differential operator

```
factor(a)
```

Note: This method is only available when the base field `Ring` is the field of rational functions in `DVar`. If `a` is of degree greater than or equal to 4 then only left and right factors of degree 1 of `a` will be found. Otherwise, a complete factorization is returned.

This method overloads the function `factor` of the system kernel.

`factors` — List of irreducible factors of a linear differential operator

```
factors(a)
```

`func_call` — Applie an expression to a linear differential operator

```
func_call(a, f, <Unsimplified>)
```

This method may be used either in the form `a(f)` or in functional notation `func_call(a,f)`.

`leftDivide` — Left division of 2 linear differential operators

```
leftDivide(a, b)
```

`leftExtendedEuclid` — Left extended Euclidean algorithm for linear differential operators

```
leftExtendedEuclid(a, b)
```

`leftExtendedGcd` — Coefficient in the left extended Euclidean algorithm

```
leftExtendedGcd(a, b)
```

`leftGcd` — Left greatest common divisor of linear differential operators

```
leftGcd(a, b)
```

`leftLcm` — Left least common multiple of linear differential operators

```
leftLcm(a, b)
```

`leftQuotient` — Left quotient of linear differential operators

```
leftQuotient(a, b)
```

`leftRemainder` — Left remainder of linear differential operators

```
leftRemainder(a, b)
```

`makeIntegral` — Integral form of a linear differential operator

```
makeIntegral(a)
```

`monic` — Normalize a linear differential operator

```
monic(a)
```

`polynomialZeros` — Polynomial zeros of a linear differential operator

```
polynomialZeros(a)
```

Note: This method only works when `Ring` is the field of rational functions in `DVar`.

`rationalZeros` — Rational zeros of a linear differential operator

```
rationalZeros(a)
```

Note: This method only works when `Ring` is the field of rational functions in `DVar`.

`rightDivide` — Right division of 2 linear differential operators

```
rightDivide(a, b)
```

`rightExtendedEuclid` — Right extended Euclidean algorithm for linear differential operators

```
rightExtendedEuclid(a, b)
```

`rightExtendedGcd` — Coefficient in the right extended Euclidean algorithm

```
rightExtendedGcd(a, b)
```

`rightGcd` — Right greatest common divisor of linear differential operators

```
rightGcd(a, b)
```

`rightLcm` — Right least common multiple of linear differential operators

```
rightLcm(a, b)
```

`rightQuotient` — Right quotient of linear differential operators

```
rightQuotient(a, b)
```

`rightRemainder` — Right remainder of linear differential operators

```
rightRemainder(a, b)
```

`solve` — Zero of a linear differential operator

```
solve(a, <Transform>, <Irreducible>)
```

The algorithm for finding liouvillian solutions is complete for operators of degree at most 2 and enables to solve partially operators of higher degree (i.e. it finds all exponential solutions). The algorithm for finding solutions in terms of special functions (nonliouvillian solutions) is not complete even for the degree 2.

When option `Transform` is given the unimodular transformation is performed unconditionally and when option `Irreducible` is given, `a` is assumed to be irreducible.

Note: This method only works when `Ring` is the field of rational functions in `DVar`.

This method overloads the function `solve` of the system kernel.

`symmetricPower` — Symmetric power of a linear differential operator

```
symmetricPower(a, m)
```

`unimodular` — Unimodular transformation of a linear differential operator

```
unimodular(a, <Transform>)
```

If the option `Transform` is given then `a` is transformed unconditionally even if `a` has yet a unimodular Galois group.

Access Methods

`coeff` — Coefficient of a linear differential operator

```
coeff(a)
```

```
coeff(a, Var, n)
```

```
coeff(a, n)
```

`coeff(a,Var,n)` returns the coefficient of the term `Var^n` as an element of the coefficient ring `Ring`, where `a` is a linear differential operator in the variable `Var`.

`coeff(a,n)` returns the coefficient of the term `Var^n` as an element of the coefficient ring `Ring`, where `a` is a linear differential operator in the variable `Var`.

This method overloads the function `coeff` of the system kernel.

`degree` — Degree of a linear differential operator

```
degree(a)
```

The degree of the zero polynomial is defined as zero.

This method overloads the function `degree` for polynomials.

`vectorize` — List of coefficients of a linear differential operator

```
vectorize(a)
```

Conversion Methods

`convert` — Conversion to a linear differential operator

```
convert(a)
```

`FAIL` is returned if the conversion fails.

`expr` — Conversion into an object of a kernel domain

```
expr(a)
```

This method overloads the function `expr` of the system kernel.

`TeX` — TeX formatting of a linear differential operator

```
TeX(a)
```

This method is used by the function `generate::TeX`.

Algorithms

Some references on linear differential equations/operators:

- `_mult`, `_divide`, `_invert`, `_power`
- `_plus`, `_negate`, `_subtract`
- `abs`

- cos, arccos, cosh, arccosh, cot, arccot, coth, arccoth, csc, arccsc, csch, arccsch, sec, arcsec, sech, arcsech, sin, arcsin, sinh, arcsinh, tan, arctan, tanh, arctanh
- dirac, heaviside
- exp, ln
- sign

See Also

MuPAD Domains
Dom::UnivariatePolynomial

Dom::Matrix

Matrices

Syntax

Domain Creation

```
Dom::Matrix(<R>)
```

Element Creation

```
Dom::Matrix(R)(Array)
Dom::Matrix(R)(List)
Dom::Matrix(R)(ListOfRows)
Dom::Matrix(R)(Matrix)
Dom::Matrix(R)(m, n)
Dom::Matrix(R)(m, n, Array)
Dom::Matrix(R)(m, n, List)
Dom::Matrix(R)(m, n, ListOfRows)
Dom::Matrix(R)(m, n, Table)
Dom::Matrix(R)(m, n, [(i_1, j_1) = value_1, (i_2, j_2) = value_2, …])
Dom::Matrix(R)(m, n, f)
Dom::Matrix(R)(m, n, List, Diagonal)
Dom::Matrix(R)(m, n, g, Diagonal)
Dom::Matrix(R)(m, n, List, Banded)
Dom::Matrix(R)(1, n, Array)
Dom::Matrix(R)(1, n, List)
Dom::Matrix(R)(1, n, Table)
```

```
Dom::Matrix(R)(1, n, [j1 = value1, j2 = value2, …])
Dom::Matrix(R)(m, 1, Array)
Dom::Matrix(R)(m, 1, List)
Dom::Matrix(R)(m, 1, Table)
Dom::Matrix(R)(m, 1, [i1 = value1, i2 = value2, …])
```

Description

Domain Creation

`Dom::Matrix(R)` creates domains of matrices over a component domain `R` of category `Cat::Rng` (a ring, possibly without unit).

If the optional parameter `R` is not given, `Dom::ExpressionField()` is used as component domain. Matrices of this type accept arbitrary MuPAD expressions (numbers, symbols etc.) as entries. The name `matrix` is an alias for this default matrix domain `Dom::Matrix()`.

A vector with n entries is either an $n \times 1$ matrix (a column vector), or a $1 \times n$ matrix (a row vector).

Arithmetical operations with matrices can be performed by using the standard arithmetical operators of MuPAD.

E.g., if `A` and `B` are two matrices defined by `Dom::Matrix(R)`, `A + B` computes the sum, and `A * B` computes the product of the two matrices, provided that the dimensions are appropriate.

Similarly, `A^(-1)` or `1/A` computes the inverse of a square matrix `A` if it exists. Otherwise, `FAIL` is returned. See “Example 1” on page 7-246.

Many system functions are overloaded for matrices, such as `map`, `subs`, `has`, `zip`, E.g., use `conjugate` to compute the complex conjugate of a matrix, `norm` to compute matrix norms, or `exp` to compute the exponential of a matrix.

Most of the functions in the MuPAD linear algebra package linalg work with matrices. For example, the command `linalg::gaussJordan(A)` performs Gauss-Jordan elimination on `A` to transform `A` to its reduced row echelon form.

See the documentation of linalg for a list of available functions of this package.

The domain `Dom::Matrix(R)` represents matrices over `R` of arbitrary size. Therefore, it does not have any algebraic structure (other than being a *set* of matrices).

In this help page, we use the following notations for a matrix A (an element of `Dom::Matrix(R)`):

- *nrows*(A) denotes the number of rows of A.
- *ncols*(A) denotes the number of columns of A.
- A *row index* is an integer in the range from 1 to *nrows*(A).
- A *column index* is an integer in the range from 1 to *ncols*(A).

Note: The number of rows and columns, respectively, of a matrix must be less than 2^{31}.

Note: The components of a matrix are no longer evaluated after the creation of the matrix, i.e., if they contain free identifiers they will not be replaced by their values.

Element Creation

`Dom::Matrix(R)(Array)` and `Dom::Matrix(R)(Matrix)` create a new matrix with the dimension and the components of `Array` and `Matrix`, respectively.

The components of `Array` or `Matrix` are converted to elements of the domain `R`. An error message is issued if one of these conversions fails.

The creation of (sparse) matrices via arrays is useful for matrices of moderate size. Note that indexed assignments to arrays are much faster than the corresponding indexed assignments to matrices. However, since all elements of the array (including the zeroes) need to be filled in before conversion to a (sparse) matrix, memory is wasted for very large and very sparse matrices. In such a situation, one should define a table containing only the non-zero elements and convert the table to a matrix (see below).

`Dom::Matrix(R)(List)` creates an $m \times 1$ column vector with components taken from the nonempty list, where m is the number of entries of `List`.

One may also use a list of equations to create an object of `Dom::Matrix`. In this case the entries of the list must be of the form `(i, j) = value`, where `i` and `j` denote the row

and column index and `value` the coefficient of the matrix. `i` and `j` need to be positive integers.

`Dom::Matrix(R)(ListOfRows)` creates an $m \times n$ matrix with components taken from the nested list `ListOfRows`, where m is the number of inner lists of `ListOfRows`, and n is the maximal number of elements of an inner list. Each inner list corresponds to a row of the matrix. Both m and n must be non-zero.

If an inner list has less than n entries, the remaining components in the corresponding row of the matrix are set to zero.

The entries of the inner lists are converted to elements of the domain `R`. An error message is issued if one of these conversions fails.

The call `Dom::Matrix(R)(m, n)` returns the $m \times n$ zero matrix.

Use the method `"identity"` to create the $n \times n$ identity matrix.

The call `Dom::Matrix(R)(m, n, Array)` creates an $m \times n$ matrix with components taken from `Array`, which must be an array or an hfarray. `Array` must have $m\,n$ operands. The first m operands define the first row, the next m operands define the second row, etc. The formatting of the array is irrelevant. E.g., any array with 6 elements can be used to create a matrix of dimension 1×6, or 2×3, or 3×2, or 6×1.

`Dom::Matrix(R)(m, n, List)` creates an $m \times n$ matrix with components taken row after row from the non-empty list. The list must contain $m\,n$ entries.

`Dom::Matrix(R)(m, n, ListOfRows)` creates an $m \times n$ matrix with components taken from the list `ListOfRows`.

If $m \geq 2$ and $n \geq 2$, then `ListOfRows` must consist of at most `m` inner lists, each having at most `n` entries. The inner lists correspond to the rows of the returned matrix.

If an inner list has less than `n` entries, the remaining components of the corresponding row of the matrix are set to zero. If there are less than `m` inner lists, the remaining lower rows of the matrix are filled with zeroes.

`Dom::Matrix(R)(m, n, Table)` creates an $m \times n$ matrix with components taken from the table `Table`.

By defining the entries of the table first, one can easily create large and sparse matrices. The entry `Table[i, j]` of the table will be the entry in the `i`-th row and the `j`-th

column of the matrix. Therefore, the table needs to be indexed by positive integers `i` and `j`.

`Dom::Matrix(R)(m, n, [(i1, j1) = value1, (i2, j2) = value2, ...])` is a further way to create a matrix specifying only the non-zero entries `A[i1, j1] = value1`, `A[i2, j2] = value2` etc. The ordering of the entries in the input list is irrelevant.

`Dom::Matrix(R)(m, n, f)` returns the matrix whose (i, j)-th component is the value of the function call `f(i, j)`. The row index i ranges from 1 to m and the column index j from 1 to n.

The function values are converted to elements of the domain `R`. An error message is issued if one of these conversions fails.

`Dom::Matrix(R)(1, n, Array)` returns the $1 \times n$ row vector with components taken from `Array`. The `array` or `hfarray` `Array` must have `n` entries.

The entries of the array are converted to elements of the domain `R`. An error message is issued if one of these conversions fails.

`Dom::Matrix(R)(1, n, List)` returns the $1 \times n$ row vector with components taken from `List`. The list `List` must have at most `n` entries. If there are fewer entries, the remaining vector components are set to zero.

The entries of the list are converted to elements of the domain `R`. An error message is issued if one of these conversions fails.

`Dom::Matrix(R)(1, n, Table)` returns the $1 \times n$ row vector with components taken from `Table`. The table `Table` must not have more than `n` entries. If there are fewer entries, the remaining vector components are regarded as zero.

`Dom::Matrix(R)(m, 1, Array)` returns the $m \times 1$ column vector with components taken from `Array`. The `array` or `hfarray` `Array` must have `m` entries.

The entries of the array are converted to elements of the domain `R`. An error message is issued if one of these conversions fails.

`Dom::Matrix(R)(m, 1, List)` returns the $m \times 1$ column vector with components taken from `List`. The list `List` must have at most `m` entries. If there are fewer entries, the remaining vector components are set to zero.

The entries of the list are converted to elements of the domain `R`. An error message is issued if one of these conversions fails.

`Dom::Matrix(R)(m, 1, Table)` returns the $m \times 1$ column vector with components taken from `Table`. The table `Table` must have no more than `m` entries. If there are fewer entries, the remaining vector components are regarded as zero.

Superdomain

`Dom::BaseDomain`

Axioms

If R has `Ax::canonicalRep`, then `Ax::canonicalRep`.

Categories

`Cat::Matrix(R)`

Examples

Example 1

Whenever possible, one should use `Dom::ExpressionField()` as the coefficient domain of matrices – therefore `Dom::ExpressionField()` is the default coefficient domain of matrices.

The components of matrices over `Dom::ExpressionField()` can be arbitrary arithmetical expressions. Consider

```
Mat := Dom::Matrix()
```

```
Dom::Matrix()
```

We assigned the domain to the identifier `Mat` and now we can define a matrix A of two rows, where each row is a list of two elements by the following line:

```
A := Mat([[1, 5], [2, 3]])
```

$$\begin{pmatrix} 1 & 5 \\ 2 & 3 \end{pmatrix}$$

In the same way, we define the following 2 ×3 matrix:

```
B := Mat([[-1, 5/2, 3], [1/3, 0, 2/5]])
```

$$\begin{pmatrix} -1 & \frac{5}{2} & 3 \\ \frac{1}{3} & 0 & \frac{2}{5} \end{pmatrix}$$

and perform matrix arithmetic using the standard arithmetical operators of MuPAD, e.g., the matrix product $A\,B$, the fourth power of A as well as the scalar multiplication of A times $\frac{1}{3}$:

```
A * B, A ^ 4, 1/3 * A
```

$$\begin{pmatrix} \frac{2}{3} & \frac{5}{2} & 5 \\ -1 & 5 & \frac{36}{5} \end{pmatrix}, \begin{pmatrix} 281 & 600 \\ 240 & 521 \end{pmatrix}, \begin{pmatrix} \frac{1}{3} & \frac{5}{3} \\ \frac{2}{3} & 1 \end{pmatrix}$$

The matrices A and B have different dimensions, and therefore the sum of A and B is not defined. MuPAD issues an error message:

```
A + B
```

```
Error: The dimensions do not match. [(Dom::Matrix(Dom::ExpressionField()))::_plus]
```

To compute the inverse of A, just enter:

```
1/A
```

$$\begin{pmatrix} -\frac{3}{7} & \frac{5}{7} \\ \frac{2}{7} & -\frac{1}{7} \end{pmatrix}$$

If a matrix is not invertible, `FAIL` is the result of this operation. For example, the matrix:

```
C := Mat(2, 2, [[2]])
```

$$\begin{pmatrix} 2 & 0 \\ 0 & 0 \end{pmatrix}$$

is not invertible, hence:

```
C^(-1)
```

FAIL

```
delete A, B, C:
```

Example 2

We create the domain of matrices over the coeffcient ring `Dom::ExpressionField()`:

```
Mat := Dom::Matrix()
```

Dom::Matrix()

Beside standard matrix arithmetic, the library linalg offers many functions dealing with matrices. For example, if one wants to compute the rank of a matrix, use `linalg::rank`:

```
A := Mat([[1, 2], [2, 4]])
```

$$\begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix}$$

```
linalg::rank(A)
```

1

Use `linalg::eigenvectors` to compute eigenvalues and eigenvectors of the matrix A:

```
linalg::eigenvectors(A)
```

$$\left[\left[0,\ 1,\ \left[\begin{pmatrix} -2 \\ 1 \end{pmatrix}\right]\right],\ \left[5,\ 1,\ \left[\begin{pmatrix} \frac{1}{2} \\ 1 \end{pmatrix}\right]\right]\right]$$

Try `info(linalg)` for a list of available functions, or enter `help(linalg)` for details about the library linalg.

Some of the functions in the `linalg` package simply serve as "interface" functions for methods of a matrix domain described above. For example, `linalg::transpose` uses the method `"transpose"` to get the transposed matrix. The function `linalg::gaussElim` applies Gaussian elimination to a matrix by calling the method `"gaussElim"`:

```
linalg::gaussElim(A) = A::dom::gaussElim(A)[1]
```

$$\begin{pmatrix} 1 & 2 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 0 & 0 \end{pmatrix}$$

In contrast to the methods of the domain `Dom::Matrix(R)`, the corresponding functions of the `linalg` packages do extended checking of their input parameters. Note that there might be minor differences in the functionality of the `linalg` functions and the matrix methods. E.g., the option `ColumnElimination` is not available in `linalg::gaussElim`, but only in the `"gaussElim"` method of the matrix domain:

```
A::dom::gaussElim(A, ColumnElimination)
```

$$\left[\begin{pmatrix} 1 & 0 \\ 2 & 0 \end{pmatrix}, 1, 0, \{1\}\right]$$

```
delete A:
```

Example 3

We create the default matrix domain `Dom::Matrix()`. As a shortcut, this domain can also be created via `matrix`:

```
A := matrix([[ 1, 2, 3, 4],
             [ 2, 0, 4, 1],
             [-1, 0, 5, 2]])
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 0 & 4 & 1 \\ -1 & 0 & 5 & 2 \end{pmatrix}$$

```
domtype(A)
```

```
Dom::Matrix()
```

Matrix components can be extracted by the index operator `[]`:

```
A[2, 1] * A[1, 2] - A[3, 1] * A[1, 3]
```

```
7
```

If one of the indices is not in its valid range, an error message is issued. Assignments to matrix components are performed similarly:

```
delete a:
A[1, 2] := a^2: A
```

$$\begin{pmatrix} 1 & a^2 & 3 & 4 \\ 2 & 0 & 4 & 1 \\ -1 & 0 & 5 & 2 \end{pmatrix}$$

Beside the usual indexing of matrix components, it is also possible to extract submatrices from a given matrix. The following call creates the submatrix of A which consists of the rows 2 to 3 and columns 1 to 3 of A:

```
A[2..3, 1..3]
```

$$\begin{pmatrix} 2 & 0 & 4 \\ -1 & 0 & 5 \end{pmatrix}$$

The index operator does not allow to insert submatrices into a given matrix. This is implemented by the function `linalg::substitute`.

```
delete A:
```

Example 4

In the following examples, we demonstrate the different ways of creating matrices. We work with matrices defined over the field $\mathbb{Z}_{19}$, i.e., the field of integers modulo 19. This component ring can be created with the domain constructor `Dom::IntegerMod`.

We start by giving a list of rows, where each row is a list of row entries:

```
MatZ19 := Dom::Matrix(Dom::IntegerMod(19)):
MatZ19([[1, 2], [2]])
```

$$\begin{pmatrix} 1 \bmod 19 & 2 \bmod 19 \\ 2 \bmod 19 & 0 \bmod 19 \end{pmatrix}$$

The elements of the two inner lists, the row entries, were converted to elements of the domain `Dom::IntegerMod(19)`.

The number of rows is the number of sublists of the argument, i.e., $m = 2$. The number of columns is determined by the length of the inner list with the most entries, which is the first inner list with two entries. Missing entries in the other inner lists are treated as zero components. The call:

```
MatZ19(4, 4, [[1, 2], [2]])
```

$$\begin{pmatrix} 1 \bmod 19 & 2 \bmod 19 & 0 \bmod 19 & 0 \bmod 19 \\ 2 \bmod 19 & 0 \bmod 19 & 0 \bmod 19 & 0 \bmod 19 \\ 0 \bmod 19 & 0 \bmod 19 & 0 \bmod 19 & 0 \bmod 19 \\ 0 \bmod 19 & 0 \bmod 19 & 0 \bmod 19 & 0 \bmod 19 \end{pmatrix}$$

fixes the dimension of the matrix. Missing entries and inner lists are treated as zero components and zero rows, respectively.

An error message is issued if one of the given entries cannot be converted to an element over $\mathbb{Z}_{19}$:

```
MatZ19([[2, 3], [-1, I]])
```

```
Error: Cannot define a matrix over 'Dom::IntegerMod(19)'. [(Dom::Matrix(Dom::IntegerMod
```

```
delete MatZ19:
```

Example 5

This example illustrates how to create a matrix with components given as values of an index function. First we create the 2 ×2 Hilbert matrix (see also the functions `linalg::hilbert` and `linalg::invhilbert`):

```
Dom::Matrix()(2, 2, (i, j) -> 1/(i + j - 1))
```

$$\begin{pmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{3} \end{pmatrix}$$

Note the difference when working with expressions and functions. If you give an expression it is treated as a function in the row and column indices:

```
delete x:
Dom::Matrix()(2, 2, x), Dom::Matrix()(2, 2, (i, j) -> x)
```

$$\begin{pmatrix} x(1,1) & x(1,2) \\ x(2,1) & x(2,2) \end{pmatrix}, \begin{pmatrix} x & x \\ x & x \end{pmatrix}$$

Example 6

Diagonal matrices can be created with the option `Diagonal` and a list of diagonal components:

```
Mat := Dom::Matrix():
Mat(3, 4, [1, 2, 3], Diagonal)
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \end{pmatrix}$$

Hence, to define the $n \times n$ identity matrix, you can enter:

```
Mat(3, 3, [1 $ 3], Diagonal)
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

or call:

```
Mat(3, 3, x -> 1, Diagonal)
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The easiest way to create the identity matrix, however, is to use the method `"identity"`:

```
Mat::identity(3)
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

```
delete Mat:
```

Example 7

Toeplitz matrices can be defined with the option `Banded`. The following call defines a three-banded matrix with the component 2 on the main diagonal and the component - 1 on the first subdiagonal and superdiagonal:

```
Dom::Matrix()(4, 4, [-1, 2, -1], Banded)
```

$$\begin{pmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{pmatrix}$$

Example 8

Some system functions can be applied to matrices, such as `norm`, `expand`, `diff`, `conjugate`, or `exp`.

For example, to expand the components of the matrix:

```
delete a, b:
A := Dom::Matrix()(
  [[(a - b)^2, a^2 + b^2], [a^2 + b^2, (a - b)*(a + b)]]
```

```
)
```

$$\begin{pmatrix} (a-b)^2 & a^2+b^2 \\ a^2+b^2 & (a+b)\,(a-b) \end{pmatrix}$$

enter:

```
expand(A)
```

$$\begin{pmatrix} a^2-2\,a\,b+b^2 & a^2+b^2 \\ a^2+b^2 & a^2-b^2 \end{pmatrix}$$

If you want to differentiate the matrix components, then call for example:

```
diff(A, a)
```

$$\begin{pmatrix} 2\,a-2\,b & 2\,a \\ 2\,a & 2\,a \end{pmatrix}$$

To substitute matrix components by some values, enter:

```
subs(A, a = 1, b = -1)
```

$$\begin{pmatrix} 4 & 2 \\ 2 & 0 \end{pmatrix}$$

The function `zip` can also be applied to matrices. The following call combines two matrices *A* and *B* by dividing each component of *A* by the corresponding component of *B*:

```
A := Dom::Matrix()([[4, 2], [9, 3]]):
B := Dom::Matrix()([[2, 1], [3,-1]]):
zip(A, B, `/`)
```

$$\begin{pmatrix} 2 & 2 \\ 3 & -3 \end{pmatrix}$$

The quoted character `` `/` `` is another notation for the function `_divide`, the functional form of the division operator `/`.

If one needs to apply a function to the components of a matrix, then use the function `map`. For example, to simplify the components of the matrix:

```
C := Dom::Matrix()(
  [[sin(x)^2 + cos(x)^2, cos(x)*tan(x)],
  [(a^2 - b^2)/(a + b), 1]]
)
```

$$\begin{pmatrix} \cos(x)^2+\sin(x)^2 & \cos(x)\tan(x) \\ \frac{a^2-b^2}{a+b} & 1 \end{pmatrix}$$

call:

```
map(C, Simplify)
```

$$\begin{pmatrix} 1 & \sin(x) \\ a-b & 1 \end{pmatrix}$$

```
delete A, B, C:
```

Example 9

However, there may appear some unexpected results using the function `diff` in the context of matrices. The derivative of the following unspecified function `f` of a matrix is computed due to the chain rule:

```
diff(f(matrix([[a*x^2, b], [c, d]])), x)
```

$$\begin{pmatrix} 2\,a\,x\,f'\left(\begin{pmatrix} a\,x^2 & b \\ c & d \end{pmatrix}\right) & 0 \\ 0 & 0 \end{pmatrix}$$

Usually, the function `f` would implicitly be assumed to be scalar. Hence, the derivative of `f` should be scalar as well. In the above situation the chain rule is applied for differentiation: the inner function is the matrix containing the symbolic components

`a*x^2`, `b`, `c` and `d`. Its derivative is computed by simply applying `diff` to each component of the matrix:

```
diff(matrix([[a*x^2, b], [c, d]]), x)
```

$$\begin{pmatrix} 2\,a\,x & 0 \\ 0 & 0 \end{pmatrix}$$

Finally, the exterior unspecified function `f` is implicitly assumed to be scalar, such that each component of the derivative of the inner function is multiplied by the exterior differentiation.

Example 10

A column vector is represented by a 2×1 matrix:

```
Mat := Dom::Matrix():
v := Mat(2, 1, [1, 2])
```

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

The dimension of this vector is:

```
Mat::matdim(v)
```

$$[2, 1]$$

The length of a vector may also be queried by `linalg::vecdim` or `nops(v)`:

```
linalg::vecdim(v)
```

$$2$$

The *i*th component of this vector can be extracted in two ways: either by `v[i, 1]` or by `v[i]`:

```
v[1], v[2]
```

$$1, 2$$

We compute the 2-norm of `v` by the following call:

```
norm(v, 2)
```

$$\sqrt{5}$$

```
delete Mat, v:
```

Example 11

We create random matrices over the field of the rational numbers. Consider a random matrix `A1` with 3 rows and 3 columns:

```
Mat := Dom::Matrix(Dom::Rational):
A1 := Mat::random(3, 3)
```

$$\begin{pmatrix} -\frac{65}{824} & \frac{221}{72} & -\frac{245}{597} \\ \frac{741}{814} & \frac{229}{220} & \frac{747}{79} \\ \frac{764}{979} & -\frac{535}{617} & -\frac{535}{477} \end{pmatrix}$$

A second matrix `A2` should contain at most 2 non-zero entries. We can create such a matrix by using 2 as the third argument for `random`:

```
A2 := Mat::random(3, 3, 2)
```

$$\begin{pmatrix} 0 & 0 & \frac{905}{906} \\ 0 & 0 & 0 \\ 0 & \frac{4}{133} & 0 \end{pmatrix}$$

The product of these matrices is given by

```
C := A1 * A2
```

$$\begin{pmatrix} 0 & -\frac{140}{11343} & -\frac{58825}{746544} \\ 0 & \frac{2988}{10507} & \frac{223535}{245828} \\ 0 & -\frac{2140}{63441} & \frac{345710}{443487} \end{pmatrix}$$

By default, matrices are displayed like 'dense' arrays with zeroes in the empty places. For sparse matrices of large column and/or row dimension, such a 'dense' print mode is not appropriate: formatting of the print output would be very time consuming. Further, a 'dense' print output is not very informative for sparse matrices. For this reason, the `"doprint"` method provides a sparse output mode printing only the non-zero entries:

```
C::dom::doprint(C)
```

```
Dom::Matrix(Dom::Rational)(3, 3, [(1, 2) = -140/11343, (1, 3) = -58825/746544, (2, 2) = 2988/10507\
, (2, 3) = 223535/245828, (3, 2) = -2140/63441, (3, 3) = 345710/443487])
```

With this method, one can also print large sparse matrices. We create a random sparse matrix with 100 rows, 200 columns and at most 6 non-zero entries:

```
X := Mat::random(100, 200, 6):  print(X)
```

```
Warning: This matrix is too large for display. To see all nonzero entries of a matrix A
```

```
Dom::Matrix(Dom::Rational)(100, 200, ["..."])
```

```
 Warning: This matrix is too large for display.
If you want to see all nonzero entries of a matrix, say A, then call
'A::dom::doprint(A)'. [(Dom::Matrix(Dom::Rational))::print]
```

```
 Warning: This matrix is too large for display.
If you want to see all nonzero entries of a matrix, say A, then call
'A::dom::doprint(A)'. [(Dom::Matrix(Dom::Rational))::print]
```

```
Dom::Matrix(Dom::Rational)(100, 200, ["..."])
```

The warning speaks for itself. `X` is regarded as 'too large for display' since, with the default 'dense' output mode, the sparse matrix would be printed as a huge array-like

structure of dimension 100×200 with (integer) zeroes in the empty places. The sparse print mode should be used:

```
X::dom::doprint(X)
```

```
Dom::Matrix(Dom::Rational)(100, 200, [(16, 64) = 448/765, (19, 4) = -61/702, (27, 126) = 343/304, \
  (46, 42) = 433/49, (68, 176) = 451/483, (100, 97) = -235/174])
```

For convenience, there is a function `doprint` that calls this method by just entering:

```
doprint(X)
```

```
Dom::Matrix(Dom::Rational)(100, 200, [(16, 64) = 448/765, (19, 4) = -61/702, (27, 126) = 343/304, \
  (46, 42) = 433/49, (68, 176) = 451/483, (100, 97) = -235/174])
```

```
delete Mat, A1, A2, C, X:
```

Parameters

`R`

A ring, i.e., a domain of category `Cat::Rng`. The default ring is `Dom::ExpressionField()`.

`Array`

A one- or two-dimensional `array` or `hfarray`

`Matrix`

A matrix, i.e., an element of a domain of category `Cat::Matrix`

`m, n`

Matrix dimension (positive integers)

`List`

A list of matrix components

`ListOfRows`

A list of at most `m` rows; each row given as a list of at most `n` matrix components

`Table`

A table of coefficients of the matrix for sparse input

`f`

A function or a functional expression with two parameters (the row and column index)

`g`

A function or a functional expression with one parameter (the row index)

Options

`Diagonal`

Create a diagonal matrix

With the option `Diagonal`, diagonal matrices can be created with diagonal elements taken from a list, or computed by a function or a functional expression.

`Dom::Matrix(R)(m, n, List, Diagonal)` creates the $m \times n$ diagonal matrix, whose diagonal elements are the entries of `List`.

`List` must have at most $min(m, n)$ entries. If it has fewer elements, the remaining diagonal elements are set to zero.

The entries of `List` are converted to elements of the domain `R`. An error message is issued if one of these conversions fails.

`Dom::Matrix(R)(m, n, g, Diagonal)` returns the sparse matrix whose ith diagonal element is `g(i, i)`, where the index i runs from 1 to $min(m, n)$.

The function values are converted to elements of the domain `R`. An error message is issued if one of these conversions fails.

`Banded`

Create a Toeplitz matrix

`Dom::Matrix(R)(m, n, List, Banded)` creates an $m \times n$ Toeplitz matrix with the elements of `List` as entries. The number of entries of `List` must be odd, say $2h + 1$, and must not exceed $2\,min(m, n) - 1$. The bandwidth of the resulting matrix is at most $2h + 1$.

A Toeplitz matrix is a matrix where the elements of each band are identical. See also "Example 7" on page 7-253.

All elements of the main diagonal of the created matrix are initialized with the middle element of `List`. All elements of the i-th subdiagonal are initialized with the $(h + 1 - i)$-th element of `List`. All elements of the i-th superdiagonal are initialized with the $(h + 1 + i)$-th element of `List`. All entries on the remaining sub- and superdiagonals are set to zero.

The entries of `List` are converted to elements of the domain `R`. An error message is issued if one of these conversions fails.

Entries

"isSparse"	is always `TRUE`.
"randomDimen"	is set to `[10, 10]`. See the method `"random"` below for details.

Methods

Mathematical Methods

`_divide` — Divide matrices

`_divide(A, B)`

An error message is issued if the dimensions of `A` and `B` do not match.

This method only exists if `R` is a commutative ring with a unit, i.e., a domain of category `Cat::Ring`.

This method overloads the system function `_divide` for matrices, i.e., one may use it in the form `A / B`, or in functional notation: `_divide(A, B)`.

_invert — Compute the inverse of a matrix

```
_invert(A, Normal = b)
```

This method only exists if R is a domain of category Cat::Ring.

This method overloads the system function _invert for matrices, i.e., one may use it in the form 1/A or A^(-1), or in functional notation: _invert(A).

If Normal = TRUE, then the matrix inverse is always returned in a normalized form. For details about normalization, see normal. If Normal = FALSE, then the matrix inverse can appear in a normalized form, but normalization is not guaranteed. By default Normal = TRUE.

Normal affects the results only if a matrix contains variables or exact expressions, such as sqrt(5) or sin(PI/7).

_mod — Map the modulo operator to the elements of a matrix

```
_mod(A, n)
```

n must be non-zero, and a mod n must be defined for every entry a of A.

This method overloads the function _mod for matrices; one may use it in the form A mod n, or in functional notation: _mod(A, n).

_mult — Multiplie matrices by matrices, vectors and scalars

```
_mult(x, y)
```

```
_mult(x, y)
```

If y is of the domain type R or can be converted to such an element, the corresponding scalar multiplication is computed.

Otherwise, y is converted to a matrix of the domain type of x. If this conversion fails, this method calls the method "_mult" of the domain of y giving all arguments in the same order.

If x is a matrix of the same domain type as y, the matrix product $x\,y$ is computed. An error message is issued if the dimensions of the matrices do not match.

If x is of the domain type R or can be converted to such an element, the corresponding scalar multiplication is computed.

Otherwise, `x` is converted to a matrix of the domain type of `y`. If this conversion fails, `FAIL` is returned.

This method handles more than two arguments by calling itself recursively with the first half of all arguments and the last half of all arguments. Then the product of these two results is computed with the system function `_mult`.

This method overloads the system function `_mult` for matrices, i.e., one may use it in the form `x * y`, or in functional notation: `_mult(x, y)`.

`_negate` — Negate a matrix

```
_negate(A)
```

This method overloads the system function `_negate` for matrices, i.e., one may use it in the form `-A`, or in functional notation: `_negate(A)`.

`_plus` — Add matrices

```
_plus(A, B, …)
```

The arguments `A, B , ...` are converted to matrices of the domain type `Dom::Matrix(R)`. `FAIL` is returned if one of these conversions fails.

This method overloads the system function `_plus` for matrices, i.e., one may use it in the form `A + B`, or in functional notation: `_plus(A, B)`.

`_power` — Integer power of a matrix

```
_power(A, n)
```

If the power `n` is a negative integer then `A` must be nonsingular and `R` must be a domain of category `Cat::IntegralDomain`. Otherwise `FAIL` is returned.

If `n` is zero and the component ring `R` is a ring with no unit (i.e., of category `Cat::Rng`, but not of category `Cat::Ring`), `FAIL` is returned.

This method overloads the system function `_power` for matrices, i.e., one may use it in the form `A^n`, or in functional notation: `_power(A, n)`.

`conjugate` — Complex conjugate of a matrix

```
conjugate(A)
```

This method only exists if R implements the method "conjugate", which computes the complex conjugate of an element of the domain R.

This method overloads the system function conjugate for matrices, i.e., one may use it in the form conjugate(A).

cos — Cosine of a matrix

```
cos(A)
```

If A is not square, an error message is issued. If the component domain of A does not allow the computation of cos(elem) for an arbitrary element elem of the component ring, FAIL is returned.

This method uses the function numeric::expMatrix for a floating-point approximation of the exponential of A if A is defined over the domain Dom::Float.

If some eigenvalues of A do not exist in R or cannot be computed, then FAIL is returned.

In the symbolic case the functions exp and linalg::jordanForm are called. The latter may not be able to compute the Jordan form of A. In this case FAIL is returned.

This method only exists if R is a domain of category Cat::Field.

This method overloads the function cos for matrices, i.e., one may use it in the form cos(A).

diff — Differentiation of matrix components

```
diff(A, …)
```

This method only exists if R implements the method "diff".

This method overloads the system function diff for matrices, i.e., one may use it in the form diff(A, ...). See “Example 8” on page 7-253 and “Example 9” on page 7-255.

equal — Equality test of matrices

```
equal(A, B)
```

Note that if R has the axiom Ax::systemRep then normal is used to simplify the components of A and B before testing their equality.

exp — Exponential of a matrix

```
exp(A, <t>)
```

If A is not square, an error message is issued. If the component domain of A does not allow the computation of `exp(elem)` for an arbitrary element `elem` of the component ring, `FAIL` is returned.

This method uses the function `numeric::expMatrix` for a floating-point approximation of the exponential of A if `A*t` contains at least one floating-point number and all entries can be converted to floating-point numbers.

If some eigenvalues of A do not exist in R or cannot be computed, then `FAIL` is returned.

In the symbolic case, the function `linalg::jordanForm` is called, which may not be able to compute the Jordan form of A. In this case `FAIL` is returned.

This method only exists if R is a domain of category `Cat::Field`.

This method overloads the system function `exp` for matrices, i.e., one may use it in the form `exp(A, ...)`.

expand — Expand matrix components

```
expand(A)
```

This method only exists if R implements the method `"expand"`, or if R has the axiom `Ax::systemRep` (in this case, the system function `expand` is used).

This method overloads the system function `expand` for matrices, i.e., one may use it in the form `expand(A)`.

factor — Scalar-matrix factorization

```
factor(A)
```

The factor s is the gcd of all components of the matrix A. Hence, this method only exists if R is of category `Cat::GcdDomain`.

This method overloads the system function `factor` for matrices, i.e., one may use it in the form `factor(A)`.

`float` — Floating-point approximation of the matrix components

```
float(A)
```

This method only exists if `R` implements the method `"float"`.

Note: Usually the floating-point approximations are not elements of `R`! For example, `Dom::Integer` implements such a method, but the floating-point approximation of an integer cannot be re-converted to an integer.

This method checks whether the resulting matrix can be converted to the domain type of `A` only if `testargs()` returns `TRUE` (e.g., if one calls this method from the interactive level of MuPAD).

Otherwise, one has to take care that the matrix returned is compatible with its component ring.

`fourier` — Fourier transform of the matrix components

```
fourier(A, t, s)
```

This method overloads the function `fourier` for matrices.

`gaussElim` — Gaussian elimination for matrices

```
gaussElim(A, <ColumnElimination>)
```

With the option `ColumnElimination`, the matrix A is reduced to a lower triangular echelon form via elementary *column* operations (without `ColumnElimination`, the Gauss algorithm uses elementray *row* operations to obtain the upper echelon form). The following relation holds: `transpose(gaussElim(A, ColumnElimination)[1]) = gaussElim(transpose(A))[1]`. With `ColumnElimination`, the last entry of the returned list is the set of characteristic *column* indices of A.

For very large $m \times n$ matrices A with m much greater n, the column elimination is faster than the row elimination.

If the matrix is not square, i.e., the determinant of `A` is not defined, then the third entry of the returned list is the value `FAIL`.

This method only exists if the component ring `R` is an integral domain, i.e., a domain of category `Cat::IntegralDomain`.

If `R` has the method `"pivotSize"`, the pivot element of smallest size is chosen at every pivoting step, whereby `pivotSize` must return a positive integer representing the "size" of an element.

If no such method is defined, Gaussian elimination without a pivot strategy is applied to `A`.

If `R` has the axiom `Ax::efficientOperation("_invert")` and is of category `Cat::Field`, ordinary Gaussian elimination is used. Otherwise, fraction-free elimination is performed on `A`.

If `R` implements the method `"normal"`, it is used to simplify subsequent computations of the Gaussian elimination process.

Note that if `R` does not implement the method `"normal"`, but the elements of `R` are represented by kernel domains, i.e., `R` has the axiom `Ax::systemRep`, the system function `normal` is used instead.

`identity` — Identity matrix

```
identity(n)
```

This method only exists if the component ring `R` is of category `Cat::Ring`, i.e., a ring with unit.

`int` — Integration of matrix components

```
int(A, …)
```

This method only exists if `R` implements the method `"int"`.

This method overloads the system function `int` for matrices, i.e., one may use it in the form `int(A, ...)`.

`ifourier` — Inverse Fourier transform of the matrix components

```
ifourier(A, s, t)
```

This method overloads the function `ifourier` for matrices.

`ilaplace` — Inverse Laplace transform of the matrix components

```
ilaplace(A, s, t)
```

This method overloads the function `ilaplace` for matrices.

`iszero` — Test for zero matrices

```
iszero(A)
```

Note that there may exist more than one representation of the zero matrix of a given dimension if `R` does not have `Ax::canonicalRep`.

If `R` implements the method `"normal"`, it is used to simplify the components of `A` for the zero-test.

Note that if `R` does not implement such a method, but the elements of `R` are represented by kernel domains, i.e., `R` has the axiom `Ax::systemRep`, the system function `normal` is used instead.

This method overloads the system function `iszero` for matrices, i.e., one may use it in the form `iszero(A)`.

`laplace` — Laplace transform of the matrix components

```
laplace(A, t, s)
```

This method overloads the function `laplace` for matrices.

`matdim` — Matrix dimension

```
matdim(A)
```

`norm` — Norm of matrices and vectors

```
norm(A, Infinity)

norm(A, Maximum)

norm(v, Infinity)

norm(v, Maximum)
```

```
norm(A, Frobenius)
```

```
norm(A, 1)
```

```
norm(v, Euclidean)
```

```
norm(v, k)
```

`norm(A, Maximum)` computes the maximum norm of the matrix `A`, which is the maximum row sum (the row sum is the sum of norms of each component in a row).

If the domain `R` does not implement the methods `"max"` and `"norm"`, `FAIL` is returned.

Using `norm(v, Infinity)` for a vector `v` the maximum norm of all elements is returned.

If the domain `R` does not implement the methods `"max"` and `"norm"`, `FAIL` is returned.

Using `norm(v, Maximum)` for a vector `v` the maximum norm of all elements is returned.

If the domain `R` does not implement the methods `"max"` and `"norm"`, `FAIL` is returned.

`norm(A, Frobenius)` computes the Frobenius norm of `A`, which is the square root of the sum of the squares of the norms of each component.

If the result is no longer an element of the domain `R`, or if `R` does not implement the method `"norm"`, `FAIL` is returned.

`norm(A, 1)` computes the 1-norm of the matrix `A`, which is the maximum sum of the norms of the elements of each column. If `R` does not implement the methods `"max"` and `"norm"`, `FAIL` is returned.

`norm(v, Euclidean)` computes the Euclidean norm (2-norm) of the vector `v`, which is defined to be the square root of the sum of the norms of the elements of `v` raised to the square.

`FAIL` is returned if the result is no longer an element of the domain `R`. The function `linalg::scalarProduct` is used to compute the Euclidean norm of the vector `v`.

If `R` does not implement the method `"norm"`, `FAIL` is returned.

`norm(v, k)` computes the k-norm of the vector `v`, which is defined to be the kth root of the sum of the norms of the elements of `v` raised to the kth power.

FAIL is returned if the result is no longer an element of the domain R. For $k = 2$, the function linalg::scalarProduct is used to compute the 2-norm of v.

If R does not implement the method "norm", FAIL is returned.

The method norm overloads the function norm for matrices, i.e., one may use it in the form norm(A k), where k is either Infinity, Frobenius, or a positive integer. The default value of k is Infinity.

normal — Simplification of matrix components

```
normal(A)
```

If R does not implement the method "normal", but the elements of R are represented by kernel domains, i.e., R has the axiom Ax::systemRep, the system function normal is applied to the components of A. Otherwise normal(A) returns A without any changes.

This method overloads the system function normal for matrices, i.e., one may use it in the form normal(A).

nonZeros — Number of non-zero components of a matrix

```
nonZeros(A)
```

nonZeroes — Number of non-zero components of a matrix

```
nonZeroes(A)
```

nonZeroOperands — Return a sequence of all non-zero operands

```
nonZeroOperands(A)
```

This method is useful for retrieving information on the non-zero entries. For example, to find out the types of the entries in the matrix, one should not consider all operands op(A), because this would also involve the zero entries. For large matrices with few entries, it is much more efficient to use this method to extract the entries.

random — Random matrix generation

```
random()

random(g)
```

```
random(m, n)

random(m, n, g)

random(m, n, p)

random(m, n, p, g)
```

The dimension of the matrix is also chosen randomly. The matrix size is limited by the values `"randomDimen"` (see "Entries" above). To change the value of the entry `"randomDimen"`, one must first unprotect the domain `Dom` (see `unprotect` for details).

When calling the `"random"` method with one parameter `g`, this parameter is regarded as a random generator. The matrix entries are created by the calls `g()` which must return elements of the coefficient ring `R`.

The dimension of the matrix is chosen randomly as above.

When calling the `"random"` method with two positive integers `m` and `n`, a random matrix with `m` rows and `n` columns is created. Its elements are generated by the method `"random"` of the component ring `R`. If `R::random` does not exist, `FAIL` is returned.

`random(m,n,g)` creates a matrix with `m` rows and `n` columns. The third parameter `g` is regarded as a random generator. The matrix entries are created by the calls `g()` which must return elements of the coefficient ring `R`.

When calling the `"random"` method with positive integers `m`, `n` and a nonnegative integer `p`, a sparse matrix with `m` rows, `n` columns and at most `p` non-zero entries is created. These entries are generated by the function `"random"` of the component ring `R`. If `R::random` does not exist, `FAIL` is returned.

The integer `p` must satisfy $0 \le p \le m\,n$.

When calling the `"random"` method with four parameters, a sparse matrix with `m` rows, `n` columns and at most `p` non-zero entries is created. The fourth parameter `g` is regarded as a random generator. The matrix entries are created by the calls `g()` which must return elements of the coefficient ring `R`.

The integer `p` must satisfy $0 \le p \le m\,n$.

`sin` — Sine of a matrix

```
sin(A)
```

If A is not square, an error message is issued. If the component domain of A does not allow the computation of `sin(elem)` for an arbitrary element `elem` of the component ring, `FAIL` is returned.

This method uses the function `numeric::expMatrix` for a floating-point approximation of the exponential of A if A is defined over the domain `Dom::Float`.

If some eigenvalues of A do not exist in R or cannot be computed, then `FAIL` is returned.

In the symbolic case the functions `exp` and `linalg::jordanForm` are called. The latter may not be able to compute the Jordan form of A. In this case `FAIL` is returned.

This method only exists if R is a domain of category `Cat::Field`.

This method overloads the function `sin` for matrices, i.e., one may use it in the form `sin(A)`.

`sqrt` — Square root of a matrix

```
sqrt(A, <sqrtfunc>)
```

Returned is a matrix B with $B^2 = A$ such that the eigenvalues of B are the square roots of the eigenvalues of A or `FAIL` if the square root of the matrix does not exist. For computing the square roots of the eigenvalues a function satisfying $sqrtfunc(a)^2 = a$ for every element a of the coefficient ring of A can be given as optional second argument.

For details we refer to the help page of the function `linalg::sqrtMatrix`.

`testeq` — Testing for equality of two matrices

```
testeq(A, B)
```

`tr` — Trace of a square matrix

```
tr(A)
```

If A is not square, an error message is issued.

`transpose` — Transpose of a matrix

```
transpose(A)
```

Access Methods

`_concat` — Horizontal concatenation of matrices

```
_concat(A, B, …)
```

This method overloads the system function `_concat` for matrices, i.e., one may use it in the form `A . B . ...`, or in functional notation: `_concat(A, B, ...)`.

`_index` — Matrix indexing

```
_index(A, i, j)

_index(A, r1 .. r2, c1 .. c2)

_index(v, i)

_index(v, i1 .. i2)
```

If `i` and `j` are not integers, the call of this method returns in its symbolic form (of type `"_index"`) with evaluated arguments.

Otherwise an error message is given, if `i` and `j` are not valid row and column indices, respectively.

Note: Note that this method uses the system function `context` to evaluate the entry in the context of the calling environment.

`_index(A,r1..r2,c1..c2)` returns the submatrix of `A` created by the rows of `A` with indices from `r1` to `r2` and the columns of `A` with indices from `c1` to `c2`.

This method returns the *i*th entry of the vector `v`.

An error message is issued if `v` is not a vector.

If `i` is not an integer, the call of `_index(v,i)` returns in its symbolic form (of type `"_index"`) with evaluated arguments.

Otherwise an error message is given, if `i` is less than one or greater than the dimension of `v`.

Note: Note that this method uses the system function `context` to evaluate the entry in the context of the calling environment.

`_index(v,i1..i2)` returns the subvector of `v`, formed by the entries with index `i1` to `i2`. See also the method `"op"`.

An error message is issued if `v` is not a vector.

`_index` overloads the system function `_index` for matrices, i.e., one may use it in the form `A[i, j]`, `A[r1..r2, c1..c2]`, `v[i]` and `v[i1..i2]`, respectively, or in functional notation: `_index(A, ...)`.

`addCol` — Addition of a multiple of one column to the multiple of another column

```
addCol(A, i, j, f, <g>)
```

`i` and `j` must be positive integers smaller than or equal to the number of columns of the matrix `A`.

If `f` and `g` are not elements of the coefficient domain `R` of the matrix `A` and cannot be converted to `R`, `FAIL` is returned.

`addRow` — Addition of a multiple of one row to the multiple of another row

```
addRow(A, i, j, f, g)
```

`i` and `j` must be positive integers smaller than or equal to the number of rows of the matrix `A`.

If `f` and `g` are not elements of the coefficient domain `R` of the matrix `A` and cannot be converted to `R`, `FAIL` is returned.

`concatMatrix` — Horizontal concatenation of matrices

```
concatMatrix(A, B, …)
```

`col` — Extracting a column of a matrix

```
col(A, c)
```

An error message is issued if `c` is less than one or greater than the number of columns of `A`.

`delCol` — Deleting a column of a matrix

```
delCol(A, c)
```

`NIL` is returned if `A` consists of only one column.

An error message is issued if `c` is less than one or greater than the number of columns of `A`.

`delRow` — Deleting a row of a matrix

```
delRow(A, r)
```

`NIL` is returned if `A` consists of only one row.

An error message is issued if `r` is less than one or greater than the number of rows of `A`.

`evalp` — Evaluating matrices of polynomials at a certain point

```
evalp(A, x = a, …)
```

This method is only defined if `R` is a polynomial ring of category `Cat::Polynomial`.

This method overloads the system function `evalp` for matrices, i.e., one may use it in the form `evalp(A, x = a)`.

`length` — Length of a matrix

```
length(A)
```

This method overloads the system function `length` for matrices, i.e., one may use it in the form `length(A)`.

`map` — Apply a function to matrix components

```
map(A, f, <p1, p2, …>)
```

Note: Note that values returned by `f` are converted to elements of the domain `R` only if `testargs()` returns `TRUE` (i.e., if one calls this method from the interactive level of MuPAD).

If `testargs()` returns `FALSE`, one must guarantee that `f` returns elements of the domain type `R`. Otherwise, the resulting matrix will have components which are not elements of the component ring `R`!

Note: If the function `f` does not map the zero element of the component ring to the zero element, a sparse matrix will change into a dense matrix. This may lead to memory problems when dealing with very large (sparse) matrices.

Note that there is the method `"mapNonZeroes"` which maps a function to the non-zero entries of the matrix only.

This method overloads the system function `map` for matrices, i.e., one may use it in the form `map(A, f p1 , p2 , , …)`.

`mapNonZeroes` — Apply a function to the non-zero components of a (sparse) matrix

```
mapNonZeroes(A, f, <p1, p2, …>)
```

`multCol` — Multiplication of one column by a scalar factor

```
multCol(A, i, f)
```

`i` must be a positive integer smaller than or equal to the number of columns of the matrix `A`.

If `f` is not an element of the coefficient domain `R` of the matrix `A` and cannot be converted to `R`, `FAIL` is returned.

`multRow` — Multiplication of one row by a scalar factor

```
multRow(A, i, f)
```

`i` must be a positive integer smaller than or equal to the number of rows of the matrix `A`.

If `f` is not an element of the coefficient domain `R` of the matrix `A` and cannot be converted to `R`, `FAIL` is returned.

`nops` — Number of components of a matrix

```
nops(A)
```

This method overloads the system function `nops` for matrices, i.e., one may use it in the form `nops(A)`.

`op` — Component of a matrix

```
op(A, i)
```

```
op(A)
```

This method returns an expression sequence of all components of A.

See also the method "_index".

This method overloads the system function op for matrices, i.e., one may use it in the form op(A, i) and op(A), respectively.

row — Extracting a row from a matrix

```
row(A, r)
```

An error message is issued if r is less than one or greater than the number of rows of A.

setCol — Replacing a column of a matrix

```
setCol(A, c, v)
```

An error message is issued if c is less than one or greater than the number of rows of A.

setRow — Replacing a row of a matrix

```
setRow(A, r, v)
```

An error message is issued if r is less than one or greater than the number of rows of A.

stackMatrix — Vertical concatenation of matrices

```
stackMatrix(A, B, …)
```

An error message is issued if the given matrices do not have the same number of columns.

subs — Substitution of matrix components

```
subs(A, …)
```

Note: Note that the function values are converted to elements of the domain R only if testargs() returns TRUE (e.g., if one calls this method from the interactive level of MuPAD).

If `testargs()` returns `FALSE`, one must guarantee that `f` returns elements of the domain type `R`. Otherwise, the resulting matrix, which is of domain type `Dom::Matrix(R)`, would have components which are not elements of the domain `R`!

This method overloads the system function `subs` for matrices, i.e., one may use it in the form `subs(A, ...)`.

subsex — Extended substitution of matrix components

```
subsex(A, …)
```

Note: Note that the results of the substitutions are converted to elements of the domain `R` only if `testargs()` returns `TRUE` (e.g., if one calls this method from the interactive level of MuPAD).

If `testargs()` returns `FALSE`, one must guarantee that the results of the substitutions are of the domain type `R`, otherwise the resulting matrix, which is of domain type `Dom::Matrix(R)`, would have components which are not elements of the domain `R`!

This method overloads the system function `subsex` for matrices, i.e., one may use it in the form `subsex(A, ...)`.

subsop — Operand substitution of matrix components

```
subsop(A, i = x, …)
```

Note: Note that `x` is converted to the domain `R` only if `testargs()` returns `TRUE` (e.g., if one calls this method from the interactive level of MuPAD).

If `testargs()` returns `FALSE`, `x` must be an element of `R`, otherwise the resulting matrix, which is of domain type `Dom::Matrix(R)`, would have components which are not elements of the domain `R`!

See also the method `"set_index"`.

This method overloads the system function `subsop` for matrices, i.e., one may use it in the form `subsop(A, ...)`.

swapCol — Swapping matrix columns

```
swapCol(A, c1, c2)
```

```
swapCol(A, c1, c2, r1 .. r2)
```

An error message is issued if one of the column indices is less than one or greater than the number of columns of `A`.

`swapCol(A,c1,c2,r1..r2)` swaps the column with index `c1` and the column with index `c2` of `A`, but by taking only those column components which lie in the rows with indices `r1` to `r2`.

An error message is issued if one of the column indices is less than one or greater than the number of columns of `A`, or if one of the row indices is less than one or greater than the number of rows of `A`.

swapRow — Swapping matrix rows

```
swapRow(A, r1, r2)
```

```
swapRow(A, r1, r2, c1 .. c2)
```

An error message is issued if one of the row indices is less than one or greater than the number of rows of `A`.

`swapCol(A,r1,r2,c1..c2)` swaps the row with index `r1` and the row with index `r2` of `A`, but by taking only those row components which lie in the columns with indices `c1` to `c2`.

An error message is issued if one of the row indices is less than one or greater than the number of rows of `A`, or if one of the column indices is less than one or greater than the number of columns of `A`.

set_index — Setting matrix components

```
set_index(A, i, j, x)
```

```
set_index(v, i, x)
```

Note: Note that `x` is converted into an element of the domain `R` only if `testargs` returns `TRUE` and `i` and `j` are integers (e.g., if one calls this method from the interactive level of

MuPAD). If x is a matrix of the same type as A or can be converted into a matrix of the same type as A and the indices i or j are ranges corresponding to a submatrix of A, then x replaces the corresponding submatrix in A.

Otherwise one has to take care that x is of domain type R.

See also the method "subsop".

set_index(v,i,x) replaces the *i*th entry of the vector v by x.

set_index on vectors overloads the function set_index for matrices, i.e., one may use it in the form A[i, j] := x and v[i] := x, respectively, or in functional notation: A := set_index(A, i, j, x) or v := set_index(v, i, x).

zip — Combine matrices component-wise

zip(A, B, f, <p1, p2, …>)

The row number of the matrix returned is the minimum of the row numbers of A and B. Its column number is the minimum of the column numbers of A and B.

Note: Note that the values returned by f are converted to elements of the domain R only if testargs() returns TRUE (i.e., if one calls this method from the interactive level of MuPAD).

If testargs() returns FALSE, one must guarantee that f returns elements of the domain type R. Otherwise, the resulting matrix will have components which are not elements of the component ring R!

This method overloads the system function zip for matrices, i.e., one may use it in the form zip(A, B, f p1 , p2 , , …).

Conversion Methods

convert — Conversion to a matrix

convert(x)

FAIL is returned if the conversion fails.

x may either be an array, a matrix, or a list (of sublists, see the parameter ListOfRows in "Creating Elements" above). Their entries must then be convertible into elements of the domain R.

convert_to — Matrix conversion

```
convert_to(A, T)
```

T may either be DOM_ARRAY, DOM_LIST, or a domain constructed by Dom::Matrix or Dom::SquareMatrix. The elements of A must be convertible into elements of the domain R.

Use the function expr to convert A into an object of a kernel domain (see below).

create — Defining matrices without component conversions

```
create(x, …)
```

This method works more efficiently than if one creates matrices by calling the method "new" of the domain, because it avoids any conversion of the components. One must guarantee that the components have the correct domain type, otherwise run-time errors can be caused.

expr — Conversion of a matrix to an object of a kernel domain

```
expr(A)
```

The result is an array representing the matrix A where each entry is an object of a kernel domain.

This method overloads the system function expr for matrices, i.e., one may use it in the form expr(A).

expr2text — Conversion of a mtrix to a string

```
expr2text(A)
```

This method overloads the system function expr2text for matrices, i.e., one may use it in the form expr2text(A).

TeX — TeX formatting of a matrix

```
TeX(A)
```

Note that in the case of very large matrices the output will not be useful. For printing large matrices use the function "doprint" to obtain a sparse matrix output displaying all non-zero entries. Alternatively, use the matrix slot "setPrintMaxSize" to set the maximal size of matrices that will be printed like "dense" arrays with zero entries displayed as the integer 0.

The method "TeX" of the component ring R is used to get the TeX-representation of each component of A.

This method is used by the function generate::TeX.

Technical Methods

assignElements — Multiple assignment to matrices

```
assignElements(A, …)
```

The assigned components must have the domain type R, an implicit conversion of the components into elements of domain type R is not performed.

This method overloads the system function assignElements for matrices, i.e., one may use it in the form assignElements(A, ...).

mkSparse — Conversion of an array or a list of lists to a sparse structure

```
mkSparse(Array)
```

```
mkSparse(List)
```

```
mkSparse(r, c, List)
```

The 'sparse structure' s is a list of c univariate polynomials that is used to store the non-trivial elements of the columns of matrices.

mkSparse(List) tries to convert the list List into a a sparse structure. The result is either FAIL if this is not possible, or the list [s, [r, c]], where the positive integers r and c are the dimension of the corresponding matrix. The 'sparse structure' s is a list

of univariate polynomials that is used to store the non-trivial elements of the columns of matrices.

See the parameters `List` and `ListOfRows` in “Creating Elements” above for admissible formats of `List`.

The matrix is regarded as a column or a row vector, if `r` or `c` is equal to one. T

`mkSparse(r,c,List)` tries to convert the list `List` into a sparse structure representing a matrix of dimension `r` times `c`.

The result is either `FAIL` if this is not possible, or the list `[s, [r, c]]`. The 'sparse structure' `s` is a list of univariate polynomials that is used to store the non-trivial elements of the columns of matrices.

The matrix is regarded as a column or a row vector, if `r` or `c` is equal to one. T

`print` — Printing matrices

```
print(A)
```

Note: Note that it will not be useful to print very large sparse matrices with lots of zero coefficients in this way – printing such matrices can be done by using the function `"doprint"`.

Use the matrix slot `"setPrintMaxSize"` to set the maximal size of matrices that will be printed like “dense” arrays with zero entries displayed as the integer 0.

`doprint` — Printing large sparse matrices

```
doprint(A)
```

`setPrintMaxSize` — Set the maximal size of matrices that will be printed like “dense” arrays

```
setPrintMaxSize(printMaxSize)
```

The value of the parameter `printMaxSize` may also be `infinity`. In this case, matrices of arbitrary size are printed like “dense” arrays.

This method returns the previous value of `printMaxSize`.

The default value is `printMaxSize` = 500.

`unapply` — Create a procedure from a matrix

```
unapply(A, <x, …>)
```

This method overloads the system function `fp::unapply` for matrices, i.e., one may use it in the form `fp::unapply(A)`.

See Also

MuPAD Domains
`Dom::DenseMatrix` | `Dom::DenseMatrix` | `Dom::MatrixGroup` | `Dom::SquareMatrix`

MuPAD Functions
`densematrix` | `matrix`

Dom::MatrixGroup

The Abelian group of matrices

Syntax

Domain Creation

```
Dom::MatrixGroup(m, n, <R>)
```

Element Creation

```
Dom::MatrixGroup(m, n, R)(Array)

Dom::MatrixGroup(m, n, R)(Matrix)

Dom::MatrixGroup(m, n, R)(<m, n>)

Dom::MatrixGroup(m, n, R)(<m, n>, List)

Dom::MatrixGroup(m, n, R)(<m, n>, ListOfRows)

Dom::MatrixGroup(m, n, R)(<m, n>, f)

Dom::MatrixGroup(m, n, R)(<m, n>, List, <Diagonal>)

Dom::MatrixGroup(m, n, R)(<m, n>, g, <Diagonal>)

Dom::MatrixGroup(m, n, R)(<m, n>, List, <Banded>)
```

Description

Domain Creation

`Dom::MatrixGroup(m, n, R)` creates a domain which represents the Abelian group of $m \times n$ matrices over the component ring `R`, i.e., it is a domain of category `Cat::AbelianGroup`.

The domain `Dom::ExpressionField()` is used as the component ring for the matrices if the optional parameter `R` is not given.

For matrices of a domain created by `Dom::MatrixGroup(m, n, R)`, matrix arithmetic is implemented by overloading the standard arithmetical operators `+, -, *, /` and `^`. All functions of the linalg package dealing with matrices can be applied.

`Dom::MatrixGroup(m, n, R)` has the domain `Dom::Matrix(R)` as its super domain, i.e., it inherits each method which is defined by `Dom::Matrix(R)` and not re-implemented by `Dom::MatrixGroup(m, n, R)`.

Methods described below are implemented by `Dom::MatrixGroup`.

The domain `Dom::Matrix(R)` represents matrices over `R` of arbitrary size, and it therefore does not have any algebraic structure (except of being a *set* of matrices).

The domain `Dom::SquareMatrix(n, R)` represents the *ring* of $n \times n$ matrices over R.

Element Creation

`Dom::MatrixGroup(m, n, R)(Array)` and `Dom::MatrixGroup(m, n, R)(Matrix)` create a new matrix formed by the entries of `Array` and `Matrix`, respectively.

The components of `Array` and `Matrix`, respectively, are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

The call `Dom::MatrixGroup(m, n, R)( m , n )` returns the $m \times n$ zero matrix. Note that the $m \times n$ zero matrix can also be found in the entry `"zero"` (see below).

`Dom::MatrixGroup(m, n, R)( m , n List)` creates an $m \times n$ matrix with components taken from the list `List`.

This call is only allowed for $m \times 1$ or $1 \times n$ matrices, i.e., if either `m` or `n` is equal to one.

If the list has too few entries, the remaining components of the matrix are set to zero.

The entries of the list are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

`Dom::MatrixGroup(m, n, R)( m , n ListOfRows)` creates an $m \times n$ matrix with components taken from the nested list `ListOfRows`. Each inner list corresponds to a row of the matrix.

If an inner list has less than `n` entries, the remaining components in the corresponding row of the matrix are set to zero. If there are less than `m` inner lists, the remaining lower rows of the matrix are filled with zeroes.

The entries of the inner lists are coerced into elements of the domain `R`. An error message is issued if one of these conversions fails.

It might be a good idea first to create a two-dimensional array from that list before calling `Dom::MatrixGroup(m, n, R)`. This is due to the fact that creating a matrix from an array is the fastest way one can achieve. However, in this case the sublists must have the same number of elements.

`Dom::MatrixGroup(m, n, R)( m , n f)` returns the matrix whose (i, j)th component is the value of the function call `f(i, j)`. The row index i ranges from 1 to `m` and the column index j from 1 to `n`.

The function values are coerced into elements of the domain `R`. An error message is issued if one of these conversions fails.

Superdomain

`Dom::Matrix(R)`

Axioms

If R has `Ax::canonicalRep`, then `Ax::canonicalRep`.

Categories

`Cat::Matrix(R)`, `Cat::AbelianGroup`

Examples

Example 1

A lot of examples can be found on the help page of the domain constructor `Dom::Matrix`, and most of them are also examples for working with domains created

by `Dom::MatrixGroup`. This example only highlights some differences with respect to working with matrices of the domain `Dom::Matrix(R)`.

The following command defines the abelian group of 3 ×4 matrices over the rationals:

```
MatGQ := Dom::MatrixGroup(3, 4, Dom::Rational)
```

Dom::MatrixGroup(3, 4, Dom::Rational)

```
MatGQ::hasProp(Cat::AbelianGroup), MatGQ::hasProp(Cat::Ring)
```

TRUE, FALSE

`MatGQ` is a commutative group with respect to the addition of matrices. The unit of this group is the 3 ×4 zero matrix:

```
MatGQ::zero
```

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Note that some operations defined by the domain `MatGQ` return matrices which are no longer elements of the matrix group. They return matrices of the domain `Dom::Matrix(Dom::Rational)`, the super-domain of `MatGQ`.

For example, if we define the matrix:

```
A := MatGQ([[1, 2, 1, 2], [-5, 3], [2, 1/3, 0, 1]])
```

$$\begin{pmatrix} 1 & 2 & 1 & 2 \\ -5 & 3 & 0 & 0 \\ 2 & \frac{1}{3} & 0 & 1 \end{pmatrix}$$

and delete its third column, we get the matrix:

```
MatGQ::delCol(A, 3)
```

$$\begin{pmatrix} 1 & 2 & 2 \\ -5 & 3 & 0 \\ 2 & \frac{1}{3} & 1 \end{pmatrix}$$

which is of the domain type:

```
domtype(%)
```

$$\text{Dom::Matrix(Dom::Rational)}$$

As another example we create the 3×3 identity matrix using the method `"identity"` of our domain:

```
E3 := MatGQ::identity(3)
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

This is also a matrix of the domain `Dom::Matrix(Dom::Rational)`:

```
domtype(E3)
```

$$\text{Dom::Matrix(Dom::Rational)}$$

If we concatenate `E3` to the right of the matrix `A` defined above, we get the 3 ×7 matrix:

```
B := A . E3
```

$$\begin{pmatrix} 1 & 2 & 1 & 2 & 1 & 0 & 0 \\ -5 & 3 & 0 & 0 & 0 & 1 & 0 \\ 2 & \frac{1}{3} & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

which is of the domain type `Dom::Matrix(Dom::Rational)`:

```
domtype(B)
```

$$\text{Dom::Matrix(Dom::Rational)}$$

Example 2

We can convert a matrix from a domain created with `Dom::MatrixGroup` into or from another matrix domain, as shown next:

```
MatGR := Dom::MatrixGroup(2, 3, Dom::Real):
MatC := Dom::Matrix(Dom::Complex):

A := MatGR((i, j) -> i*j)
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \end{pmatrix}$$

To convert `A` into a matrix of the domain `MatC`, enter:

```
coerce(A, MatC)
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \end{pmatrix}$$

```
domtype(%)
```

$$\text{Dom::Matrix(Dom::Complex)}$$

The conversion is done component-wise. For example, we define the following matrix:

```
B := MatC([[0, 1, 0], [exp(I), 0, 1]])
```

$$\begin{pmatrix} 0 & 1 & 0 \\ e^{i} & 0 & 1 \end{pmatrix}$$

The matrix B has one complex component and therefore cannot be converted into the domain `MatGR`:

```
coerce(B, MatGR)
```

```
      FAIL
```

Note: The system function `coerce` uses the methods `"convert"` and `"convert_to"` implemented by any domain created with `Dom::MatrixGroup` and `Dom::Matrix`.

Parameters

`m, n`

Positive integers (matrix dimension)

`R`

A commutative ring, i.e., a domain of category `Cat::CommutativeRing`; the default is `Dom::ExpressionField()`

`Array`

An $m \times n$ array

`Matrix`

An $m \times n$ matrix, i.e., an element of a domain of category `Cat::Matrix`

`List`

A list of matrix components

`ListOfRows`

A list of at most `m` rows; each row is a list of at most `n` matrix components

`f`

A function or a functional expression with two parameters (the row and column index)

`g`

A function or a functional expression with one parameter (the row index)

Options

`Diagonal`

Create a diagonal matrix

With the option `Diagonal`, diagonal matrices can be created with diagonal elements taken from a list, or computed by a function.

`Dom::MatrixGroup(m, n, R)( m , n List, Diagonal)` creates the $m \times n$ diagonal matrix whose diagonal elements are the entries of `List`.

`List` must have at most $min(m, n)$ entries. If it has fewer elements, then the remaining diagonal elements are set to zero.

The entries of `List` are coerced into elements of the domain `R`. An error message is issued if one of these conversions fails.

`Dom::MatrixGroup(m, n, R)( m , n g, Diagonal)` returns the matrix whose ith diagonal element is `g(i, i)`, where the index i runs from 1 to $min(m, n)$.

The function values are coerced into elements of the domain `R`. An error message is issued if one of these conversions fails.

`Banded`

Create a banded Toeplitz matrix

With the option `Banded`, banded matrices can be created.

A *banded matrix* has all entries zero outside the main diagonal and some of the adjacent sub- and superdiagonals.

`Dom::MatrixGroup(m, n, R)( m , n List, Banded)` creates an $m \times n$ banded Toeplitz matrix with the elements of `List` as entries. The number of entries of `List` must be odd, say $2h + 1$, and must not exceed `n`. The resulting matrix has bandwidth at most $2h + 1$.

All elements of the main diagonal of the created matrix are initialized with the middle element of `List`. All elements of the *i*th subdiagonal are initialized with the (*h* + 1 - *i*)th element of `List`. All elements of the *i*th superdiagonal are initialized with the (*h* + 1 + *i*)th element of `List`. All entries on the remaining sub- and superdiagonals are set to zero.

The entries of `List` are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

Entries

"one"	is only defined if `m` is equal to `n`; in that case it defines the $n \times n$ identity matrix.
"randomDimen"	is set to `[m, n]`.
"zero"	is the $m \times n$ zero matrix.

Methods

Mathematical Methods

`evalp` — Evaluating matrices of polynomials at a certain point

```
evalp(A, x = a, …)
```

This method is only defined if `R` is a polynomial ring of category `Cat::Polynomial`.

This method overloads the function `evalp` for matrices, i.e., one may use it in the form `evalp(A, x = a)`.

`identity` — Identity matrix

```
identity(k)
```

Note: The matrix returned is of the domain `Dom::Matrix(R)`, if $m \neq n$ or if $k \neq n$.

`matdim` — Matrix dimension

```
matdim(A)
```

`random` — Random matrix generation

```
random()
```

The components of the random matrix are randomly generated with the method `"random"` of the component ring `R`.

Access Methods

`_concat` — Horizontally concatenation of matrices

```
_concat(A, B, …)
```

An error message is issued if the given matrices do not have the same number of rows.

Note: The returned matrix is of the domain `Dom::Matrix(R)`.

This method overloads the function `_concat` for matrices, i.e., one may use it in the form `A . B . ...`, or in functional notation: `_concat(A, B, ...)`.

`_index` — Matrix indexing

```
_index(A, i, j)
```

```
_index(A, r1 .. r2, c1 .. c2)
```

```
_index(A, i)
```

```
_index(A, i1 .. i2)
```

If `i` and `j` are not integers, then the call of this method returns in its symbolic form (of type `"_index"`) with evaluated arguments.

Otherwise an error message is given, if `i` and `j` are not valid row and column indices, respectively.

Note: Note that the system function `context` is used to evaluate the entry in the context of the calling environment.

This method overloads the function `_index` for matrices, i.e., one may use it in the form `A[i, j]` or in functional notation: `_index(A, i, j)`.

Returns the submatrix of `A`, created by the rows of `A` with indices from `r1` to `r2` and the columns of `A` with indices from `c1` to `c2`.

Note: The submatrix is of the domain `Dom::Matrix(R)`.

This method returns the *i*th entry of `A`.

If `i` is not an integer, then the call of this method returns in its symbolic form (of type `"_index"`) with evaluated arguments.

Otherwise an error message is given, if `i` is less than one or greater than the dimension of `v`.

This call is only allowed for $1 \times n$ or $m \times 1$ matrices, i.e., either `m` or `n` must be equal to one. Otherwise an error message is issued.

Note: Note that the system function `context` is used to evaluate the entry in the context of the calling environment.

This method returns the subvector of `A`, formed by the entries with index `i1` to `i2` (see also the method `"op"`).

This call is only allowed for $1 \times n$ or $m \times 1$ matrices, i.e., either `m` or `n` must be equal to one. Otherwise an error message is issued.

This method overloads the function `_index` for matrices, i.e., one may use it in the form `A[i,j]`, `A[r1..r2,c1..c2]`, `A[i]` or `A[i1..i2]`, respectively, or in functional notation: `_index(A, ...)`.

`concatMatrix` — Horizontally concatenation of matrices

```
concatMatrix(A, B, …)
```

col — Extracting a column

```
col(A, c)
```

An error message is issued if `c` is less than one or greater than `n`.

delCol — Deleting a column

```
delCol(A, c)
```

`NIL` is returned if `A` only consists of one column.

Note: The returned matrix is of the domain `Dom::Matrix(R)`.

An error message is issued if `c` is less than one or greater than `n`.

delRow — Deleting a row

```
delRow(A, r)
```

`NIL` is returned if `A` only consists of one row.

Note: The returned matrix is of the domain `Dom::Matrix(R)`.

An error message is issued if `r` is less than one or greater than `m`.

row — Extracting a row

```
row(A, r)
```

An error message is issued if `r` is less than one or greater than `m`.

stackMatrix — Concatenating of matrices vertically

```
stackMatrix(A, B, …)
```

An error message is issued if the given matrices do not have the same number of columns.

Note: The matrix returned is of the domain `Dom::Matrix(R)`.

Conversion Methods

`convert` — Conversion into a matrix

```
convert(x)
```

`FAIL` is returned if the conversion fails.

`x` may either be an $m \times n$ array, or an $m \times n$ matrix of category `Cat::Matrix`.

`x` can also be a list. See the parameter `List` and `ListOfRows` in "Creating Elements" above for admissible values of `x`.

The entries of `x` must be convertable into elements of the domain `R`, otherwise `FAIL` is returned.

See Also

MuPAD Domains
`Dom::Matrix` | `Dom::SquareMatrix`

Dom::MonomOrdering

Monomial orderings

Syntax

```
Dom::MonomOrdering(Lex(n))
```

```
Dom::MonomOrdering(RevLex(n))
```

```
Dom::MonomOrdering(DegLex(n))
```

```
Dom::MonomOrdering(DegRevLex(n))
```

```
Dom::MonomOrdering(DegInvLex(n))
```

```
Dom::MonomOrdering(WeightedLex(w1, …,wn))
```

```
Dom::MonomOrdering(WeightedDegLex(w1, …,wn))
```

```
Dom::MonomOrdering(WeightedDegRevLex(w1, …,wn))
```

```
Dom::MonomOrdering(WeightedRevLex(w1, …,wn))
```

```
Dom::MonomOrdering(Block(o1, …))
```

```
Dom::MonomOrdering(Matrix(params))
```

Description

`Dom::MonomOrdering` represents the set of all possible monomial orderings. A monomial ordering is a well-ordering of the set of all k-tuples of nonnegative integers for some k.

In MuPAD, a monomial ordering is implemented as a function that, when applied to two lists of nonnegative integers, returns `-1`, `0`, or `1` if the first list is respectively smaller than, equal to, or greater than the second list. Each ordering can only compare lists of one fixed length, called its its *order length*. Since the lists under consideration will be exponent vectors in most cases, their length is also referred to as the number of indeterminates.

Monomial orderings are used in algebraic geometry for comparing terms $\prod_{i=1}^{n} X_i^{\alpha_i}$ and $\prod_{i=1}^{n} X_i^{\beta_i}$ in a polynomial ring. Since `Dom::MonomOrdering` works on the exponent vectors $[\alpha_1, \ldots, \alpha_n]$ and $[\beta_1, \ldots, \beta_n]$, `degreevec` must be applied to the terms to be compared before applying `Dom::MonomOrdering`.

Elements of `Dom::MonomOrdering` can be used as arguments for `lcoeff`, `lmonomial`, `lterm`, and `tcoeff` as well as for the functions of the groebner package in order to specify the monomial ordering to be considered.

Monomial orderings are created by calling `Dom::MonomOrdering(someIdentifier(parameters))`, where `someIdentifier` is one of a certain set of predefined identifiers, as stated below. Converting `someIdentifier` into a string gives the *order type* of the monomial ordering.

`Dom::MonomOrdering(Lex(n))` creates the lexicographical order on n indeterminates.

`Dom::MonomOrdering(RevLex(n))` creates the reverse lexicographical order on n indeterminates, i.e., `Dom::MonomOrdering(RevLex(n))([a1,...,an])= Dom::MonomOrdering(Lex(n))([an,...,a1])`.

`Dom::MonomOrdering(DegLex(n))` creates the degree order on n indeterminates with the lexicographical order used for tie-break.

`Dom::MonomOrdering(DegRevLex(n))` creates the degree order on n indeterminates with the reverse lexicographical order used for tie-break .

`Dom::MonomOrdering(DegInvLex(n))` creates the degree order on n indeterminates, with the tie break being the opposite to the lexicographical order.

`Dom::MonomOrdering(Weighted...(w1,...,wn))` returns a weighted degree order with weights w_1 through w_n. The word following the word `Weighted` specifies the tie-break used. Note that MuPAD uses the ordinary degree order as the first tie-break.

`Dom::MonomOrdering(Matrix(params))` creates a matrix order, with the order matrix defined by `Dom::Matrix()(params)`.

`Dom::MonomOrdering(Block(o1, ..., on))` or, equivalently, `Dom::MonomOrdering([o1, ..., on])`, creates a block order such that `Dom::MonomOrdering(o1)` is used on the first indeterminates, then `Dom::MonomOrdering(o2)` is used as a tie-break on the following indeterminates etc.

Block orders may be nested, i.e., the blocks may be block orders, too.

Weight vectors with negative entries and order matrices do not define well-orderings in general. You may enter such orderings, but it may cause trouble, e.g., to use them with the groebner package.

Superdomain

`Dom::BaseDomain`

Categories

`Cat::BaseCategory`

Examples

Example 1

We define `ORD` by prescribing that lists $[a, b, c]$ are ordered according to their weighted degrees $5\,a + 2\,b + \pi\,c$. For lists with equal weighted degree, the non-weighted degree $a + b + c$ is used as a tie-break. Finally, the lexicographical order decides (in fact, this last step is not necessary because π is irrational).

```
ORD:=Dom::MonomOrdering(WeightedDegLex(5, 2, PI))
```

$$\text{WeightedDegLex}(5, 2, \pi)$$

With respect to `ORD`, `[1, 6, 1]` is smaller than `[2, 1, 3]`:

```
ORD([1,6,1], [2,1,3])
```

$$-1$$

Parameters

`n`

Positive integer

$w_1, \ldots$

Numerical expressions

$o_1, \ldots$

Valid arguments to `Dom::MonomOrdering`

`params`

A sequence valid as the sequence of arguments to `Dom::Matrix()`.

Methods

Mathematical Methods

`func_call` — Compare two lists of integers

```
func_call(o, l1, l2)
```

The lengths of `l1` and `l2` must not exceed the order length of `o`. If `l1` or `l2` is too short, the necessary number of zeroes is appended.

Access Methods

`ordertype` — Return the type of an order

```
ordertype(o)
```

If `o` equals `Dom::MonomOrdering(someIdentifier(params))`, then converting `someIdentifier` into a string gives the order type of `o`.

`orderlength` — Return the length of an order

```
orderlength(o)
```

`nops` — Number of blocks

```
nops(o)
```

`block` — Get a particular block

```
block(o, i)
```

`blocktype` — Get the order type of a particular block

```
blocktype(o, i)
```

`blocklength` — Get the order length of a particular block

```
blocklength(o, i)
```

Conversion Methods

`expr` — Return an expression from which the order can be restored

```
expr(o)
```

See Also

MuPAD Functions
`groebner::gbasis`

Dom::Multiset

Multisets

Syntax

```
Dom::Multiset(<s1, s2, …>)
```

Description

`Dom::Multiset` is the domain of multisets, i.e., sets with possibly multiple identical elements.

A multiset is represented by a set of lists of the form [*s*, *m*], where *s* is an element of the multiset and *m* its multiplicity.

Multisets can be returned by the system solver `solve`. For example, the input `solve(x^3 - 4*x^2 + 5*x - 2, x, Multiple)` gives all roots of the polynomial $x^3 - 4x^2 + 5x - 2$ in form of the multiset `{[1, 2], [2, 1]}`.

The standard set operations such as union, intersection and subtraction of sets have been extended to deal with multisets.

These operations can handle different types of sets, such as sets of type `DOM_SET` and multisets. One may, for example, compute the union of the multiset `{[a, 2], [b, 1]}` and the set `{c}`, which results in the multiset `{[a, 2], [b, 1], [c, 1]}`.

The elements of the multiset are sorted at the time where the multiset is created. The system function `sort` is used in order to guarantee that exactly one representation exists for a multiset, independent of the sequence in which the arguments appear.

`Dom::Multiset(s1, s2, ...)` creates the multiset consisting of the elements `s1, s2, ...`

Multiple identical elements in `s1, s2`, ... are collected. For example, the call `Dom::Multiset(a, b, a, c)` creates a multiset with the elements `a, b, c`. The element `a` has multiplicity two, the other two elements `b` and `c` both have multiplicity one.

Superdomain

`Dom::BaseDomain`

Categories

`Cat::Set`

Examples

Example 1

The multiset $\{a, a, b\}$ consists of the two different elements a and b, where a has multiplicity two and b has multiplicity one:

```
delete a, b, c:
set1 := Dom::Multiset(a, a, b)
```

$$\{[a, 2], [b, 1]\}$$

We create another multiset:

```
set2 := Dom::Multiset(a, c, c)
```

$$\{[a, 1], [c, 2]\}$$

Standard set operations such as disjoint union, intersection or subtraction are implemented for multisets and can be performed using the standard set operators of MuPAD:

```
set1 union set2
```

$$\{[a, 3], [b, 1], [c, 2]\}$$

```
set1 intersect set2
```

$$\{[a, 1]\}$$

```
contains(set1, a), contains(set1, d)
```

$$\text{TRUE, FALSE}$$

Example 2

Some system functions were overloaded for multisets, such as `expand`, `normal` or `split`.

If we apply `expand` to a multiset, for example, we get an expression sequence of all elements of the multiset (appearing in correspondence to their multiplicity):

```
delete a, b, c, d, e:
set := Dom::Multiset(a, b, c, a, c, d, c, e, c)
```

$$\{[a, 2], [b, 1], [c, 4], [d, 1], [e, 1]\}$$

```
expand(set)
```

$$a, a, b, c, c, c, c, d, e$$

If you want to convert a multiset into an ordinary set of the domain type `DOM_SET`, use `coerce`:

```
coerce(set, DOM_SET)
```

$$\{a, b, c, d, e\}$$

Note: The system function `coerce` uses the methods `"convert"` and `"convert_to"` of the domain `Dom::Multiset`.

Compare the last result with the return value of the function `expr`, when it is applied for multisets:

```
expr(set)
```

$$\{[a, 2], [b, 1], [c, 4], [d, 1], [e, 1]\}$$

The result is a set of the domain type DOM_SET, consisting of lists of the domain type DOM_LIST with two entries, an element of the multiset and the corresponding multiplicity of that element.

Parameters

s1, s2, …

Objects of any type

Entries

"isFinite"	is TRUE because Dom::Multiset represents finite sets.
"inhomog_intersect"	a table of the form T = Proc(multiset, setoftypeT). This entry is used internally by the implementation, and thus should not be touched.
"inhomog_union"	a table of the form T = Proc(multiset, setoftypeT). This entry is used internally by the implementation, and thus should not be touched.

Methods

Mathematical Methods

normal — Normalization of multisets

```
normal(set)
```

This method overloads the function normal for multisets, i.e., one may use it in the form normal(set).

`powerset` — Power set of a multiset

```
powerset(set)
```

The power set of `set` is returned as a set of multisets.

`random` — Random multiset generation

```
random()
```

The number of elements created, including their multiplicities, is restricted to 20.

Access Methods

`_index` — Multiset indexing

```
_index(set, i)
```

See the method `"op"`.

This method overloads the function `_index` for multisets, i.e., one may use it in the form `set[i]`, or in functional notation: `_index(set, i)`.

`contains` — Check on existence of set elements

```
contains(set, s)
```

This method overloads the function `contains` for multisets, i.e., one may use it in the form `contains(set, s)`.

`equal` — Test on equality of multisets

```
equal(set1, set2)
```

The system function `_equal` is used for the test.

`expand` — Expand a multiset to a sequence of its elements

```
expand(set)
```

This method overloads the function `expand` for multisets, i.e., one may use it in the form `expand(set)`.

`getElement` — Extract one element from a multiset

```
getElement(set)
```

Note that the elements of the multiset are sorted with the use of the system function `sort`, and thus the order of a multiset depends on the sorting criteria specified by this function.

This method overloads the function `solvelib::getElement`, i.e., one may use it in the form `solvelib::getElement(set)`.

`has` — Check on existence of (sub-)expressions

```
has(set, expr)
```

To check whether `expr` is contained as an element of `set` and not as a subexpression of the elements of `set`, the function `contains` must be used.

This method overloads the function `has` for multisets, i.e., one may use it in the form `has(set, expr)`.

`map` — Apply a function to multiset elements

```
map(set, func, <expr, …>)
```

It overloads the function `map` for multisets, i.e., one may use it in the form `map(set, func, ...)`.

`multiplicity` — Multiplicity of an element

```
multiplicity(set, s)
```

Elements which are not contained in `set` have multiplicity zero.

`card` — Number of elements in a multiset

```
card(set)
```

This method overloads the function `card`.

`nops` — Number of different elements in a multiset

```
nops(set)
```

This method overloads the function `nops` for multisets, i.e., one may use it in the form `nops(set)`.

`op` — Element of a multiset

```
op(set)

op(set, i)
```

Returns the *i*-th element *s* of the multiset `set` and its multiplicity *m* in form of the list [*s*, *m*].

See also the method `"_index"`.

Note that the elements of the multiset are sorted with the use of the system function `sort`, and thus the order of a multiset depends on the sorting criteria specified by this function.

This method overloads the function `op` for multisets, i.e., one may use it in the form `op(s, i)`.

`select` — Selecting of multiset elements

```
select(set, func, <expr, …>)
```

This method overloads the function `select` for multisets, i.e., one may use it in the form `select(set, func, ...)`. See `select` for details.

`split` — Splitting a multiset

```
split(set, func, <expr, …>)
```

This method overloads the function `split` for multisets, i.e., one may use it in the form `split(set, func, ...)`. See `split` for details.

`subs` — Substitution of elements in multisets

```
subs(set, …)
```

This method overloads the function `subs` for multisets, i.e., one may use it in the form `subs(set, ...)`.

Conversion Methods

convert — Conversion into a multiset

```
convert(x)
```

FAIL is returned if the conversion fails.

Currently only sets of type DOM_SET can be converted into multisets.

convert_to — Multiset conversion

```
convert_to(set, T)
```

FAIL is returned if the conversion fails.

Currently T may either be DOM_SET to convert the multiset set into a set (loosing the multiplicities and the order of the elements of set), or DOM_EXPR or "_exprseq" to convert set into an expression sequence (see the method "expand" for details).

See also the method "expr".

expr — Multiset conversion into an object of a kernel domain

```
expr(set)
```

This method overloads the function expr for multisets, i.e., one may use it in the form expr(set).

sort — Sorting of multisets

```
sort(set)
```

This method overloads the function sort for multisets, i.e., one may use it in the form sort(set).

Technical Methods

bin_intersect — Intersection of two multisets

```
bin_intersect(set1, set2)
```

This method is called from routines defined in the category `Cat::Set`, which implements among others the overloading of the function `intersect` for multisets. One may intersect two multisets directly by `set1 intersect set2`, or in functional notation by `_intersect(set1, set2)`.

`bin_minus` — Subtraction of two multisets

```
bin_minus(set1, set2)
```

This method is called from routines defined in the category `Cat::Set`, which implements among others the overloading of the function `minus` for multisets. One may subtract two multisets directly by `set1 minus set2`, or in functional notation by `_minus(set1, set2)`.

`homog_union` — Union of multisets

```
homog_union(set, …)
```

This method is called from routines defined in the category `Cat::Set`, which implements among others the overloading of the function `union` for multisets. One may compute the union of two multisets directly by `set1 union set2`, or in functional notation by `_union(set1, set2)`.

`nested_union` — Union of nested sets

```
nested_union(setofsets)
```

This method is called from routines defined in the category `Cat::Set`, which implements among others the overloading of the function `union` for multisets and sets. One may compute the union of multisets and sets directly by `set1 union set2`, or in functional notation by `_union(set1, set2)`.

See Also

MuPAD Domains
`Dom::ImageSet` | `DOM_SET`

Dom::MultivariatePolynomial

Domains of multivariate polynomials

Syntax

Domain Creation

```
Dom::MultivariatePolynomial(<Vars, <R, <Order>>>)
```

Element Creation

```
Dom::MultivariatePolynomial(Vars, R, Order)(p)
```

```
Dom::MultivariatePolynomial(Vars, R, Order)(lm)
```

Description

`Dom::MultivariatePolynomial(Vars, R, ..)` creates the domain of multivariate polynomials in the variable list `Vars` over the commutative ring `R` in distributed representation.

`Dom::MultivariatePolynomial` represents multivariate polynomials over arbitrary commutative rings.

All usual algebraic and arithmetical polynomial operations are implemented, including Gröbner basis computation and some classical construction tools used in invariant theory.

Note: It is highly recommend to use only coefficient rings with unique zero representation. Otherwise it may happen that, e.g., a polynomial division will not terminate or a wrong degree will be returned.

`Dom::MultivariatePolynomial(Vars, R, Order)` creates a domain of multivariate polynomials in the variable list `Vars` over a domain `R` of category `Cat::CommutativeRing` in sparse distributed representation with respect to the monomial ordering `Order`.

`Dom::MultivariatePolynomial()` creates a polynomial domain in the variable list `[x,y,z]` over the domain `Dom::ExpressionField(normal)` with respect to the lexicographic monomial ordering.

`Dom::MultivariatePolynomial(Vars)` generates the polynomial domain in the variable list `Vars` over the domain `Dom::ExpressionField(normal)` with respect to the lexicographic monomial ordering is created.

Note: Only commutative coefficient rings of type `DOM_DOMAIN` which inherit from `Dom::BaseDomain` are allowed. If `R` is of type `DOM_DOMAIN` but does not inherit from `Dom::BaseDomain`, the domain `Dom::ExpressionField(normal)` will be used instead.

In contrast to the domain `Dom::DistributedPolynomial`, `Dom::MultivariatePolynomial` accepts only identifiers (`DOM_IDENT`) as indeterminates. This restriction enables some further methods described below.

Please note: For reasons of efficiency not all methods check their arguments, not even at the interactive level. In particular this is true for many access methods, converting methods and technical methods. This may cause strange error messages.

Superdomain

`Dom::DistributedPolynomial`

Axioms

If R has `Ax::normalRep`, then `Ax::normalRep`.

If R has `Ax::canonicalRep`, then `Ax::canonicalRep`.

Categories

If Vars has a single variable, then `Cat::UnivariatePolynomial(R)`, else `Cat::Polynomial(R)`.

Examples

Example 1

To create the ring of multivariate polynomials in x, y and z over the rationals one may define

```
MP := Dom::MultivariatePolynomial([x, y, z], Dom::Rational)
```

$$\text{Dom::MultivariatePolynomial}([x, y, z], \text{Dom::Rational}, \text{LexOrder})$$

The elementary symmetric polynomials of this domain are

```
s1 := MP(x + y + z)
```

$$x + y + z$$

```
s2 := MP(x*y + x*z + y*z)
```

$$x\,y + x\,z + y\,z$$

```
s3:=MP(x*y*z)
```

$$x\,y\,z$$

A polynomial is called symmetric if it remains unchanged under every possible permutation of variables as, e.g.:

```
s3=s3(MP(y), MP(z), MP(x))
```

$$x\,y\,z = x\,y\,z$$

These polynomials arise naturally in studying the roots of a polynomial. To show this, we first have to create an univariate polynomial, e.g., in U over MP, and generate a polynomial in U with roots in x, y and z.

```
UP:=Dom::UnivariatePolynomial(U, MP)
```

Dom::UnivariatePolynomial(*U*, Dom::MultivariatePolynomial([*x*, *y*, *z*], Dom::Rational, LexOrder), LexOrder)

```
f := UP((U - x)*(U - y)*(U - z))
```

$$U^3 + (-x - y - z)\,U^2 + (x\,y + x\,z + y\,z)\,U - x\,y\,z$$

```
UP(U^3)-s1*UP(U^2)+s2*UP(U)+(-1)^3*s3
```

$$U^3 + (-x - y - z)\,U^2 + (x\,y + x\,z + y\,z)\,U - x\,y\,z$$

This exemplifies that the coefficients of `f` are (elementary) symmetric polynomials in its roots.

From the fundamental theorem of symmetric polynomials we know that every symmetric polynomial can be written uniquely as a polynomial in the elementary symmetric polynomials. Thus we can rewrite the following symmetric polynomial `s` in the elementary symmetric polynomials `s1`, `s2` and `s3`,

```
s:=MP(x^3*y+x^3*z+x*y^3+x*z^3+y^3*z+y*z^3)
```

$$x^3\,y + x^3\,z + x\,y^3 + x\,z^3 + y^3\,z + y\,z^3$$

```
S:=MP::rewritePoly(s,[s1=S1,s2=S2,s3=S3])
```

$$S1^2\,S2 - S1\,S3 - 2\,S2^2$$

where these polynomials are represented by the three new variables `S1`, `S2` and `S3` respectively. To see that this new polynomial `S` in the new variables indeed represents the old original polynomial `s`, we simply have to plug in the three elementary symmetric polynomials into `S`:

```
poly(S, Expr)(s1,s2,s3)
```

$$x^3\,y + x^3\,z + x\,y^3 + x\,z^3 + y^3\,z + y\,z^3$$

When one has a given list of polynomials, e.g., like:

```
l:=[3*s1,2*s1,s1,s3]
```

$$[3x+3y+3z,\ 2x+2y+2z,\ x+y+z,\ xyz]$$

and one wants to sort them in an appropriate order, one may use one of the following two methods.

```
MP::sortList(l,Dom::MonomOrdering(DegLex(3)))
```

$$[xyz,\ 2x+2y+2z,\ x+y+z,\ 3x+3y+3z]$$

```
MP::stableSort(l,Dom::MonomOrdering(DegLex(3)))
```

$$[xyz,\ 3x+3y+3z,\ 2x+2y+2z,\ x+y+z]$$

In the first sorted list the order of the three polynomials of the same degree has changed, while with the second method this order remains stable.

Example 2

Let $G \subset \mathrm{GL}(n, k)$ be a finite (matrix) subgroup of the general linear group. Then a polynomial $f \in k[x_1, \ldots, x_n]$ is called *invariant under* G, if for all $A \in G$

$$f(x) = f(A\,x)$$

where $x = (x_1 \ \cdots \ x_n)$. The symmetric polynomials `s1`, `s2` and `s3` from the previous example are invariants under the symmetric group S_3. In fact, these three fundamental invariants yet generate the whole ring of invariants of S_3.

Now let us examine the invariants of the famous icosahedral group. One may find a representation of this group on page 73 of H. F. Blichfeldt: Finite collineation groups, University of Chicago Press, 1917.

$$S' = \begin{pmatrix} \varepsilon^3 & 0 \\ 0 & \varepsilon^2 \end{pmatrix},\ U' = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix},\ T' = \begin{pmatrix} \alpha & \beta \\ \beta & -\alpha \end{pmatrix},\ \varepsilon^5 = 1,\ \alpha = \frac{\varepsilon^4 - \varepsilon}{\sqrt{5}},\ \beta = \frac{\varepsilon^2 - \varepsilon^3}{\sqrt{5}}$$

The group is generated from these three matrices, has 120 elements and is thus a finite subgroup, even of the special linear group $SL(2, \mathbb{Q}(\varepsilon))$. It is also well known that

$$i_1 = x_1 x_2^{11} - x_1^6 x_2^6 - x_1^{11} x_2$$

is a fundamental invariant of degree 12 of this group. To declare i_1 in MuPAD one has first to define the polynomial domain.

```
MP:=Dom::MultivariatePolynomial([x1,x2],Dom::Rational)
```

$$\text{Dom::MultivariatePolynomial}([x1, x2], \text{Dom::Rational}, \text{LexOrder})$$

```
i1:=MP(x1*x2^(11)-11*x1^6*x2^6-x1^(11)*x2)
```

$$-x1^{11}\,x2 - 11\,x1^6\,x2^6 + x1\,x2^{11}$$

From the invariant i_1 one can compute a further fundamental invariant i_2 with

```
i2:=MP::hessianDet(i1)
```

$$-121\,x1^{20} + 27588\,x1^{15}\,x2^5 - 59774\,x1^{10}\,x2^{10} - 27588\,x1^5\,x2^{15} - 121\,x2^{20}$$

But to get more simple coefficients we choose i_2 as

```
i2:=-1/121*MP::hessianDet(i1)
```

$$x1^{20} - 228\,x1^{15}\,x2^5 + 494\,x1^{10}\,x2^{10} + 228\,x1^5\,x2^{15} + x2^{20}$$

instead. Similar we obtain a third fundamental invariant i_3 with

```
i3:=1/20*MP::jacobianDet([i1,i2])
```

$$x1^{30} + 522\,x1^{25}\,x2^5 - 10005\,x1^{20}\,x2^{10} - 10005\,x1^{10}\,x2^{20} - 522\,x1^5\,x2^{25} + x2^{30}$$

In contrast to the symmetric groups, where all invariants can be uniquely represented by the fundamental invariants, the fundamental invariants of this group have an algebraic relation, a so-called syzygy between them. It is possible to represent i_3^2 in two ways:

```
MP::rewritePoly(i3^2,[i1=I1,i2=I2,i3=I3])
```

$$-1728\,I1^5 + I2^3$$

```
MP::rewritePoly(i3^2,[i1=I1,i2=I2,i3=I3],Unsorted)
```

$$I3^2$$

And hence we get the syzygy:

```
MP::rewritePoly(i3^2,[i1=I1,i2=I2,i3=I3],Unsorted)-
MP::rewritePoly(i3^2,[i1=I1,i2=I2,i3=I3]) = 0
```

$$1728\,I1^5 - I2^3 + I3^2 = 0$$

Parameters

`Vars`

A list of indeterminates. Default: `[x,y,z]`.

`R`

A commutative ring, i.e., a domain of category `Cat::CommutativeRing`. Default: `Dom::ExpressionField(normal)`.

`Order`

A monomial ordering, i.e., one of the predefined orderings `LexOrder`, `DegreeOrder`, or `DegInvLexOrder` or any object of type `Dom::MonomOrdering`. Default: `LexOrder`.

`p`

A polynomial or a polynomial expression.

`lm`

List of monomials, which are represented as lists containing the coefficients together with the exponents or exponent vectors.

Entries

"characteristic"	The characteristic of this domain.
"coeffRing"	The coefficient ring of this domain as defined by the parameter `R`.
"key"	The name of the domain created.
"one"	The neutral element w.r.t. `"_mult"`.
"ordering"	The monomial ordering defined by the parameter `Order`.
"variables"	The list of variables defined by the parameter `Vars`.
"zero"	The neutral element w.r.t. `"_plus"`.

Methods

Mathematical Methods

`D` — Differential operator for polynomials

Inherited from `Dom::DistributedPolynomial`.

`Dpoly` — Differential operator for polynomials

Inherited from `Dom::DistributedPolynomial`.

`SPolynomial` — Compute the S-polynomial of two polynomials

Inherited from `Dom::DistributedPolynomial`.

`_divide` — Exact polynomial division

Inherited from `Dom::DistributedPolynomial`.

`_invert` — Inverse of an element

Inherited from `Dom::DistributedPolynomial`.

_mult — Multiplie polynomials and coefficient ring elements

Inherited from `Dom::DistributedPolynomial`.

_negate — Negate a polynomial

Inherited from `Dom::DistributedPolynomial`.

_plus — Add polynomials and coefficent ring elements

Inherited from `Dom::DistributedPolynomial`.

_power — Nth power of a polynomial

Inherited from `Dom::DistributedPolynomial`.

_subtract — Subtract a polynomial or a coefficient ring element

Inherited from `Dom::DistributedPolynomial`.

associates — Test if elements are associates

Inherited from `Cat::IntegralDomain`.

borderedHessianDet — Bordered Hessian determinant of a polynomial

```
borderedHessianDet(a, b, <v>)
```

borderedHessianMat — Bordered Hessian matrix of a polynomial

```
borderedHessianMat(a, b, <v>)
```

content — Content of a polynomial

Inherited from `Dom::DistributedPolynomial`.

decompose — Functional decomposition of a polynomial

Inherited from `Dom::DistributedPolynomial`.

degLex — Compare two polynomials w.r.t. the graded lexicographical order

```
degLex(a, b)
```

`degRevLex` — Compare two polynomials w.r.t. the graded reverse lexicographical order

```
degRevLex(a, b)
```

`diff` — Differentiate a polynomial

Inherited from `Dom::DistributedPolynomial`.

`dimension` — Dimension of affine variety

Inherited from `Dom::DistributedPolynomial`.

`divide` — Divide polynomials

Inherited from `Dom::DistributedPolynomial`.

`divides` — Test if elements divides another

Inherited from `Cat::IntegralDomain`.

`equal` — Test for mathematical equality

Inherited from `Dom::BaseDomain`.

`equiv` — Test for equivalence

Inherited from `Cat::BaseCategory`.

`evalp` — Evaluate a polynomial

Inherited from `Dom::DistributedPolynomial`.

`factor` — Factor a polynomial

Inherited from `Dom::DistributedPolynomial`.

`func_call` — Applie expressions to a polynomial

Inherited from `Dom::DistributedPolynomial`.

`gcd` — Greatest common divisor of polynomials

Inherited from `Dom::DistributedPolynomial`.

`gcdex` — Extended Euclidean algorithm for polynomials

Inherited from `Dom::DistributedPolynomial`.

`groebner` — Reduced Gröbner basis

Inherited from `Dom::DistributedPolynomial`.

`hessianDet` — Hessian determinant of a polynomial

```
hessianDet(a, <v>)
```

`hessianMat` — Hessian matrix of a polynomial

```
hessianMat(a, <v>)
```

`homogeneousComponents` — List of homogeneous components of a polynomial

```
homogeneousComponents(a)
```

`idealGenerator` — Generator of finitely generated ideal

Inherited from `Cat::EuclideanDomain`.

`int` — Definite and indefinite integration of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`intmult` — Multiplie a polynomial with an integer

Inherited from `Dom::DistributedPolynomial`.

`irreducible` — Test if element is irreducible

Inherited from `Cat::FactorialDomain`.

`isHomogeneous` — Test if a polynomial is homogeneous

```
isHomogeneous(a)
```

`isUnit` — Test if element is a unit

Inherited from `Cat::Polynomial`.

`isone` — Test for one

Inherited from `Dom::DistributedPolynomial`.

`iszero` — Test for zero

Inherited from `Dom::DistributedPolynomial`.

`jacobianDet` — Jacobian determinant of a polynomial

```
jacobianDet(ais, <v>)
```

`jacobianMat` — Jacobian matrix of a polynomial

```
jacobianMat(ais, <v>)
```

`lcm` — Least common multiple of polynomials

Inherited from `Dom::DistributedPolynomial`.

`makeIntegral` — Make the coefficients fraction free

Inherited from `Dom::DistributedPolynomial`.

`monic` — Normalize a polynomial

Inherited from `Dom::DistributedPolynomial`.

`normalForm` — Complete reduction modulo an ideal

Inherited from `Dom::DistributedPolynomial`.

`pdioe` — Solve polynomial Diophantine equations

Inherited from `Dom::DistributedPolynomial`.

`pdivide` — Pseudo-division of polynomials

Inherited from `Dom::DistributedPolynomial`.

`pquo` — Pseudo-quotient of polynomials

Inherited from `Dom::DistributedPolynomial`.

prem — Pseudo-remainder of polynomials

Inherited from `Dom::DistributedPolynomial`.

primpart — Return primitive part

Inherited from `Cat::Polynomial`.

quo — Euclidean quotient

Inherited from `Cat::EuclideanDomain`.

random — Create a random polynomial

Inherited from `Dom::DistributedPolynomial`.

rem — Euclidean remainder

Inherited from `Cat::EuclideanDomain`.

resultant — Resultant of two polynomials

Inherited from `Dom::DistributedPolynomial`.

rewriteHomPoly — Rewrite a polynomial in terms of other polynomials

```
rewriteHomPoly(a, ais, v)
```

All the polynomials `a` and `ais` must be homogeneous.

The variables of `v` should be new variables.

rewritePoly — Rewrite a polynomial in terms of other polynomials

```
rewritePoly(a, [ai = vi], <Unsorted>)
```

This method can be used for representing a polynomial with respect to a given polynomial basis.

When option `Unsorted` is given, the list `[ai=vi]` is not sorted. Otherwise, in a precomputation step this list will be sorted in the `ai`'s w.r.t. the graded lexicographical order (`"degLex"`).

Please note: the algorithm depends on the order of `Vars` and `ais`.

All the polynomials `ai` must be homogeneous.

The variables of `vi` should be new variables.

`ringmult` — Multiplie a polynomial with a coefficient ring element

Inherited from `Dom::DistributedPolynomial`.

`solve` — Zero of polynomials

Inherited from `Dom::DistributedPolynomial`.

`sqrfree` — Square-free factorization of polynomials

Inherited from `Dom::DistributedPolynomial`.

`unitNormal` — Return unit normal

Inherited from `Cat::Polynomial`.

`unitNormalRep` — Return unit normal representation

Inherited from `Cat::Polynomial`.

Access Methods

`coeff` — Coefficient of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`degree` — Degree of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`degreevec` — Vector of exponents of the leading term of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`euclideanDegree` — Euclidean degree function

Inherited from `Dom::DistributedPolynomial`.

`ground` — Ground term of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`has` — Existence of an object in a polynomial

Inherited from `Dom::DistributedPolynomial`.

`indets` — Indeterminate of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`lcoeff` — Leading coefficient of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`ldegree` — Lowest degree of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`lmonomial` — Leading monomial of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`lterm` — Leading term of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`mainvar` — Main variable of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`mapcoeffs` — Applie a function to the coefficients of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`multcoeffs` — Multiplie the coefficients of a polynomial with a factor

Inherited from `Dom::DistributedPolynomial`.

`nterms` — Number of terms of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`nthcoeff` — N-th coefficient of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`nthmonomial` — N-th monomial of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`nthterm` — N-th term of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`order` — Compare two polynomials w.r.t. a given order

```
order(a, b, o)
```

`orderedVariableList` — Ordered list of indeterminates of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`pivotSize` — Size of a pivot element

Inherited from `Dom::DistributedPolynomial`.

`reductum` — Reductum of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`sortList` — Sort a list of polynomials w.r.t. a given order

```
sortList(ais, o)
```

This sorting method may be not stable if `o` is not a total order.

`stableSort` — Sort a list of polynomials w.r.t. a given order

```
stableSort(ais, o)
```

This sorting method is stable, even if `o` is not a total order.

`subs` — Avoid substitution

Inherited from `Dom::BaseDomain`.

`subsex` — Avoid extended substitution

Inherited from `Dom::BaseDomain`.

`tcoeff` — Lowest coefficient of a polynomial

Inherited from `Dom::DistributedPolynomial`.

References

[1] Winfried Fakler. “Algorithmen zur symbolischen Lösung homogener linearer Differentialgleichungen”. Diplomarbeit, Universität Karlsruhe, 1994.

See Also

MuPAD Domains

`Dom::DistributedPolynomial` | `Dom::Polynomial` | `Dom::UnivariatePolynomial`

Dom::Natural

Semi-ring of natural integer numbers

Syntax

```
Dom::Natural(x)
```

Description

`Dom::Natural` is the semi-ring of integer numbers represented by elements of the domain `DOM_INT`.

`Dom::Natural` is the domain of natural integer numbers represented by expressions of type `DOM_INT`.

Elements of `Dom::Natural` are usually not created explicitly. However, if one creates elements using the usual syntax, it is checked whether the input is an integer number. This means that `Dom::Natural` is a façade domain which creates elements of domain type `DOM_INT`.

Viewed as a differential ring `Dom::Natural` is trivial, it contains constants only.

`Dom::Natural` has the domain `Dom::Numerical` as its super domain, i.e., it inherits each method which is defined by `Dom::Numerical` and not re-implemented by `Dom::Natural`. Methods described below are those implemented by `Dom::Natural`.

Superdomain

`Dom::Numerical`

Axioms

`Ax::canonicalRep`, `Ax::systemRep`, `Ax::canonicalOrder`, `Ax::canonicalUnitNormal`, `Ax::closedUnitNormals`, `Ax::efficientOperation("_divide")`, `Ax::efficientOperation("_mult")`

Categories

`Cat::EuclideanDomain`, `Cat::FactorialDomain`, `Cat::DifferentialRing`, `Cat::OrderedSet`

Examples

Example 1

Creating some integer numbers using `Dom::Natural`. This example also shows that `Dom::Natural` is a façade domain.

```
Dom::Natural(2); domtype(%)
```

```
    2

    DOM_INT
```

```
Dom::Natural(2/3)
```

```
Error: The arguments are invalid. [Dom::Natural::new]
```

Example 2

By tracing the method `Dom::Natural::testtypeDom` we can see the interaction between `testtype` and `Dom::Natural::testtypeDom`.

```
prog::trace(Dom::Natural::testtypeDom):
delete x:
testtype(x, Dom::Natural);
testtype(3, Dom::Natural);
prog::untrace(Dom::Natural::testtypeDom):
```

```
enter Dom::Natural::testtypeDom(x, Dom::Natural)
computed FAIL

    FALSE
```

```
enter Dom::Natural::testtypeDom(3, Dom::Natural)
computed TRUE

    TRUE
```

Parameters

x

An integer

Methods

Mathematical Methods

`_divide` — Division of two objects

```
_divide(x, y)
```

`_divides` — Decide if a number divides another one

```
_divides(x, y)
```

`euclideanDegree` — Euclidean degree

```
euclideanDegree(x)
```

`factor` — Factorization

```
factor(x)
```

`gcd` — Gcd computation

```
gcd(x1, x2, …)
```

`gcdex` — Applie the extended Euclidean algorithm

```
gcdex(x, y)
```

`_invert` — Inverse of an element

```
_invert(x)
```

`irreducible` — Prime number test

```
irreducible(x)
```

`isUnit` — Test if an element is a unit

```
isUnit(x)
```

`lcm` — Compute the lcm

```
lcm(x1, x2, …)
```

`quo` — Compute the euclidean quotient

```
quo(x, y)
```

`random` — Random number generation

```
random()
```

```
random(n)
```

```
random(m, …, n)
```

This methods returns a random number between 0 and n - 1.

This methods returns a random number between `m` and `n`.

`rem` — Compute the Euclidean reminder

```
rem(x, y)
```

`unitNormal` — Unit normal part

```
unitNormal(x)
```

`unitNormalRep` — Unit normal representation

```
unitNormalRep(x)
```

Conversion Methods

`convert` — Conversion of objects

```
convert(x)
```

`convert_to` — Conversion to other domains

```
convert_to(x, T)
```

The following domains are allowed for for `T`: `DOM_INT`, `Dom::Natural`, `Dom::Rational`, `DOM_FLOAT`, `Dom::Float` and `Dom::Numerical`.

`testtype` — Type checking

```
testtype(x, T)
```

Usually, this method is called from the function `testtype` and not directly by the user. “Example 2” on page 7-330 demonstrates this behavior.

See Also

MuPAD Domains
`Dom::Complex` | `Dom::Float` | `Dom::Integer` | `Dom::Numerical` | `Dom::Rational`

Dom::Numerical

Field of numbers

Syntax

```
Dom::Numerical(x)
```

Description

`Dom::Numerical` is the field of numbers.

`Dom::Numerical` is the domain of numbers represented by one of the kernel domains `DOM_INT`, `DOM_RAT`, `DOM_FLOAT`, or `DOM_COMPLEX`.

`Dom::Numerical` is of category `Cat::Field` due to pragmatism. This domain actually is not a field because `bool(1.0 = float(3) / float(3))` returns `FALSE`, for example.

Elements of `Dom::Numerical` are usually not created explicitly. However, if one creates elements using the usual syntax, it is checked whether the input expression can be converted into a number (see below).

This means that `Dom::Numerical` is a façade domain which creates elements of domain type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT` or `DOM_COMPLEX`. Every system function dealing with numbers can be applied, and computations in this domain are performed efficiently.

`Dom::Numerical` has no normal representation, because `0` and `0.0` both represent zero.

Viewed as a differential ring, `Dom::Numerical` is trivial. It only contains constants.

If `x` is a constant arithmetical expression such as `sin(2)` or `PI + 2`, the system function `float` is applied to convert `x` into a floating-point approximation.

An error message is issued if the result of this conversion is not of domain type `DOM_FLOAT` or `DOM_COMPLEX`.

Superdomain

`Dom::ArithmeticalExpression`

Axioms

`Ax::canonicalRep`, `Ax::systemRep`, `Ax::efficientOperation("_divide")`, `Ax::efficientOperation("_mult")`, `Ax::efficientOperation("_invert")`

Categories

`Cat::DifferentialRing`, `Cat::Field`

Examples

Example 1

`Dom::Numerical` contains numbers of the domains `DOM_INT`, `DOM_RAT`, `DOM_FLOAT` and `DOM_COMPLEX`:

```
Dom::Numerical(2), Dom::Numerical(2/3),
Dom::Numerical(3.141), Dom::Numerical(2 + 3*I)
```

$$2,\ \frac{2}{3},\ 3.141,\ 2+3\,i$$

Constant arithmetical expressions are converted into a real and complex floating-point number, respectively, i.e., into an element of the domain `DOM_FLOAT` or `DOM_COMPLEX` (see the function `float` for details):

```
Dom::Numerical(exp(5)), Dom::Numerical(sin(2/3*I) + 3)
```

$$148.4131591,\ 3.0+0.717158461\,i$$

Note that the elements of this domain are elements of kernel domains, there are no elements of the domain type `Dom::Numerical`!

An error message is issued for non-constant arithmetical expressions:

```
Dom::Numerical(sin(x))
```

```
Error: The arguments are invalid. [Dom::Numerical::new]
```

Example 2

`Dom::Numerical` is regarded as a field, and it therefore can be used as a coefficient ring of polynomials or as a component ring of matrices, for example.

We create the domain of matrices of arbitrary size (see `Dom::Matrix`) with numerical components:

```
MatN := Dom::Matrix(Dom::Numerical)
```

$$\text{Dom::Matrix(Dom::Numerical)}$$

Next we create a banded matrix, such as:

```
A := MatN(4, 4, [-PI, 0, PI], Banded)
```

$$\begin{pmatrix} 0 & 3.141592654 & 0 & 0 \\ -3.141592654 & 0 & 3.141592654 & 0 \\ 0 & -3.141592654 & 0 & 3.141592654 \\ 0 & 0 & -3.141592654 & 0 \end{pmatrix}$$

and a row vector with four components as a 1 ×4 matrix:

```
v := MatN([[2, 3, -1, 0]])
```

$$\begin{pmatrix} 2 & 3 & -1 & 0 \end{pmatrix}$$

Vector-matrix multiplication can be performed with the standard operator `*` for multiplication:

```
v * A
```

$$\begin{pmatrix} -9.424777961 & 9.424777961 & 9.424777961 & -3.141592654 \end{pmatrix}$$

Finally we compute the determinant of the matrix A, using the function `det`:

```
det(A)
```

```
97.40909103
```

Parameters

x

An arithmetical expression

Entries

"characteristic" is zero.

Methods

Mathematical Methods

D — Differential operator for numbers

```
D(a)
```

See the function D for details and further calling sequences.

diff — Differentiation of numbers

```
diff(a, x)
```

See the function `diff` for details and further calling sequences.

norm — Absolute value of numbers

```
norm(a)
```

`random` — Random number generation

```
random()
```

Conversion Methods

`convert` — Conversion of objects into numbers

```
convert(x)
```

If `x` is of the domain type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT` or `DOM_COMPLEX`, `x` is returned.

Otherwise `float(x)` is computed and the result is returned, if it is of the domain type `DOM_FLOAT` or `DOM_COMPLEX`. If it is not, `FAIL` is returned.

`convert_to` — Conversion into other domains

```
convert_to(a, T)
```

If the conversion fails, `FAIL` is returned.

It currently handles the following domains for `T`: `DOM_INT`, `Dom::Integer`, `DOM_RAT`, `Dom::Rational`, `DOM_FLOAT`, `Dom::Float` and `DOM_COMPLEX`.

`testtype` — Type checking

```
testtype(a, T)
```

This method is called from the function `testtype`.

See Also

MuPAD Domains
`Dom::Complex` | `Dom::Float` | `Dom::Integer` | `Dom::Rational` | `Dom::Real`

Dom::Polynomial

Domains of polynomials in arbitrarily many indeterminates

Syntax

Domain Creation

```
Dom::Polynomial(<R, <Order>>)
```

Element Creation

```
Dom::PolynomialRorder(p)
```

```
Dom::PolynomialRorder(lm, v)
```

Description

`Dom::Polynomial(R, ..)` creates the domain of polynomials in arbitrarily many indeterminates over the commutative ring `R` in distributed representation.

`Dom::Polynomial` represents polynomials in arbitrarily many indeterminates over arbitrary commutative rings.

It is simply a front end to the domain `Dom::DistributedPolynomial([],R,Order)` and thus all usual algebraic and arithmetical polynomial operations are implemented. Please see the documentation for `Dom::DistributedPolynomial` for a list of methods.

`Dom::Polynomial(R, Order)` creates a domain of polynomials in arbitrarily many indeterminates over a domain of category `Cat::CommutativeRing` in sparse distributed representation with respect to the monomial ordering `Order`.

If `Dom::Polynomial` is called without any argument, a polynomial domain over the domain `Dom::ExpressionField(normal)` with respect to the lexicographic monomial ordering is created.

Note: Only commutative coefficient rings of type `DOM_DOMAIN` which inherit from `Dom::BaseDomain` are allowed. If `R` is of type `DOM_DOMAIN` but does not inherit from

`Dom::BaseDomain`, the domain `Dom::ExpressionField(normal)` will be used instead.

Only identifiers should be used as polynomial indeterminates, since when creating a new element from a polynomial or a polynomial expression the function `indets` is first called to get the identifiers and then the polynomial is created with respect to these identifiers.

Note: It is highly recommend to use only coefficient rings with unique zero representation. Otherwise it may happen that, e.g., a polynomial division will not terminate or a wrong degree will be returned.

Please note that for reasons of efficiency not all methods check their arguments, not even at the interactive level. In particular, this is true for many access methods, converting methods and technical methods. Thus, improper use of these methods may result in confusing error messages.

Superdomain

`Dom::DistributedPolynomial`

Axioms

`Ax::indetElements`

Categories

`Cat::Polynomial(R)`

Examples

Example 1

The following call creates the polynomial domain over the rationals.

```
PR:=Dom::Polynomial(Dom::Rational)
```

Dom::Polynomial(Dom::Rational, LexOrder)

Since the monomial ordering was not specified, this domain is created with the default value for this parameter.

It is rather easy to create elements of this domain, as, e.g.,

```
a := PR(x*(2*x + y^3) - 7/2)
```

$$2x^2 + x y^3 - \frac{7}{2}$$

```
b := PR(x*(2*t + z^3) - 6)
```

$$2tx + x z^3 - 6$$

```
c := a^2-b/3+3
```

$$-\frac{2tx}{3} + 4x^4 + 4x^3 y^3 + x^2 y^6 - 14x^2 - 7x y^3 - \frac{x z^3}{3} + \frac{69}{4}$$

Parameters

R

A commutative ring, i.e., a domain of category `Cat::CommutativeRing`. Default: `Dom::ExpressionField(normal)`.

Order

A monomial ordering, i.e., one of the predefined orderings `LexOrder`, `DegreeOrder`, or `DegInvLexOrder` or an element of the domain `Dom::MonomOrdering`. Default: `LexOrder`.

p

A polynomial or a polynomial expression.

`lm`

List of monomials, which are represented as lists containing the coefficients together with the exponents or exponent vectors.

`v`

List of indeterminates.

Entries

"characteristic"	The characteristic of this domain, which is the characteristic of `R`.
"coeffRing"	The coefficient ring of this domain as defined by the parameter `R`.
"key"	The name of the domain created.
"one"	The neutral element w.r.t. `"_mult"`, which is `R::one`.
"ordering"	The monomial order as defined by the parameter `Order`.
"zero"	The neutral element w.r.t. `"_plus"`, which is `R::zero`.

Algorithms

To create polynomials from expressions with no suitable indeterminates the dummy variable `_dummy` is introduced. With this variable it is possible to create elements from constants which otherwise would fail. The drawback of this approach is that two mathematically equal polynomials may have variable lists which differ by this dummy variable.

See Also

MuPAD Domains

`Dom::DistributedPolynomial` | `Dom::MultivariatePolynomial` | `Dom::UnivariatePolynomial`

Dom::Product

Homogeneous direct products

Syntax

Domain Creation

```
Dom::Product(Set, <n>)
```

```
Dom::ProductSetn(e1, e2, …, en)
```

```
Dom::ProductSetn(List)
```

Description

`Dom::Product(Set, n)` is an `n`-fold direct product of the domain *Set*.

`Dom::Product(Set, n)(e1, e2, ..., en)` creates the `n`-tuple (e_1, e_2, …, e_n).

The objects `e1`, `e2`, ..., `en` must be convertible into elements of the domain `Set`, otherwise an error message is issued.

`Dom::Product(Set, n)(List)` creates the `n`-tuple (l_1, l_2, …, l_n).

The `n` elements l_i of `List` must be convertible into elements of the domain `Set`, otherwise an error message is issued.

The list must consist of exactly `n` elements, otherwise an error message is issued.

Following to the definition of a direct product many of the methods such as `"D"` and `"_negate"` just map the operation to all the components of the tuple.

Most `n`-ary methods like `"_plus"` and `"_mult"` apply the operation component-wise to the tuples.

Superdomain

`Dom::BaseDomain`

Axioms

If Set has `Ax::canonicalRep`, then `Ax::canonicalRep`.

If Set has `Cat::AbelianMonoid`, then `Ax::normalRep`.

Categories

`Cat::HomogeneousFiniteProduct(Set)`

Examples

Example 1

Define the 3-fold direct product of the rational numbers:

```
P3 := Dom::Product(Dom::Rational, 3)
```

$$\text{Dom::Product}(\text{Dom::Rational}, 3)$$

and create elements:

```
a := P3([1, 2/3, 0])
```

$$\left[1, \frac{2}{3}, 0\right]$$

```
b := P3(2/3, 4, 1/2)
```

$$\left[\frac{2}{3}, 4, \frac{1}{2}\right]$$

We use the standard arithmetical operators to calculate with such tuples:

```
a + b, a*b, 2*a
```

$$\left[\frac{5}{3}, \frac{14}{3}, \frac{1}{2}\right], \left[\frac{2}{3}, \frac{8}{3}, 0\right], \left[2, \frac{4}{3}, 0\right]$$

Some system functions were overloaded for such elements, such as `diff`, `map` or `zip` (see the description of the corresponding methods `"diff"`, `"map"` and `"zip"` above).

For example, to divide each component of `a` by 2 we enter:

```
map(a, `/`, 2)
```

$$\left[\frac{1}{2}, \frac{1}{3}, 0\right]$$

The quoted character `` `/` `` is another notation for the function `_divide`, the functional form of the division operator `/`.

Be careful that the mapping function returns elements of the domain the product is defined over. This is not checked by the function `map` (for efficiency reasons) and may lead to "invalid" tuples. For example:

```
b := map(a, sin); domtype(b)
```

$$\left[\sin(1), \sin\left(\frac{2}{3}\right), 0\right]$$

$$\text{Dom::Product}(\text{Dom::Rational}, 3)$$

But the components of `b` are no longer rational numbers!

Parameters

`Set`

An arbitrary domain of elements, i.e., a domain of category `Cat::BaseCategory`

`n`

The dimension of the product (a positive integer); default is `1`

`e1e2, en, …`

Elements of `Set` or objects convertible into such

`List`

A list of `n` elements of `Set` or objects convertible into such

Entries

"dimen"	is the dimension of `Dom::Product(Set, n)`, which is equal to `n`.
"coeffRing"	is the domain `S`.
"one"	is the `n`-tuple `(Set::one, Set::one, ..., Set::one)`. This entry only exists if `Set` is a monoid, i.e., a domain of category `Cat::Monoid`.
"zero"	is the `n`-tuple `(Set::zero, Set::zero, ..., Set::zero)`. This entry only exists if `Set` is an Abelian group, i.e., a domain of category `Cat::AbelianGroup`.

Methods

Mathematical Methods

`_divide` — Divide tuples

```
_divide(x, y)
```

This method only exists if `Set` is a (multiplicative) group, i.e., a domain of category `Cat::Group`.

This method overloads the function `_divide` for `n`-tuples, i.e., one may use it in the form `x / y`, or in functional notation: `_divide(x, y)`.

`_invert` — Compute the inverse of a tuple

```
_invert(x)
```

This method only exists if `Set` is a (multiplicative) group, i.e., a domain of category `Cat::Group`.

This method overloads the function `_invert` for `n`-tuples, i.e., one may use it in the form `1/x` or `x^(-1)`, or in functional notation: `_inverse(x)`.

`_less` — Less-than relation

```
_less(x, y)
```

An implementation is provided only if `Set` is an ordered set, i.e., a domain of category `Cat::OrderedSet`.

This method overloads the function `_less` for `n`-tuples, i.e., one may use it in the form `x < y`, or in functional notation: `_less(x, y)`.

`_mult` — Multiplie tuples by tuples and scalars

```
_mult(x, y, …)
```

If `x` is not of the type `Dom::Product(Set,n)`, it is considered as a scalar which is multiplied to each component of the `n`-tuple `y` (and vice versa).

This method only exists if `Set` is a semigroup, i.e., a domain of category `Cat::SemiGroup`.

This method also handles more than two arguments. In this case, the argument list is split into two parts of the same length which both are multiplied with the function `_mult`. These two result are multiplied again with `_mult` whose result then is returned.

This method overloads the function `_mult` for `n`-tuples, i.e., one may use it in the form `x * y`, or in functional notation: `_mult(x, y)`.

`_negate` — Negate an n-tuple

```
_negate(x)
```

This method overloads the function `_negate` for `n`-tuples, i.e., one may use it in the form `-x`, or in functional notation: `_negate(x)`.

`_power` — ith power of a tuple

```
_power(x, i)
```

An implementation is provided only if `Set` is a semigroup, i.e., a domain of category `Cat::SemiGroup`.

This method overloads the function `_power` for `n`-tuples, i.e., one may use it in the form `x^i`, or in functional notation: `_power(x, i)`.

`_plus` — Add tuples

```
_plus(x, y, …)
```

The sum of two `n`-tuples x and y is defined component-wise as $(x_1 + y_1, \ldots, x_n + y_n)$.

This method overloads the function `_plus` for `n`-tuples, i.e., one may use it in the form `x + y`, or in functional notation: `_plus(x, y)`.

`D` — Differential operator

```
D(x)
```

An implementation is provided only if `Set` is a partial differential ring, i.e., a domain of category `Cat::PartialDifferentialRing`.

This method overloads the operator `D` for `n`-tuples, i.e., one may use it in the form `D(x)`.

`diff` — Differentiation of n-tuples

```
diff(a, x)
```

This method overloads the function `diff` for `n`-tuples, i.e., one may use it in the form `diff(a, x)`.

An implementation is provided only if `Set` is a partial differential ring, i.e., a domain of category `Cat::PartialDifferentialRing`.

`equal` — Test on equality of n-tuples

```
equal(x, y)
```

`intmult` — Multiple of a tuple

```
intmult(x, k)
```

An implementation is provided only if `Set` is an Abelian semigroup, i.e., a domain of category `Cat::AbelianSemiGroup`.

`iszero` — Test on zero

```
iszero(x)
```

Note that there may be more than one representation of the zero `n`-tuple if `R` does not have `Ax::canonicalRep`.

This method overloads the function `iszero` for `n`-tuples, i.e., one may use it in the form `iszero(x)`.

`random` — Random tuple generation

```
random()
```

Access Methods

`_index` — Tuple indexing

```
_index(x, i)
```

See also the method `"op"`.

This method overloads the function `_index` for `n`-tuples, i.e., one may use it in the form `x[i]`, or in functional notation: `_index(x, i)`.

`map` — Apply a function to tuple components

```
map(x, func, <expr, …>)
```

Note: Note that the function values will *not* be implicitly converted into elements of the domain `Set`. One has to take care that the function calls return elements of the domain type `Set`.

This method overloads the function `map` for `n`-tuples, i.e., one may use it in the form `map(x, func, ...)`.

`mapCanFail` — Apply a function to tuple components

```
mapCanFail(x, func, <expr, …>)
```

op — Component of a tuple

```
op(x, i)
```

```
op(x)
```

See also the method "_index".

This method overloads the function op for n-tuples, i.e., one may use it in the form op(x, i).

Returns a sequence of all components of x.

set_index — Assigning tuple components

```
set_index(x, i, e)
```

See also the method "subsop".

Note: This method does not check whether e has the correct type.

This method overloads the indexed assignment _assign for n-tuples, i.e., one may use it in the form x[i] := e, or in functional notation: _assign(x[i], e).

sort — Sorting the components of a tuple

```
sort(x)
```

This method overloads function sort for tuples, i.e. one may use it in the form sort(x).

subs — Substitution of tuple components

```
subs(x, …)
```

Note: The objects obtained by the substitutions will not be implicitly converted into elements of the domain Set. One has to take care that the substitutions return elements of the domain Set.

This method overloads the function subs for n-tuples, i.e., one may use it in the form subs(x, ...). See subs for details and calling sequences.

`testEach` — Check every component for a certain condition

```
testEach(x, func, <expr, …>)
```

`func` must return either `TRUE` or `FALSE`, otherwise a runtime error is raised.

`testOne` — Check an component for a certain condition

```
testOne(x, func, <expr, …>)
```

`func` must return either `TRUE` or `FALSE`, otherwise a runtime error is raised.

`zip` — Combine tuples component-wise

```
zip(x, y, func, <expr, …>)
```

Note: The function values will not be implicitly converted into elements of the domain `Set`. One has to take care that the function calls return elements of the domain `Set`.

This method overloads the function `zip` for `n`-tuples, i.e., one may use it in the form `zip(x, y, func, ...)`.

`zipCanFail` — Combine tuples component-wise

```
zipCanFail(x, y, func, <expr, …>)
```

Conversion Methods

`convert` — Conversion into an n-tuple

```
convert(List)
```

```
convert(e1, <e2, …>)
```

`FAIL` is returned if this conversion fails.

Tries to convert the arguments into an element of the domain `Dom::Product(Set, n)`. This can be done if exactly `n` arguments are given where each argument can be converted into an element of the domain `Set`.

FAIL is returned if this conversion fails.

expr — Conversion into an object of a kernel domain

```
expr(x)
```

This method overloads the function expr for n-tuples, i.e., one may use it in the form expr(x).

Dom::Quaternion

Skew field of quaternions

Syntax

```
Dom::Quaternion(listi)
```

```
Dom::Quaternion(ex)
```

```
Dom::Quaternion(M)
```

Description

Domain `Dom::Quaternion` represents the skew field of quaternions.

Quaternions are usually defined to be complex 2 ×2 matrices of the special form

$$\begin{pmatrix} a+b\,\mathrm{i} & -c-d\,\mathrm{i} \\ c-d\,\mathrm{i} & a-b\,\mathrm{i} \end{pmatrix},$$

where a, b, c, d are real numbers. Another usual notation is $a + bi + cj + dk$; the subfield of those quaternions for which $c = d = 0$ is isomorphic to the field of complex numbers.

The domain `Dom::Quaternion` regards these fields as being identical, and it allows both notations that have been mentioned, as well as simply `[a,b,c,d]`.

If you enter a quaternion as an arithmetical expression `ex`, the identifiers `i`, `j`, and `k` are understood in the way mentioned above; `I`, `J`, and `K` may be used alternatively, and you may also mix small and capital letters. Every subexpression of `ex` not containing one of these must be real and constant.

Note: Be sure that you have not assigned a value to one of the identifiers mentioned.

`Dom::Quaternion` has the domain `Dom::BaseDomain` as its super domain, i.e., it inherits each method which is defined by `Dom::BaseDomain` and not re-

implemented by `Dom::Quaternion`. Methods described below are re-implemented by `Dom::Quaternion`.

Superdomain

`Dom::BaseDomain`

Axioms

`Ax::canonicalRep`

Categories

`Cat::SkewField`

Examples

Example 1

Creating some quaternions.

```
Dom::Quaternion([1,2,3,4]),
Dom::Quaternion(11+12*i+13*j+14*k);
M := Dom::Matrix(Dom::Complex)([[3+4*I,-6-2*I],[6-2*I,3-4*I]]):
M, Dom::Quaternion(M)
```

$$1+2\,\mathrm{i}+3\,j+4\,k,\ 11+12\,\mathrm{i}+13\,j+14\,k$$

$$\begin{pmatrix} 3+4\,\mathrm{i} & -6-2\,\mathrm{i} \\ 6-2\,\mathrm{i} & 3-4\,\mathrm{i} \end{pmatrix},\ 3+4\,\mathrm{i}+6\,j+2\,k$$

Example 2

Doing some standard arithmetic.

```
a:=Dom::Quaternion([1,2,3,4]):
b:=Dom::Quaternion([11,2,33.3,2/3]):
a*b, a+b, a^2/3, b^3;
```

$$-95.56666667 - 107.2\,\mathrm{i} + 72.96666667\,j + 105.2666667\,k,\ 12 + 4\,\mathrm{i} + 36.3\,j + \frac{14\,k}{3},$$

$$-\frac{28}{3} + \frac{4\,\mathrm{i}}{3} + 2\,j + \frac{8\,k}{3},\ -35409.03667 - 1500.668889\,\mathrm{i} - 24986.137\,j - 500.222963\,k$$

Example 3

More mathematical operations:

```
a:=Dom::Quaternion([1,2,3,4]):
b:=Dom::Quaternion([11,2,33.3,2/3]):
Dom::Quaternion::nthroot(b,3);
abs(a), sign(b)
```

$$2.993953193 + 0.07959236197\,\mathrm{i} + 1.325212827\,j + 0.02653078732\,k$$

$$\sqrt{30},\ 0.3130950929 + 0.05692638053\,\mathrm{i} + 0.9478242358\,j + 0.01897546018\,k$$

Example 4

Some miscellaneous operations.

```
a:=Dom::Quaternion([1,2,3,4]):
Dom::Quaternion::matrixform(a);
map(a, sqrt), map(a, _plus, 1);
```

$$\begin{pmatrix} 1+2\,\mathrm{i} & -3-4\,\mathrm{i} \\ 3-4\,\mathrm{i} & 1-2\,\mathrm{i} \end{pmatrix}$$

$$1 + \sqrt{2}\,\mathrm{i} + \sqrt{3}\,j + 2\,k,\ 2 + 3\,\mathrm{i} + 4\,j + 5\,k$$

Parameters

`listi`

A list containing four elements of type `Type::Real`

`ex`

Arithmetical expression

`M`

A matrix of type `Dom::Matrix(Dom::Complex)`. It has to be of a special form described in the Details section.

Entries

"characteristic"	the characteristic of this domain is 0
"one"	the unit element; it equals `Dom::Quaternion([1,0,0,0])`.
"size"	the number of quaternions is `infinity`.
"zero"	The zero element; it equals `Dom::Quaternion([0,0,0,0])`.

Methods

Mathematical Methods

`_mult` — Multiplie quaternions

```
_mult(x, y, …)
```

`_plus` — Add quaternions

```
_plus(x, y, …)
```

`_power` — n-th power of a quaternion

```
_power(x, n)
```

`Im` — Return the imaginary (vectorial) part of a quaternion.

```
Im(x)
```

The result is still a quaternion.

`Re` — Return the real part of a quaternion.

```
Re(x)
```

The result is of type `Type::Real`.

`abs` — Absolute value of a quaternion

```
abs(x)
```

The result is of type `Type::Real`.

`conjugate` — Conjugate element

```
conjugate(x)
```

`intpower` — Multiplie quaternions

```
intpower(x, {DOM_INT})
```

The implementation uses “repeated squaring”.

`Dom::Quaternion` is used by `"_power"`.

`nthroot` — N-th root of a quaternion

```
nthroot(x, n)
```

The implementation uses “repeated squaring”.

`Dom::Quaternion` is used by `"_power"`.

`norm` — Norm of a quaternion

```
norm(x)
```

The result is of type `Type::Real`.

`random` — Random number generation

```
random()
```

`scalarmult` — Scalar multiplication

```
scalarmult(s, x)
```

`scalarprod` — Inner product

```
scalarprod(x, y)
```

`sign` — Sign of a quaternion

```
sign(x)
```

The result is of type `Type::Real`.

Conversion Methods

`convert` — Conversion of objects

```
convert(x)
```

`convert_to` — Conversion to other domains

```
convert_to(x, T)
```

It currently handles the following domains for `T`: `DOM_EXPR`, `DOM_LIST`, `Dom::Matrix(Dom::Complex)`.

`expr` — Convert a quaternion to an object of a kernel domain

```
expr(x)
```

The result is an object of the kernel domain `DOM_EXPR`.

This method overloads the function `expr` for quaternions, i.e., you may use it in the form `expr(x)`.

`matrixform` — Convert a quaternion to a 2 x 2 matrix with complex entries.

```
matrixform(x)
```

The result is an object of the domain `Dom::Matrix(Dom::Complex)`.

Technical Methods

`TeX` — Generate TeX-formatted string

```
TeX(x)
```

`map` — Apply a function to all components of a quaternion

```
map(x, f, arg, …)
```

If optional arguments are present, then each component `co` of `x` is replaced by f(co, arg...). So for the quaternion $x := a + bi + cj + dk$, `Dom::Quaternion(x, f, arg, ...)` returns the quaternion $f(a, arg, \ldots) + f(b, arg, \ldots)\, i + f(c, arg, \ldots)\, j + f(d, arg, \ldots)\, k$.

`simplify` — Simplification of a quaternion

```
simplify(x)
```

See Also

MuPAD Domains
`Dom::Complex`

Dom::Rational

Field of rational numbers

Syntax

```
Dom::Rational(x)
```

Description

`Dom::Rational` is the domain of rational numbers represented by elements of the domains `DOM_INT` or `DOM_RAT`. `Dom::Rational` represents the field of rational numbers.

Elements of `Dom::Rational` are usually not created explicitly. However, if one creates elements using the usual syntax, it is checked whether the input is of type `DOM_INT` or `DOM_RAT`. This means `Dom::Rational` is a façade domain which creates elements of domain type `DOM_INT` or `DOM_RAT`.

Viewed as a differential ring `Dom::Rational` is trivial, it contains constants only.

`Dom::Rational` has the domain `Dom::Numerical` as its super domain, i.e., it inherits each method which is defined by `Dom::Numerical` and not re-implemented by `Dom::Rational`. Methods described below are re-implemented by `Dom::Rational`.

Superdomain

`Dom::Numerical`

Axioms

`Ax::canonicalRep`, `Ax::systemRep`, `Ax::canonicalOrder`, `Ax::efficientOperation("_divide")`, `Ax::efficientOperation("_mult")`, `Ax::efficientOperation("_invert")`

Categories

`Cat::QuotientField(Dom::Integer)`, `Cat::DifferentialRing`, `Cat::OrderedSet`

Examples

Example 1

Creating some rational numbers using `Dom::Rational`. This example also shows that `Dom::Rational` is a façade domain.

```
Dom::Rational(2/3) ; domtype(%)
```

$$\frac{2}{3}$$

DOM_RAT

```
Dom::Rational(2.0)
```

```
Error: The arguments are invalid. [Dom::Rational::new]
```

Example 2

By tracing the method `Dom::Rational::testtypeDom` we can see the interaction between `testtype` and `Dom::Rational::testtypeDom`.

```
prog::trace(Dom::Rational::testtypeDom):
delete x:
testtype(x, Dom::Rational);
testtype(3/4, Dom::Rational);
prog::untrace(Dom::Rational::testtypeDom):
```

```
enter Dom::Rational::testtypeDom(x, Dom::Rational)
computed FAIL
```

FALSE

```
enter Dom::Rational::testtypeDom(3/4, Dom::Rational)
computed TRUE

    TRUE
```

Parameters

x

An integer or a rational number

Methods

Mathematical Methods

`denom` — Denominator of a rational number

```
denom(x)
```

`diff` — Differentiates

```
diff(z, <x, …>)
```

`numer` — Numerator of the rational number

```
numer(x)
```

`random` — Random number generation

```
random()
```

`retract` — Retract to an integer element

```
retract(x)
```

Conversion Methods

`convert` — Conversion of objects

```
convert(x)
```

`convert_to` — Conversion to other domains

```
convert_to(x, T)
```

The following domains are allowed for `T`: `DOM_INT`, `Dom::Integer`, `Dom::Rational`, `DOM_RAT`, `DOM_FLOAT`, `Dom::Float` and `Dom::Numerical`.

`testtype` — Type checking

```
testtype(x, T)
```

In general this method is called from the function `testtype` and not directly by the user. “Example 2” on page 7-361 demonstrates this behaviour.

See Also

MuPAD Domains

`Dom::Complex` | `Dom::Float` | `Dom::Numerical` | `Dom::Rational` | `Dom::Real`

Dom::Real

Field of real numbers

Syntax

```
Dom::Real(x)
```

Description

`Dom::Real` is the field of real numbers represented by elements of the kernel domains `DOM_INT`, `DOM_RAT`, `DOM_FLOAT`, and `DOM_EXPR`.

`Dom::Real` is the domain of real numbers represented by expressions of type `DOM_INT`, `DOM_RAT` or `DOM_FLOAT`. An expression of type `DOM_EXPR` is considered as a real number if it is of type `Type::Arithmetical` and if it contains no indeterminates which are not of type `Type::ConstantIdents` and if it contains no imaginary part. See "Example 2" on page 7-365.

`Dom::Real` has category `Cat::Field` due to practical reasons. This domain actually is not a field because `bool(1.0 = 1e100 + 1.0 - 1e100)` returns `FALSE` for example.

Elements may not have an unique representation, for example `bool(0 = sin(2)^2 + cos(2)^2 - 1)` returns `FALSE`.

Elements of `Dom::Real` are usually not created explicitly. However, if one creates elements using the usual syntax, it is checked whether the input expression can be converted to a number. This means `Dom::Real` is a façade domain which creates elements of domain type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT`, or `DOM_EXPR`.

`Dom::Real` has no normal representation, because `0` and `0.0` both represent zero.

Viewed as a differential ring, `Dom::Real` is trivial, it contains constants only.

`Dom::Real` has the domain `Dom::Complex` as its super domain, i.e., it inherits each method which is defined by `Dom::Complex` and not re-implemented by `Dom::Real`. Methods described below are re-implemented by `Dom::Real`.

Superdomain

`Dom::Complex`

Axioms

`Ax::systemRep`, `Ax::canonicalOrder`, `Ax::efficientOperation("_divide")`, `Ax::efficientOperation("_mult")`, `Ax::efficientOperation("_invert")`

Categories

`Cat::DifferentialRing`, `Cat::Field`, `Cat::OrderedSet`

Examples

Example 1

The following lines demonstrate how to generate elements of `Dom::Real`. The rational and the floating-point numbers are elements of the real numbers:

```
Dom::Real(2/3)
```

$$\frac{2}{3}$$

```
Dom::Real(0.5666)
```

$$0.5666$$

Example 2

The numbers `PI` and `sin(2)` are real numbers whereas `sin(2/3 * I) + 3` and `sin(x)` for general symbolic `x` are not real numbers. If we try to create the elements

`Dom::Real(sin(2/3 * I) + 3)` and `Dom::Real(sin(x))` an error message is produced.

```
Dom::Real(PI)
```

$$\pi$$

```
Dom::Real(sin(2))
```

$$\sin(2)$$

```
Dom::Real(sin(2/3 * I) + 3)

Error: The arguments are invalid. [Dom::Real::new]
Dom::Real(sin(x))

Error: The arguments are invalid. [Dom::Real::new]
```

Parameters

x

An expression of type `DOM_INT`, `DOM_RAT`, or `DOM_FLOAT`. An expression of type `DOM_EXPR` is also allowed if it is of type `Type::Arithmetical` and if it contains no indeterminates which are not of type `Type::ConstantIdents` and if it contains no imaginary part.

Methods

Mathematical Methods

`_less` — Boolean operator "less"

```
_less(x, y)
```

`_leequal` — Boolean operator "less or equal"

```
_leequal(x, y)
```

`_power` — Power operator

```
_power(z, n)
```

`conjugate` — Complex conjugate

```
conjugate(x)
```

`Im` — Imaginary part of a real number

```
Im(x)
```

`random` — Random number generation

```
random()
```

```
random(n)
```

```
random(m .. n)
```

This method returns a random number generator which creates positive integer between 0 and n - 1.

This method returns a random number generator which creates positive integer between m and n.

`Re` — Real part of a real number

```
Re(x)
```

Conversion Methods

`convert` — Conversion of objects

```
convert(x)
```

`convert_to` — Conversion to other domains

```
convert_to(x, T)
```

The following domains are allowed for T: DOM_INT, Dom::Integer, DOM_RAT, Dom::Rational, DOM_FLOAT, Dom::Float, Dom::Numerical, Dom::ArithmeticalExpression, Dom::Complex.

See Also

MuPAD Domains

Dom::Complex | Dom::Float | Dom::Integer | Dom::Numerical | Dom::Rational

Dom::SquareMatrix

Rings of square matrices

Syntax

Domain Creation

```
Dom::SquareMatrix(n, <R>)
```

Element Creation

```
Dom::SquareMatrix(n, R)(Array)

Dom::SquareMatrix(n, R)(Matrix)

Dom::SquareMatrix(n, R)(<n, n>)

Dom::SquareMatrix(n, R)(<n, n>, ListOfRows)

Dom::SquareMatrix(n, R)(<n, n>, f)

Dom::SquareMatrix(n, R)(<n, n>, List, <Diagonal>)

Dom::SquareMatrix(n, R)(<n, n>, g, <Diagonal>)

Dom::SquareMatrix(n, R)(<n, n>, List, <Banded>)
```

Description

Domain Creation

`Dom::SquareMatrix(n, R)` creates a domain which represents the ring of $n \times n$ matrices over a component domain `R`. The domain `R` must be of category `Cat::Rng` (a ring, possibly without unit).

If the optional parameter `R` is not given, the domain `Dom::ExpressionField()` is used as the component ring for the square matrices.

For matrices of a domain created by `Dom::SquareMatrix(n, R)`, standard matrix arithmetic is implemented by overloading the standard arithmetical operators `+, -, *, /` and `^`. All functions of the `linalg` package dealing with matrices can also be applied.

`Dom::SquareMatrix(n, R)` has the domain `Dom::Matrix(R)` as its super domain, i.e., it inherits each method which is defined by `Dom::Matrix(R)` and not re-implemented by `Dom::SquareMatrix(n, R)`.

Methods described below are re-implemented by `Dom::SquareMatrix`.

The domain `Dom::Matrix(R)` represents matrices over `R` of arbitrary size, and it therefore does not have any algebraic structure (except of being a *set* of matrices).

The domain `Dom::MatrixGroup(m, n, R)` represents the Abelian group of $m \times n$ matrices over R.

Element Creation

`Dom::SquareMatrix(n, R)(Array)` and `Dom::SquareMatrix(n, R)(Matrix)` create a new matrix formed by the entries of `Array` and `Matrix`, respectively.

The components of `Array` and `Matrix`, respectively, are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

The call `Dom::SquareMatrix(n, R)( n , n )` returns the $n \times n$ zero matrix. Note that the $n \times n$ zero matrix is also defined by the entry `"zero"` (see below).

`Dom::SquareMatrix(n, R)( n , n ListOfRows)` creates an $n \times n$ matrix with components taken from the nested list `ListOfRows`. Each inner list corresponds to a row of the matrix.

If an inner list has less than `n` entries, the remaining components in the corresponding row of the matrix are set to zero. If there are less than `n` inner lists, the remaining lower rows of the matrix are filled with zeroes.

The entries of the inner lists are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

It might be a good idea first to create a two-dimensional array from that list before calling `Dom::SquareMatrix(n, R)`. This is due to the fact that creating a matrix from

an array is the fastest way one can achieve. However, in this case the sublists must have the same number of elements.

`Dom::SquareMatrix(n, R)( n , n f)` returns the matrix whose (i, j)th component is the value of the function call `f(i, j)`. The row and column indices i and j range from 1 to `n`.

The function values are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

Superdomain

`Dom::Matrix(R)`

Axioms

If R has `Ax::canonicalRep`, then `Ax::canonicalRep`.

Categories

`Cat::SquareMatrix(R)`

Examples

Example 1

A lot of examples can be found on the help page of the domain constructor `Dom::Matrix`, and most of them are also examples for working with domains created by `Dom::SquareMatrix`.

These examples only concentrate on some differences with respect to working with matrices of the domain `Dom::Matrix(R)`.

The following command defines the ring of two-dimensional matrices over the rationals:

```
SqMatQ := Dom::SquareMatrix(2, Dom::Rational)
```

$$\text{Dom::SquareMatrix}(2, \text{Dom::Rational})$$

```
SqMatQ::hasProp(Cat::Ring)
```

$$\text{TRUE}$$

The unit is defined by the entry `"one"`, which is the 2×2 identity matrix:

```
SqMatQ::one
```

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Note that some operations defined by the domain `SqMatQ` return matrices which are no longer square. They return therefore matrices of the domain `Dom::Matrix(Dom::Rational)`, the super-domain of `SqMatQ`. For example, if we delete the first row of the matrix:

```
A := SqMatQ([[1, 2], [-5, 3]])
```

$$\begin{pmatrix} 1 & 2 \\ -5 & 3 \end{pmatrix}$$

we get the matrix:

```
SqMatQ::delRow(A, 1)
```

$$\begin{pmatrix} -5 & 3 \end{pmatrix}$$

which is of the domain type:

```
domtype(%)
```

$$\text{Dom::Matrix}(\text{Dom::Rational})$$

Example 2

We can convert a square matrix into or from another matrix domain, as shown next:

```
SqMatR := Dom::SquareMatrix(3, Dom::Real):
MatC := Dom::Matrix(Dom::Complex):

A := SqMatR((i, j) -> sin(i*j))
```

$$\begin{pmatrix} \sin(1) & \sin(2) & \sin(3) \\ \sin(2) & \sin(4) & \sin(6) \\ \sin(3) & \sin(6) & \sin(9) \end{pmatrix}$$

To convert `A` into a matrix of the domain `MatC`, enter:

```
coerce(A, MatC)
```

$$\begin{pmatrix} \sin(1) & \sin(2) & \sin(3) \\ \sin(2) & \sin(4) & \sin(6) \\ \sin(3) & \sin(6) & \sin(9) \end{pmatrix}$$

```
domtype(%)
```

$$\mathrm{Dom::Matrix(Dom::Complex)}$$

The conversion is done component-wise, as the following examples shows:

```
B := MatC([[0, 1], [exp(I), 0]])
```

$$\begin{pmatrix} 0 & 1 \\ e^{i} & 0 \end{pmatrix}$$

The matrix B is square but has one complex component and therefore cannot be converted into the domain `SqMatR`:

```
coerce(B, SqMatR)
```

FAIL

Parameters

`n`

A positive integer

`R`

A ring, i.e., a domain of category `Cat::Rng`; default is `Dom::ExpressionField()`

`Array`

An $n \times n$ array

`Matrix`

An $n \times n$ matrix, i.e., an element of a domain of category `Cat::Matrix`

`List`

A list of matrix components

`ListOfRows`

A list of at most `n` rows; each row is a list of at most `n` matrix components

`f`

A function or a functional expression with two parameters (the row and column index)

`g`

A function or a functional expression with one parameter (the row index)

Options

`Diagonal`

Create a diagonal matrix

With the option `Diagonal`, diagonal matrices can be created with diagonal elements taken from a list, or computed by a function.

`Dom::SquareMatrix(n, R)( n , n List, Diagonal)` creates the $n \times n$ diagonal matrix whose diagonal elements are the entries of `List`.

`List` must have at most `n` entries. If it has fewer elements, the remaining diagonal elements are set to zero.

The entries of `List` are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

`Dom::SquareMatrix(n, R)( n , n g, Diagonal)` returns the matrix whose ith diagonal element is `g(i)`, where the index i runs from 1 to `n`.

The function values are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

`Banded`

Create a banded Toeplitz matrix

`Dom::SquareMatrix(n, R)( n , n List, Banded)` creates an $n \times n$ banded Toeplitz matrix with the elements of `List` as entries. The number of entries of `List` must be odd, say $2h + 1$, and must not exceed `n`. The resulting matrix has bandwidth at most $2h + 1$.

All elements of the main diagonal of the created matrix are initialized with the middle element of `List`. All elements of the ith subdiagonal are initialized with the $(h + 1 - i)$th element of `List`. All elements of the ith superdiagonal are initialized with the $(h + 1 + i)$th element of `List`. All entries on the remaining sub- and superdiagonals are set to zero.

The entries of `List` are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

Entries

"one"	is the $n \times n$ identity matrix. This entry exists if the component ring `R` is a domain of category `Cat::Ring`, i.e., a ring with unit.

"randomDimen"	is set to `[n, n]`.
"zero"	is the $n \times n$ zero matrix.

Methods

Mathematical Methods

`evalp` — Evaluating matrices of polynomials at a certain point

```
evalp(A, x = a, …)
```

This method is only defined if `R` is a polynomial ring of category `Cat::Polynomial`.

This method overloads the function `evalp` for matrices, i.e., one may use it in the form `evalp(A, x = a)`.

`identity` — Identity matrix

```
identity(k)
```

Note: The matrix returned is of the domain `Dom::Matrix(R)` if $k \neq n$.

This method only exists if the component ring `R` is of category `Cat::Ring`, i.e., a ring with unit.

`matdim` — Matrix dimension

```
matdim(A)
```

`random` — Random matrix generation

```
random()
```

The components of the random matrix are generated with the method `"random"` of the component ring `R`.

Access Methods

`_concat` — Horizontal concatenation of matrices

```
_concat(A, B, …)
```

An error message is issued if the given matrices do not have the same number of rows.

Note: The matrix returned is of the domain `Dom::Matrix(R)`!

This method overloads the function `_concat` for matrices, i.e., one may use it in the form `A . B . ...`, or in functional notation: `_concat(A, B, ...)`.

`_index` — Matrix indexing

```
_index(A, i, j)
```

```
_index(A, r1 .. r2, c1 .. c2)
```

If `i` and `j` are not integers, then the call of this method returns in its symbolic form (of type `"_index"`) with evaluated arguments.

Otherwise an error message is given, if `i` and `j` are not valid row and column indices, respectively.

Note: Note that the system function `context` is used to evaluate the entry in the context of the calling environment.

Returns the submatrix of `A`, created by the rows of `A` with indices from `r1` to `r2` and the columns of `A` with indices from `c1` to `c2`.

Note: The submatrix returned is of the domain `Dom::Matrix(R)`!

This method overloads the function `_index` for matrices, i.e., one may use it in the form `A[i, j]` and `A[r1..r2, c1..c2]`, respectively, or in functional notation: `_index(A, ...)`.

concatMatrix — Horizontal concatenation of matrices

```
concatMatrix(A, B, …)
```

col — Extracting a column

```
col(A, c)
```

An error message is issued if `c` is less than one or greater than `n`.

delCol — Deleting a column

```
delCol(A, c)
```

`NIL` is returned if `A` only consists of one column.

Note: The matrix returned is of the domain `Dom::Matrix(R)`.

An error message is issued if `c` is less than one or greater than `n`.

delRow — Deleting a row

```
delRow(A, r)
```

`NIL` is returned if `A` only consists of one row.

Note: The matrix returned is of the domain `Dom::Matrix(R)`.

An error message is issued if `r` is less than one or greater than `n`.

row — Extracting a row

```
row(A, r)
```

An error message is issued if `r` is less than one or greater than `n`.

stackMatrix — Vertical concatenation of matrices

```
stackMatrix(A, B, …)
```

An error message is issued if the given matrices do not have the same number of columns.

Note: The matrix returned is of the domain `Dom::Matrix(R)`!

Conversion Methods

`create` — Defining matrices without component conversions

```
create(x, …)
```

This method should be used if the elements of the parameters `x`, ... are elements of the domain type `R`. This is often the case if a matrix is to be created whose components come from preceding matrix and scalar operations.

See Also

MuPAD Domains
`Dom::Matrix` | `Dom::MatrixGroup`

Dom::SymmetricGroup

Symmetric groups

Syntax

```
Dom::SymmetricGroup(n)
```

```
Dom::SymmetricGroup(n)(l)
```

Description

`Dom::SymmetricGroup(n)` creates the symmetric group of order n, that is, the domain of all the permutations of $\{1, \ldots, n\}$ elements.

A permutation of n elements is a bijective mapping of the set $\{1, \ldots, n\}$ onto itself.

The domain element `Dom::SymmetricGroup(n)(l)` represents the bijective mapping of the first n positive integers that maps the integer i to `l[i]`, for $1 \le i \le n$.

Superdomain

`Dom::BaseDomain`

Axioms

`Ax::canonicalRep`

Categories

`Cat::Group`

Examples

Example 1

Consider the group of permutations of the first seven positive integers:

```
G := Dom::SymmetricGroup(7)
```

```
Dom::SymmetricGroup(7)
```

We create an element of `G` by providing the image of 1, 2, etc.:

```
a:=G([2,4,6,1,3,5,7])
```

```
[2, 4, 6, 1, 3, 5, 7]
```

```
a(3)
```

```
6
```

Parameters

`n`

Positive integer

`1`

List or array consisting of the first n integers in some order.

Entries

"one"	the identical mapping of the set {1, …, n} to itself.

Methods

Mathematical Methods

`_mult` — Product of permutations

```
_mult(a1, …)
```

This method overloads the function `_mult`.

`_invert` — Inverse of a permutation

```
_invert(a)
```

This method overloads the function `_invert`.

`func_call` — Function value of a permutation at a point

```
func_call(a, i)
```

It computes the function value of `a` at `i`, i.e., the integer that `i` is mapped to by the permutation `a`; `i` must be an integer between 1 and n.

`cycles` — Cycle representation of a permutation

```
cycles(a)
```

`order` — Order of a permutation

```
order(a)
```

`inversions` — Number of inversions

```
inversions(a)
```

`sign` — Sign of a permutation

```
sign(a)
```

`random` — Random permutation

```
random()
```

Access Methods

`allElements` — Return all elements of the group

```
allElements()
```

`size` — Return the size of the group

```
size()
```

Conversion Methods

`convert` — Conversion of an object into a permutation

```
convert(x)
```

`convert_to` — Conversion of a permutation into another type

```
convert_to(a, T)
```

`expr` — Convert a permutation into a list

```
expr(a)
```

Dom::UnivariatePolynomial

Domains of univariate polynomials

Syntax

Domain Creation

```
Dom::UnivariatePolynomial(<Var, <R, <Order>>>)
```

Element Creation

```
Dom::UnivariatePolynomial(Var, R, Order)(p)
```

```
Dom::UnivariatePolynomial(Var, R, Order)(lm)
```

Description

`Dom::UnivariatePolynomial(Var, R, ..)` creates the domain of univariate polynomials in the variable `Var` over the commutative ring `R`.

`Dom::UnivariatePolynomial` represents univariate polynomials over arbitrary commutative rings.

All usual algebraic and arithmetical polynomial operations are implemented, including Gröbner basis computations.

`Dom::UnivariatePolynomial(Var, R, Order)` creates a domain of univariate polynomials in the variable `Var` over a domain of category `Cat::CommutativeRing` in sparse representation with respect to the monomial ordering `Order`.

`Dom::UnivariatePolynomial()` creates the univariate polynomial domain in the variable `x` over the domain `Dom::ExpressionField(normal)` with respect to the lexicographic monomial ordering.

`Dom::UnivariatePolynomial(Var)` creates the univariate polynomial domain in the variable `Var` over the domain `Dom::ExpressionField(normal)` with respect to the lexicographic monomial ordering.

Note: Only commutative coefficient rings of type `DOM_DOMAIN` which inherit from `Dom::BaseDomain` are allowed. If `R` is of type `DOM_DOMAIN` but inherits not from `Dom::BaseDomain`, the domain `Dom::ExpressionField(normal)` will be used instead.

For this domain only identifiers are valid variables.

Note: It is highly recommend to use only coefficient rings with unique zero representation. Otherwise it may happen that, e.g., a polynomial division will not terminate or a wrong degree will be returned.

Please note that for reasons of efficiency not all methods check their arguments, not even at the interactive level. In particular this is true for many access methods, converting methods and technical methods. Therefore, using these methods inappropriately may result in strange error messages.

Superdomain

`Dom::MultivariatePolynomial`

Axioms

If R has `Ax::normalRep`, then `Ax::normalRep`.

If R has `Ax::canonicalRep`, then `Ax::canonicalRep`.

Categories

`Cat::UnivariatePolynomial(R)`

Examples

Example 1

To create the ring of univariate polynomials in x over the integers one may define

```
UP:=Dom::UnivariatePolynomial(x,Dom::Integer)
```

$$\text{Dom::UnivariatePolynomial}(x, \text{Dom::Integer}, \text{LexOrder})$$

Now, let us create two univariate polynomials.

```
a:=UP((2*x-1)^2*(3*x+1))
```

$$12\,x^3 - 8\,x^2 - x + 1$$

```
b:=UP(((2*x-1)*(3*x+1))^2)
```

$$36\,x^4 - 12\,x^3 - 11\,x^2 + 2\,x + 1$$

The usual arithmetical operations for polynomials are available:

```
a^2+a*b
```

$$432\,x^7 - 288\,x^6 - 264\,x^5 + 200\,x^4 + 35\,x^3 - 36\,x^2 - x + 2$$

The leading coefficient, leading term, leading monomial and reductum of a are

```
lcoeff(a),lterm(a),lmonomial(a),UP::reductum(a)
```

$$12,\ x^3,\ 12\,x^3,\ -8\,x^2 - x + 1$$

and a is of degree

```
degree(a)
```

$$3$$

The method `gcd` computes the greatest common divisor of two polynomials

```
gcd(a,b)
```

$$12\,x^3 - 8\,x^2 - x + 1$$

and `lcm` the least common multiple:

```
lcm(a,b)
```

$$36\,x^4 - 12\,x^3 - 11\,x^2 + 2\,x + 1$$

Computing the definite and indefinite integral of a polynomial is also possible,

```
int(a)
```

$$3\,x^4 - \frac{8\,x^3}{3} - \frac{x^2}{2} + x$$

which is in the case of indefinite integration simply the antiderivative of the polynomial.

```
D(int(a)), domtype(D(int(a)))
```

$$12\,x^3 - 8\,x^2 - x + 1,\ \text{Dom::UnivariatePolynomial}(x,\ \text{Dom::Fraction}(\text{Dom::Integer}),\ \text{LexOrder})$$

But, since for representing the indefinite integral of `a` the coefficient ring chosen as the integers is not appropriate, the polynomial ring over its quotient field is used instead.

Furthermore, interpreting the polynomials as polynomial functions is also allowed in applying coefficient ring elements, polynomials of this domain or arbitrary expressions with option `Expr` to them:

```
a(5)
```

$$1296$$

```
a(b)
```

$$559872\,x^{12} - 559872\,x^{11} - 326592\,x^{10} + 414720\,x^9 + 73872\,x^8 - 123120\,x^7 - 9924\,x^6 + 18408\,x^5 + 1144\,x^4 - 1364\,x^3 - 97\,x^2 + 38\,x + 4$$

```
a(sin(x),Expr)
```

$$12\,\sin(x)^3 - 8\,\sin(x)^2 - \sin(x) + 1$$

To get a vector of coefficients of a polynomial, which gives the dense representation of it, one may use the method `vectorize`.

```
UP::vectorize(a), UP::vectorize(a,6)
```

$$[1, -1, -8, 12], [1, -1, -8, 12, 0, 0]$$

Parameters

`Var`

An indeterminate given by an identifier; default is `x`.

`R`

A commutative ring, i.e. a domain of category `Cat::CommutativeRing`; default is `Dom::ExpressionField(normal)`.

`Order`

A monomial ordering, i.e. one of the predefined orderings `LexOrder`, `DegreeOrder` or `DegInvLexOrder` or an element of domain `Dom::MonomOrdering`; default is `LexOrder`.

`p`

A polynomial or a polynomial expression.

`lm`

List of monomials, which are represented as lists containing the coefficients together with the exponents or exponent vectors.

Entries

"characteristic"	The characteristic of this domain.
"coeffRing"	The coefficient ring of this domain as defined by the parameter `R`.
"key"	The name of the domain created.
"one"	The neutral element w.r.t. `"_mult"`.
"ordering"	The monomial order as defined by the parameter `Order`.
"variables"	The list of the variable as defined by the parameter `Var`.
"zero"	The neutral element w.r.t. `"_plus"`.

Methods

Access Methods

`coeff` — Coefficient of a polynomial

```
coeff(a)
```

```
coeff(a, Var, n)
```

```
coeff(a, n)
```

`coeff(a, Var, n)` returns the coefficient of the term `Var^n` as an element of `R`.

`coeff(a, n)` returns the coefficient of the term `Var^n` as an element of `R`.

This method overloads the function `coeff` for polynomials.

`degree` — Degree of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`degreevec` — Vector of exponents of the leading term of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`euclideanDegree` — Euclidean degree function

Inherited from `Dom::DistributedPolynomial`.

`ground` — Ground term of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`has` — Existence of an object in a polynomial

Inherited from `Dom::DistributedPolynomial`.

`indets` — Indeterminate of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`lcoeff` — Leading coefficient of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`ldegree` — Lowest degree of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`lmonomial` — Leading monomial of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`lterm` — Leading term of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`mainvar` — Main variable of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`mapcoeffs` — Applie a function to the coefficients of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`multcoeffs` — Multiplie the coefficients of a polynomial with a factor

Inherited from `Dom::DistributedPolynomial`.

`nterms` — Number of terms of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`nthcoeff` — N-th coefficient of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`nthmonomial` — N-th monomial of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`nthterm` — N-th term of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`order` — Compare two polynomials w.r.t. a given order

Inherited from `Dom::MultivariatePolynomial`.

`orderedVariableList` — Ordered list of indeterminates of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`pivotSize` — Size of a pivot element

Inherited from `Dom::DistributedPolynomial`.

`reductum` — Reductum of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`sortList` — Sort a list of polynomials w.r.t. a given order

Inherited from `Dom::MultivariatePolynomial`.

`stableSort` — Sort a list of polynomials w.r.t. a given order

Inherited from `Dom::MultivariatePolynomial`.

`subs` — Avoid substitution

Inherited from `Dom::BaseDomain`.

`subsex` — Avoid extended substitution

Inherited from `Dom::BaseDomain`.

`tcoeff` — Lowest coefficient of a polynomial

Inherited from `Dom::DistributedPolynomial`.

`vectorize` — Vectorized form of a polynomial

```
vectorize(a, <n>)
```

See Also

MuPAD Domains

`Dom::DistributedPolynomial` | `Dom::MultivariatePolynomial` | `Dom::Polynomial`

Factored

Objects kept in factored form

Syntax

```
Factored(list, <type>, <ring>)
```

```
Factored(f, <type>, <ring>)
```

Description

`Factored` is the domain of objects kept in factored form, such as prime factorization of integers, square-free factorization of polynomials, or the factorization of polynomials in irreducible factors.

The argument `list` must be a list of odd length and of the form `[u, f1, e1, f2, e2, ..., fr, er]`, where the entries u and f_i are elements of the domain `ring`, or can be converted into such elements. The e_i must be integers. Here, i ranges from 1 to r.

See section “Operands” below for the meaning of the entries of that list.

An error message is reported, if one of the list entries is of wrong type.

An arithmetical expression `f` given as the first argument is the same as giving the list `[ring::one, f, 1]`.

See section “Operands” below for the meaning of the entries of that list.

`f` must be an element of the domain `ring`, or must be convertible into such an element, otherwise an error message would be given.

The argument `type` indicates what is known about the factorization. Currently, the following types are known:

- `"unknown"` – nothing is known about the factorization.
- `"irreducible"` – the f_i are irreducible over the domain `ring`.

- `"squarefree"` – the f_i are square-free over the domain `ring`.

If this argument is missing, then the type of the created factored object is set to `"unknown"`.

The type of factorization is known to any element of `Factored`. Use the methods `"getType"` and `"setType"` (see below) to read and set the type of factorization of a given factored object.

The argument `ring` is the ring of factorization. It must be an integral domain, i.e., a domain of category `Cat::IntegralDomain`.

If this argument is missing, then the domain `Dom::ExpressionField()` is used.

The ring of factorization is known to any element of `Factored`. Use the methods `"getRing"` and `"setRing"` (see below) to read and set the ring of factorization of a given factored object.

You can use the index operator `[ ]` to extract the factors of an element `f` of the domain `Factored`. E.g., for $f = u\, f_1^{e_1} f_2^{e_2} \ldots$, you have $f[1] = u$, $f[2] = f_1^{e_1}$, $f[3] = f_2^{e_2}$ etc.

You can also use the methods `"factors"` and `"exponents"` (see below) to access the operands, i.e., the call `Factored::factors(f)` returns a list of the factors f_i, and `Factored::exponents(g)` returns a list of the exponents e_i ($1 \le i \le r$).

The system functions `ifactor`, `factor` and `polylib::sqrfree` are the main application of this domain, they return their result in form of such factored objects (see their help pages for information about the type and ring of factorization).

There may be no need to explicitly create factored objects, but to work with the results of the mentioned system functions.

Note that an element of `Factored` is printed like an expression and behaves like that. As an example, the result of `f := factor(x^2 + 2*x + 1)` is an element of `Factored` and printed as `(x + 1)^2`. The call `type(f)` returns `"_power"` as the expression type of `f`.

For an element `f` of `Factored`, the call `Factored::convert(f, DOM_LIST)` gives a list of all operands of `f`.

Examples

Example 1

The following computes the prime factorization of the integer 20:

```
f := ifactor(20)
```

$$2^2\,5$$

The result is an element of the domain `Factored`:

```
domtype(f)
```

Factored

which consists of the following five operands:

```
op(f)
```

1, 2, 2, 5, 1

They represent the integer 20 in the following form: $20 = 1\,2^2\,5$. The factors are prime numbers and can be extracted via `Factor::factors`:

```
Factored::factors(f)
```

[2, 5]

`ifactor` kept the information that the factorization ring is the ring of integers (represented by the domain `Dom::Integer`), and that the factors of `f` are prime (and therefore irreducible, because $\mathbb{Z}$ is an integral domain):

```
Factored::getRing(f), Factored::getType(f)
```

Dom::Integer, "irreducible"

We can convert such an object into different forms, such as into a list of its operands:

```
Factored::convert_to(f, DOM_LIST)
```

$$[1, 2, 2, 5, 1]$$

or into an unevaluated expression, keeping the factored form:

```
Factored::convert_to(f, DOM_EXPR)
```

$$2^2\,5$$

or back into an integer:

```
Factored::convert_to(f, Dom::Integer)
```

$$20$$

You may also use the system function `coerce` here, which has the same effect.

Example 2

We compute the factorization of the integers 108 and 512:

```
n1 := ifactor(108); n2 := ifactor(512)
```

$$2^2\,3^3$$

$$2^9$$

The multiplication of these two integers gives the prime factorization of 55296 = 108 512:

```
n1*n2
```

$$2^{11}\,3^3$$

Note that the most operations on such objects lead to an un-factored form, such as adding these two integers:

```
n1 + n2
```

$$620$$

You may apply the function `ifactor` to the result, if you are interested in its prime factorization:

```
ifactor(%)
```

$$2^2\ 5\ 31$$

You an apply (almost) each function to factored objects, functions that mainly expect arithmetical expressions as their input. Note that, before the operation is applied, the factored object is converted into an arithmetical expression in un-factored form:

```
Re(n1)
```

$$108$$

Example 3

The second system function which deals with elements of `Factored`, is `factor`, which computes all irreducible factors of a polynomial.

For example, if we define the following polynomial of $\mathbb{Z}_{101}$:

```
p := poly(x^12 + x + 1, [x], Dom::IntegerMod(101)):
```

and compute its factorization into irreducible factors, we get:

```
f := factor(p)
```

$$\text{poly}(x^2 + 73\,x + 29, [x], \text{Dom::IntegerMod}(101))$$

$$\text{poly}(x^5 + 62\,x^4 + 64\,x^3 + 63\,x^2 + 58\,x + 100, [x], \text{Dom::IntegerMod}(101))$$

$$\text{poly}(x^5 + 67\,x^4 + 72\,x^3 + 100\,x^2 + 33\,x + 94, [x], \text{Dom::IntegerMod}(101))$$

If we multiply the factored object with an element that can be converted into an element of the ring of factorization, then we get a new factored object, which then is of the factorization type `"unknown"`:

```
x*f
```

$$\text{poly}(x^2 + 73\,x + 29,\ [x],\ \text{Dom::IntegerMod}(101))$$

$$\text{poly}(x^5 + 62\,x^4 + 64\,x^3 + 63\,x^2 + 58\,x + 100,\ [x],\ \text{Dom::IntegerMod}(101))$$

$$\text{poly}(x^5 + 67\,x^4 + 72\,x^3 + 100\,x^2 + 33\,x + 94,\ [x],\ \text{Dom::IntegerMod}(101))\ x$$

```
Factored::getType(%)
```

"unknown"

You may use the function `expand` which returns the factored object in expanded form as an element of the factorization ring:

```
expand(f)
```

$$\text{poly}(x^{12} + x + 1,\ [x],\ \text{Dom::IntegerMod}(101))$$

Example 4

The third system function which return elements of `Factored` is `polylib::sqrfree`, which computes the square-free factorization of polynomials. For example:

```
f := polylib::sqrfree(x^2 + 2*x + 1)
```

$$(x + 1)^2$$

The factorization type, of course, is `"squarefree"`:

```
Factored::getType(f)
```

"squarefree"

Parameters

`list`

A list of odd length

`f`

An arithmetical expression

`type`

A string (default: `"unknown"`)

`ring`

A domain of category `Cat::IntegralDomain` (default: `Dom::ExpressionField()`)

Function Calls

Calling a factored object as a function yields the object itself, regardless of the arguments. The arguments are *not* evaluated.

Operations

You can apply (almost) every function to factored objects, functions that mainly expect arithmetical expressions as their input.

For example, one may add or multiply those objects, or apply functions such as `expand` and `diff` to them. But the result of such an operation then is usually not any longer of the domain `Factored`, as the factored form could be lost due to the operation (see examples below).

Call `expr(f)` to convert the factored object `f` into an arithmetical expression (as an element of a kernel domain).

The call `coerce(f, DOM_LIST)` returns a list of operands of the factored object `f` (see method `"convert_to"` below).

Operands

An element f of `Factored` consists of the $r + 1$ operands u, f_1, e_1, f_2, e_2, …, f_r, e_r, such that $f = u\, f_1^{e_1} f_2^{e_2} \ldots f_r^{e_r}$.

The first operand u and the factors f_i are elements of the domain `ring`. The exponents e_i are integers.

Methods

Mathematical Methods

`_mult` — Multiply factored objects

```
_mult(f, g, …)
```

Suppose that `g` is an element of the domain `ring` (or can be converted into such an element).

If `g` is a unit of `ring` or a factor of `f`, then the result is a factored object of the same factorization type as `f`. Otherwise, the result is an element of `Factored` with the factorization type `"unknown"`.

If both `f` and `g` are factored objects with factorization type `"irreducible"`, then the result is again a factored object of this type, i.e., the result is still in factored form.

Otherwise, the factored form of `f` is lost, and the result of this method is an element of `ring`.

This method overloads the function `_mult` for factored objects, i.e., one may use it in the form `f*g*...`, or in functional notation: `_mult(f, g, ...)`.

`_power` — Raise a factored object to a certain power

```
_power(f, n)
```

If `n` is a positive integer and `f` a factored object with factorization type `"irreducible"` or `"squarefree"`, then the result is still a factored object of this type.

Otherwise, the factored form of `f` is lost, and the result of this method is an element of `ring`.

This method overloads the function `_power` for factored objects, i.e., one may use it in the form `f^n`, or in functional notation: `_power(f, n)`.

`expand` — Expand a factored object

```
expand(f)
```

`exponents` — Get the exponents of a factored object

```
exponents(f)
```

`factor` — Factor a factored object

```
factor(f)
```

If `f` already is of the factorization type `"irreducible"`, then this method just return `f`.

Otherwise, this method converts `f` into an element of the domain `ring` and calls the method `"factor"` of `ring`.

This method returns a factored object of the domain `Factored` with factorization type `"irreducible"`, if the factorization of `f` can be computed (otherwise, `FAIL` is returned).

This method overloads the function `factor` for factored objects, i.e., one may use it in the form `factor(f)`.

`factors` — Get the factors of a factored object

```
factors(f)
```

`irreducible` — Test if a factored object is irreducible

```
irreducible(f)
```

The test on irreducible is trivial, if `f` has the factorization type `"irreducible"`.

Otherwise, this method converts `f` into an element of `ring` and calls the method `"irreducible"` of `ring`. The value `FAIL` is returned, if the domain `ring` cannot test if `f` is irreducible.

`iszero` — Test on zero for factored objects

```
iszero(f)
```

This method overloads the function `iszero` for factored objects, i.e., one may use it in the form `iszero(f)`.

`sqrfree` — Compute a square-free factorization of a factored object

```
sqrfree(f)
```

If `f` already is of the factorization type `"squarefree"`, then this method just return `f`.

Otherwise, this method converts `f` into an element of the domain `ring` and calls the method `"squarefree"` of `ring`.

This method returns a factored object of the domain `Factored` with factorization type `"squarefree"`, if the square-free factorization of `f` can be computed (otherwise, `FAIL` is returned).

This method overloads the function `polylib::sqrfree` for factored objects, i.e., one may use it in the form `polylib::sqrfree(f)`.

Access Methods

`_index` — Extract a term of a factored object

```
_index(f, i)
```

Responds with an error message, if `i` is greater than the number of terms of `f`.

This method overloads the index operator `[ ]` for factored objects, i.e., one may use it in the form `f[i]`.

`getRing` — Get the ring of factorization

```
getRing(f)
```

`getType` — Get the type of factorization

```
getType(f)
```

`has` — Existence of an object in a factored object

```
has(f, x, …)
```

This method overloads the function `has` for factored objects, i.e., one may use it in the form `has(f, x, ...)`.

`map` — Map a function to the operands of factored objects

```
map(f, func, …)
```

See the system function `map` for details.

This method overloads the function `map` for factored objects, i.e., one may use it in the form `map(f, func, ...)`.

`nops` — Number of operands of a factored object

```
nops(f)
```

This method overloads the function `nops` for factored objects, i.e., one may use it in the form `nops(f)`.

`op` — Extract an operand of a factored object

```
op(f, i)
```

Returns `FAIL`, if `i` is greater than the number of operands of `f`.

This method overloads the function `op` for factored objects, i.e., one may use it in the form `op(f, i)`.

`select` — Select operands of a factored object

```
select(f, func, …)
```

This method overloads the function `select` for factored objects, i.e., one may use it in the form `select(f, func, ...)`.

`set_index` — Set/change a term of a factored object

```
set_index(f, i, x)
```

Responds with an error message, if `i` is greater than the number of terms of `f`.

Note: Make sure that `x` either is an element of the domain `ring`, or an integer.

This method overloads the index operator `[ ]` for factored objects, i.e., one may use it in the form `f[i] := x`.

`setRing` — Set the ring of factorization

```
setRing(f, ring)
```

Note: Use this method with caution! Make sure that the factorization of `f` is still valid over the new ring, and that the operands of `f` have the correct domain type.

`ring` must be a domain of category `Cat::IntegralDomain`, which is not checked by this method.

`setType` — Set the type of factorization

```
setType(f, type)
```

Note: Use this method with caution! Make sure that the factorization type corresponds with the factorization of `f`.

`subs` — Substitute subexpressions in the operands of a factored object

```
subs(f, x = a, …)
```

This method overloads the function `subs` for factored objects, i.e., one may use it in the form `subs(f, x = a, ...)`.

`subsop` — Substitute operands of a factored object

```
subsop(f, i = a, …)
```

This method overloads the function `subsop` for factored objects, i.e., one may use it in the form `subsop(f, i = a, ...)`.

`type` — Expression type of factored objects

```
type(f)
```

Conversion Methods

`convert` — Convert an object into a factored object

```
convert(x)
```

If the conversion fails, then `FAIL` is returned.

`x` may either be a list of the form `[u, f1, e1, ..., fr, er]` of odd length (where `u, f1, ..., fr` are of the domain type `ring`, or can be converted into such elements, and `e1, ..., er` are integers), or an element that can be converted into the domain `ring`. The latter case corresponds to the list `[ring::one,x,1]`.

`convert_to` — Convert factored objects into other domains

```
convert_to(f, T)
```

If the conversion fails, then `FAIL` is returned.

If `T` is the domain `DOM_LIST`, then the list of operands of `f` is returned.

If `T` is the domain `DOM_EXPR`, then the unevaluated expression `u*f1^e1*f2^e2*...*fr^er` is returned, where `u, f1, e1, ...` are the operands of `f`.

Otherwise, the method `"convert"` of the domain `T` is called to convert `f` into an element of the domain `T` (which could return `FAIL`).

Use the function `expr` to convert `f` into an object of a kernel domain (see below).

`create` — Create simple and fast a factored objects

```
create(list)
```

```
create(x)
```

This method creates a new factored object with the operands `ring::one, x, 1`.

expr — Convert a factored object into a kernel domain

```
expr(f)
```

Note: Note that the factored form of `f` may be lost due to this conversion.

expr2text — Convert a factored object into a string

```
expr2text(f)
```

testtype — Type testing for factored objects

```
testtype(f, T)
```

This method is called from the system function `testtype`.

TeX — LaTeX formatting of a factored object

```
TeX(f)
```

The method `"TeX"` of the domain `ring` is used to get the LaTeX-representation of the corresponding operands of `f`.

This method is called from the system function `generate::TeX`.

Technical Methods

_concat — Concatenate operands of factored objects

```
_concat(f, g)
```

`f` and `g` must have the same factorization type and factorization ring, otherwise an error message is given.

maprec — Allow recursive mapping for factored objects

```
maprec(f, x, …)
```

First `f` is converted into the unevaluated expression `u*f1^e1*f2^e2*...*fr^er`, where `u, f1, e1, ...` are the operands of `f`. Then the function `misc::maprec` is called with this expression as its first parameter.

Note that the result of this method is not longer an object of `Factored`!

`print` — Pretty-print routine for factored objects

```
print(f)
```

`unapply` — Create a procedure from a factored object

```
unapply(f, <x>)
```

This method overloads the function `fp::unapply` for factored objects, i.e., one may use it in the form `fp::unapply(f)`. See `fp::unapply` for details.

Series::Puiseux

Truncated Puiseux series expansions

Syntax

```
Series::Puiseux(f, x, <order>, <dir>)
```

```
Series::Puiseux(f, x = x0, <order>, <dir>)
```

Description

`Series::Puiseux` is a domain for truncated series expansions. Elements of this domain represent initial segments of Taylor, Laurent, or Puiseux series expansions, as well as slightly more general types of series expansions.

The system function `series` is the main application of this domain. It tries to compute a Taylor, Laurent, or Puiseux series or a more general series expansion of a given arithmetical expression, and the result is returned as an element of `Series::Puiseux` or, possibly, of the more general domain `Series::gseries`.

There is usually no need for you to explicitly create elements of this domain. The methods described on this help page apply if you want to process a result returned by `series` further.

Note: If you create elements explicitly as described above, then any special mathematical function, such as `sin` or `exp`, involving the series variable is considered as a coefficient. Use `series` to expand such functions as well, and use the constructor only if `f` does not contain any special mathematical functions. Cf. “Example 1” on page 7-409.

Use the type specifier `Type::Series` to determine for an element of this domain, which kind of series expansion it is.

Note: The coefficients are allowed to depend sublinearly on the variable of the series expansion. For example, logarithmic terms in the series variable may appear as

coefficients. Be aware that this is no Puiseux series in the mathematical sense. Cf. “Example 4” on page 7-417 and the help page for `series`.

Environment Interactions

The function is sensitive to the global variable `ORDER`, which determines the default number of terms of the expansion.

Examples

Example 1

You can create objects of `Series::Puiseux` in various ways. The standard method is to use the constructor. The second argument specifies the series variable and the expansion point, with default 0 if omitted:

```
Series::Puiseux(x/(1 - x), x);
Series::Puiseux(x/(1 - x), x = 2);
Series::Puiseux(x/(1 - x), x = complexInfinity);
```

$$x + x^2 + x^3 + x^4 + x^5 + x^6 + O(x^7)$$

$$-2 + x - 2 - (x-2)^2 + (x-2)^3 - (x-2)^4 + (x-2)^5 + O((x-2)^6)$$

$$-1 - \frac{1}{x} - \frac{1}{x^2} - \frac{1}{x^3} - \frac{1}{x^4} - \frac{1}{x^5} + O\left(\frac{1}{x^6}\right)$$

The third argument, if present, specifies the desired number of terms. If it is omitted, the value of the environment variable `ORDER` is used:

```
Series::Puiseux(x/(1 - x), x = 2, 4);
ORDER := 2:
Series::Puiseux(x/(1 - x), x);
delete ORDER:
```

$$-2 + x - 2 - (x-2)^2 + (x-2)^3 + O((x-2)^4)$$

$$x + x^2 + O(x^3)$$

The methods `const`, `one`, and `zero` provide shortcuts for creating series expansions with only a constant term or no non-zero term at all. Specifying the order of the error term is mandatory:

```
Series::Puiseux::const(PI, x, 4);
Series::Puiseux::one(x = 2, 3);
Series::Puiseux::zero(x = 0, 3/2);
Series::Puiseux::zero(x = complexInfinity, 5);
```

$$\pi + O(x^4)$$

$$1 + O((x-2)^3)$$

$$O\left(x^{3/2}\right)$$

$$O\left(\frac{1}{x^5}\right)$$

Note that, e.g., `O(x^(3/2))` is not an element of `Series::Puiseux`, but can be converted by the constructor:

```
f := O(x^(3/2));
g := Series::Puiseux(f, x);
domtype(f), domtype(g)
```

$$O\left(x^{3/2}\right)$$

$$O\left(x^{3/2}\right)$$

$$O, \text{Series::Puiseux}$$

Both the constructor `Series::Puiseux` and the method `const` regard special mathematical functions, such as `exp` or `sin`, as coefficients:

```
Series::Puiseux(sin(x)/(1 - x), x, 4);
Series::Puiseux::const(cos(x), x = 1, 3);
```

$$\sin(x) + x\sin(x) + x^2\sin(x) + x^3\sin(x) + O(x^4)$$

$$\cos(x) + O((x-1)^3)$$

Use the system function `series` if you want to have special functions expanded as well:

```
series(sin(x)/(1 - x), x, 4);
```

$$x + x^2 + \frac{5x^3}{6} + \frac{5x^4}{6} + O(x^5)$$

The constructor returns `FAIL`, if it cannot convert the input into an element of `Series::Puiseux`. Then `series` may be able to produce a more general expansion:

```
delete a:
Series::Puiseux(x^a/(1 - x), x);
f := series(x^a/(1 - x), x);
domtype(f);
```

FAIL

$$x^a + x\,x^a + x^a\,x^2 + x^a\,x^3 + x^a\,x^4 + x^a\,x^5 + O(x^a\,x^6)$$

Series::gseries

The method `create` is a purely syntactical constructor, where the operands are specified explicitly. The sixth and seventh arguments are optional and default to 0 and `Undirected`, respectively:

```
Series::Puiseux::create(3, 1, 5, [1/2, 5], x) =
Series::Puiseux::create(3, 1, 5, [1/2, 5], x, 0, Undirected)
```

$$\frac{x^{1/3}}{2} + 5\,x^{2/3} + O\left(x^{5/3}\right) = \frac{x^{1/3}}{2} + 5\,x^{2/3} + O\left(x^{5/3}\right)$$

```
Series::Puiseux::create(1, -2, 1, [ln(x), 0, 3], x, complexInfinity);
```

$$x^2 \ln(x) + 3 + O\left(\frac{1}{x}\right)$$

Example 2

We demonstrate the internal structure of objects of type `Series::Puiseux`:

```
f := series(exp(x), x = 1);
g := series(sin(sqrt(1/x)), x = infinity);
h := series(sin(sqrt(-x))/x, x = 0)
```

$$e + e\,(x-1) + \frac{e\,(x-1)^2}{2} + \frac{e\,(x-1)^3}{6} + \frac{e\,(x-1)^4}{24} + \frac{e\,(x-1)^5}{120} + O((x-1)^6)$$

$$\frac{1}{\sqrt{x}} - \frac{1}{6\,x^{3/2}} + \frac{1}{120\,x^{5/2}} + O\left(\frac{1}{x^{7/2}}\right)$$

$$-\frac{1}{\sqrt{-x}} + \frac{\sqrt{-x}}{6} - \frac{(-x)^{3/2}}{120} + O\left(x^{5/2}\right)$$

```
op(f);
op(g);
op(h)
```

$$0,\ 1,\ 0,\ 6,\ \left[e,\ e,\ \frac{e}{2},\ \frac{e}{6},\ \frac{e}{24},\ \frac{e}{120}\right],\ x = 1,\ \text{Undirected}$$

$$0,\ 2,\ 1,\ 7,\ \left[1,\ 0,\ -\frac{1}{6},\ 0,\ \frac{1}{120}\right],\ x = \text{complexInfinity},\ \text{Left}$$

$$1,\ 2,\ -1,\ 5,\ \left[-\frac{1}{\sqrt{-x}},\ 0,\ \frac{\sqrt{-x}}{6},\ 0,\ -\frac{(-x)^{3/2}}{120}\right],\ x = 0,\ \text{Undirected}$$

The series `f` and `g` are of type 0, while `h` is of type `1`:

```
op(f, 1), op(g, 1), op(h, 1)
```

$$0,\ 0,\ 1$$

The branching order of `f` is 1, and the branching order of both `g` and `h` is 2:

```
op(f, 2), op(g, 2), op(h, 2)
```

$$1,\ 2,\ 2$$

The third and the fourth operand determine the order of the leading term and the error term, respectively:

```
ldegree(f) = op(f, 3)/op(f, 2),
ldegree(g) = op(g, 3)/op(g, 2),
ldegree(h) = op(h, 3)/op(h, 2);
Series::Puiseux::order(f) = op(f, 4)/op(f, 2),
Series::Puiseux::order(g) = op(g, 4)/op(g, 2),
Series::Puiseux::order(h) = op(h, 4)/op(h, 2);
```

$$0 = 0,\ \frac{1}{2} = \frac{1}{2},\ -\frac{1}{2} = -\frac{1}{2}$$

$$6 = 6,\ \frac{7}{2} = \frac{7}{2},\ \frac{5}{2} = \frac{5}{2}$$

For series expansions of type 0, the fifth operand contains the coefficients of the expansion:

```
op(f, 5) = [coeff(f)];
op(g, 5) = [coeff(g)];
```

$$\left[e,\ e,\ \frac{e}{2},\ \frac{e}{6},\ \frac{e}{24},\ \frac{e}{120}\right] = \left[e,\ e,\ \frac{e}{2},\ \frac{e}{6},\ \frac{e}{24},\ \frac{e}{120}\right]$$

$$\left[1,\ 0,\ -\frac{1}{6},\ 0,\ \frac{1}{120}\right] = \left[1,\ 0,\ -\frac{1}{6},\ 0,\ \frac{1}{120}\right]$$

However, `h` is an expansion of type 1, and then the fifth operand stores the summands:

```
op(h, 5);
[coeff(h)];
```

$$\left[-\frac{1}{\sqrt{-x}},\ 0,\ \frac{\sqrt{-x}}{6},\ 0,\ -\frac{(-x)^{3/2}}{120}\right]$$

$$\left[-\frac{\sqrt{x}}{\sqrt{-x}},\ 0,\ \frac{\sqrt{-x}}{6\sqrt{x}},\ 0,\ -\frac{(-x)^{3/2}}{120\,x^{3/2}}\right]$$

The sixth operand contains the series variable and the expansion point:

```
op(f, 6), Series::Puiseux::indet(f), Series::Puiseux::point(f);
op(g, 6), Series::Puiseux::indet(g), Series::Puiseux::point(g);
op(h, 6), Series::Puiseux::indet(h), Series::Puiseux::point(h);
```

$$x = 1,\ x,\ 1$$

$$x = \text{complexInfinity},\ x,\ \text{complexInfinity}$$

$$x = 0,\ x,\ 0$$

The expansions `f` and `h` are undirected, while `g` is a directed expansion from the left along the real line to the positive infinity:

```
op(f, 7) = Series::Puiseux::direction(f),
op(g, 7) = Series::Puiseux::direction(g),
op(h, 7) = Series::Puiseux::direction(h);
```

$$\text{Undirected} = \text{Undirected},\ \text{Left} = \text{Left},\ \text{Undirected} = \text{Undirected}$$

Note: Since the internal structure may be subject to changes, accessing the operands of and element of `Series::Puiseux` via `op` should be avoided. Use the corresponding access methods instead.

Example 3

Around branch points, the series expansions of type 1 can approximate a function in a wider range than those of type 0:

```
f := x -> arcsin(x + 1):
g := series(f(x), x, 2);
h := series(f(x), x, 2, Right);
```

$$\frac{\pi}{2} - \sqrt{2}\,\sqrt{-x} - \frac{\sqrt{2}\,(-x)^{3/2}}{12} + O\left(x^{5/2}\right)$$

$$\frac{\pi}{2} - \sqrt{2}\,\sqrt{x}\,\mathrm{i} + \frac{\sqrt{2}\,x^{3/2}\,\mathrm{i}}{12} + O\left(x^{5/2}\right)$$

The expansion `g`, of type 1, approximates `f` well in an open disc centered at the origin. However, the expansion `h`, of type 0, was requested for positive real values of `x` only, and in fact it does not approximate `f` on the negative real axis and in the upper half plane:

```
op(g);
op(h);
```

$$1,\ 2,\ 0,\ 5,\ \left[\frac{\pi}{2},\ -\sqrt{2}\,\sqrt{-x},\ 0,\ -\frac{\sqrt{2}\,(-x)^{3/2}}{12}\right],\ x = 0,\ \text{Undirected}$$

$$0,\ 2,\ 0,\ 5,\ \left[\frac{\pi}{2},\ -\sqrt{2}\,\mathrm{i},\ 0,\ \frac{\sqrt{2}\,\mathrm{i}}{12}\right],\ x = 0,\ \text{Right}$$

```
DIGITS := 4:
[f(0.01), f(0.01*I), f(-0.01), f(-0.01*I)];
map([g(0.01), g(0.01*I), g(-0.01), g(-0.01*I)], float);
map([h(0.01), h(0.01*I), h(-0.01), h(-0.01*I)], float);
```

```
delete DIGITS:
```

$$[1.571 - 0.1413\,i,\ 1.471 + 0.1001\,i,\ 1.429,\ 1.471 - 0.1001\,i]$$

$$[1.571 - 0.1413\,i,\ 1.471 + 0.1001\,i,\ 1.429,\ 1.471 - 0.1001\,i]$$

$$[1.571 - 0.1413\,i,\ 1.671 - 0.1001\,i,\ 1.712,\ 1.471 - 0.1001\,i]$$

The method `convert01` converts a series expansion of type 0 into one of type 1:

```
h1 := Series::Puiseux::convert01(h);
op(h1);
```

$$\frac{\pi}{2} - \sqrt{2}\,\sqrt{x}\,i + \frac{\sqrt{2}\,x^{3/2}\,i}{12} + O\left(x^{5/2}\right)$$

$$1,\ 2,\ 0,\ 5,\ \left[\frac{\pi}{2},\ -\sqrt{2}\,\sqrt{x}\,i,\ 0,\ \frac{\sqrt{2}\,x^{3/2}\,i}{12}\right],\ x = 0,\ \text{Right}$$

The reverse conversion, using the method `convert10`, is in not always possible:

```
op(Series::Puiseux::convert10(h1));
op(Series::Puiseux::convert10(g));
```

$$0,\ 2,\ 0,\ 5,\ \left[\frac{\pi}{2},\ -\sqrt{2}\,i,\ 0,\ \frac{\sqrt{2}\,i}{12}\right],\ x = 0,\ \text{Right}$$

$$1,\ 2,\ 0,\ 5,\ \left[\frac{\pi}{2},\ -\sqrt{2}\,\sqrt{-x},\ 0,\ -\frac{\sqrt{2}\,(-x)^{3/2}}{12}\right],\ x = 0,\ \text{Undirected}$$

You can enforce a conversion by using properties:

```
assume(x > 0):
op(Series::Puiseux::convert10(g));
```

```
unassume(x):
```

$$0,\ 2,\ 0,\ 5,\ \left[\frac{\pi}{2},\ -\sqrt{2}\,i,\ 0,\ \frac{\sqrt{2}\,i}{12}\right],\ x = 0,\ \text{Undirected}$$

Example 4

Despite the name, elements of `Series::Puiseux` may contain coefficient functions depending on the series variable:

```
f := series(psi(x), x = infinity, 4);
domtype(f), coeff(f, 0)
```

$$\ln(x) - \frac{1}{2\,x} - \frac{1}{12\,x^2} + O\left(\frac{1}{x^4}\right)$$

$$\text{Series::Puiseux},\ \ln(x)$$

With respect to differentiation, integration, and composition, such expansions behave like functions of the series variable and not like formal series:

```
diff(f, x) = series(diff(psi(x), x), x = infinity, 4)
```

$$\frac{1}{x} + \frac{1}{2\,x^2} + \frac{1}{6\,x^3} + O\left(\frac{1}{x^5}\right) = \frac{1}{x} + \frac{1}{2\,x^2} + \frac{1}{6\,x^3} + O\left(\frac{1}{x^5}\right)$$

```
int(f, x) = series(int(psi(x), x), x = infinity, 4)
```

$$x\,(\ln(x) - 1) - \frac{\ln(x)}{2} + \frac{1}{12\,x} + O\left(\frac{1}{x^3}\right) = x\,(\ln(x) - 1) + \frac{\ln(2)}{2} + \frac{\ln(\pi)}{2} - \frac{\ln(x)}{2} + \frac{1}{12\,x} + O\left(\frac{1}{x^3}\right)$$

```
f @ series(2*x, x = infinity) = series(psi(2*x), x = infinity, 4)
```

$$\ln(2) + \ln(x) - \frac{1}{4\,x} - \frac{1}{48\,x^2} + O\left(\frac{1}{x^4}\right) = \ln(2) + \ln(x) - \frac{1}{4\,x} - \frac{1}{48\,x^2} + O\left(\frac{1}{x^4}\right)$$

Example 5

The basic arithmetical operations are implemented for elements of `Series::Puiseux`:

```
f := series(exp(x), x, 4);
g := series(sqrt(x)/(1 - x), x, 4);
h := series(cot(x), x, 4);
```

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + O(x^4)$$

$$\sqrt{x} + x^{3/2} + x^{5/2} + x^{7/2} + O\left(x^{9/2}\right)$$

$$\frac{1}{x} - \frac{x}{3} + O(x^3)$$

```
f + g + h;
_plus(f, g, h)
```

$$\frac{1}{x} + 1 + \sqrt{x} + \frac{2x}{3} + x^{3/2} + \frac{x^2}{2} + x^{5/2} + O(x^3)$$

$$\frac{1}{x} + 1 + \sqrt{x} + \frac{2x}{3} + x^{3/2} + \frac{x^2}{2} + x^{5/2} + O(x^3)$$

```
f - h = _subtract(f, h);
-g = _negate(g);
```

$$-\frac{1}{x} + 1 + \frac{4x}{3} + \frac{x^2}{2} + O(x^3) = -\frac{1}{x} + 1 + \frac{4x}{3} + \frac{x^2}{2} + O(x^3)$$

$$-\sqrt{x} - x^{3/2} - x^{5/2} - x^{7/2} + O\left(x^{9/2}\right) = -\sqrt{x} - x^{3/2} - x^{5/2} - x^{7/2} + O\left(x^{9/2}\right)$$

```
f*g*h;
```

```
_mult(f, g, h)
```

$$\frac{1}{\sqrt{x}} + 2\sqrt{x} + \frac{13\,x^{3/2}}{6} + 2\,x^{5/2} + O\left(x^{7/2}\right)$$

$$\frac{1}{\sqrt{x}} + 2\sqrt{x} + \frac{13\,x^{3/2}}{6} + 2\,x^{5/2} + O\left(x^{7/2}\right)$$

```
f/g = _divide(f, g);
1/h = _invert(h);
```

$$\frac{1}{\sqrt{x}} - \frac{x^{3/2}}{2} - \frac{x^{5/2}}{3} + O\left(x^{7/2}\right) = \frac{1}{\sqrt{x}} - \frac{x^{3/2}}{2} - \frac{x^{5/2}}{3} + O\left(x^{7/2}\right)$$

$$x + \frac{x^3}{3} + O(x^5) = x + \frac{x^3}{3} + O(x^5)$$

Operands that are not of type `Series::Puiseux` are implicitly converted into series expansions with the same expansion point via the constructor before the arithmetical operation is performed:

```
f - 1 - x;
h * (sin(x) + x);
```

$$\frac{x^2}{2} + \frac{x^3}{6} + O(x^4)$$

$$\frac{\sin(x)}{x} + 1 - \frac{x\,\sin(x)}{3} - \frac{x^2}{3} + O(x^3)$$

An error occurs when the expansion points differ or the directions of expansion are incompatible:

```
f := series(arccot(x), x = 0, Left);
g := series(sqrt(sin(x)), x = 0, Right);
f + g
```

$$-\frac{\pi}{2}-x+\frac{x^3}{3}-\frac{x^5}{5}+O(x^7)$$

$$\sqrt{x}-\frac{x^{5/2}}{12}+\frac{x^{9/2}}{1440}+O\left(x^{13/2}\right)$$

```
Error: Inconsistent direction. [Series::Puiseux::plus]
```

```
h := series(1/x, x = 2, 4);
f * h
```

$$\frac{1}{2}-\frac{x-2}{4}+\frac{(x-2)^2}{8}-\frac{(x-2)^3}{16}+O((x-2)^4)$$

```
Error: Both series must use the same variables and expansion points. [Series::Puiseux:
```

If the directions are compatible, then the direction of the result specifies the minimal range where all operands are defined:

```
s := series(tanh(x), x, Real);
```

$$x-\frac{x^3}{3}+\frac{2\,x^5}{15}+O(x^7)$$

```
f + s;
Series::Puiseux::direction(%)
```

$$-\frac{\pi}{2}-\frac{x^5}{15}+O(x^7)$$

Left

Example 6

The method `scalmult` implements multiplication by a constant or a single term:

```
f := series(1 + 2*x^3, x);
Series::Puiseux::scalmult(f, 5) = 5*f;
Series::Puiseux::scalmult(f, 5, 3) = 5*x^3*f;
```

$$1 + 2\,x^3 + O(x^6)$$

$$5 + 10\,x^3 + O(x^6) = 5 + 10\,x^3 + O(x^6)$$

$$5\,x^3 + 10\,x^6 + O(x^9) = 5\,x^3 + 10\,x^6 + O(x^9)$$

```
g := series(1 + 2*x^3, x = 2, 3);
Series::Puiseux::scalmult(g, 1, 3) = (x - 2)^3*g
```

$$17 + 24\,(x-2) + 12\,(x-2)^2 + O((x-2)^3)$$

$$17\,(x-2)^3 + 24\,(x-2)^4 + 12\,(x-2)^5 + O((x-2)^6) =$$

$$17\,(x-2)^3 + 24\,(x-2)^4 + 12\,(x-2)^5 + O((x-2)^6)$$

```
h := series(1 + 2*x^3, x = complexInfinity);
Series::Puiseux::scalmult(h, 1, 1/2) = x^(-1/2)*h
```

$$2\,x^3 + 1 + O\left(\frac{1}{x^3}\right)$$

$$2\,x^{5/2} + \frac{1}{\sqrt{x}} + O\left(\frac{1}{x^{7/2}}\right) = 2\,x^{5/2} + \frac{1}{\sqrt{x}} + O\left(\frac{1}{x^{7/2}}\right)$$

Example 7

Exponentiation is implemented for integral and rational exponents:

```
f := series(exp(x), x, 3);
```

```
f^2 = _power(f, 2);
f^(1/3) = _power(f, 1/3)
```

$$1 + x + \frac{x^2}{2} + O(x^3)$$

$$1 + 2\,x + 2\,x^2 + O(x^3) = 1 + 2\,x + 2\,x^2 + O(x^3)$$

$$1 + \frac{x}{3} + \frac{x^2}{18} + O(x^3) = 1 + \frac{x}{3} + \frac{x^2}{18} + O(x^3)$$

Exponents are allowed to be non-rational, if the series expansion starts with a constant summand independent of the series variable:

```
f^I = series(exp(I*x), x, 3);
```

$$1 + x\,\mathrm{i} - \frac{x^2}{2} + O(x^3) = 1 + x\,\mathrm{i} - \frac{x^2}{2} + O(x^3)$$

```
g := series(sin(-x), x);
g^I
```

$$-x + \frac{x^3}{6} - \frac{x^5}{120} + O(x^7)$$

```
Error: The exponent must be a rational number. [Series::Puiseux::_power]
```

If the exponent contains the series variable, then an error occurs:

```
f^x
```

```
Error: The exponent must not contain the series variable. [Series::Puiseux::_power]
```

For undirected expansions and rational exponents that are not integral, the result has type 1 in general:

```
g^(1/2);
```

```
op(%, 1);
```

$$\sqrt{-x} - \frac{(-x)^{5/2}}{12} + \frac{(-x)^{9/2}}{1440} + O\left(x^{13/2}\right)$$

$$1$$

The result simplifies when you specify one of the directions `Left` or `Right`:

```
g := series(sin(-x), x, Left):
g^(1/2);
op(%, 1);
```

$$-\sqrt{x}\,\mathrm{i} + \frac{x^{5/2}\,\mathrm{i}}{12} - \frac{x^{9/2}\,\mathrm{i}}{1440} + O\left(x^{13/2}\right)$$

$$0$$

```
g := series(sin(-x), x, Right):
g^(1/2);
```

$$\sqrt{x}\,\mathrm{i} - \frac{x^{5/2}\,\mathrm{i}}{12} + \frac{x^{9/2}\,\mathrm{i}}{1440} + O\left(x^{13/2}\right)$$

Example 8

Functional composition of elements of `Series::Puiseux` is implemented by the method `_fconcat`:

```
f := series(ln(x), x = 1, 4);
g := series(cos(y), y = 0);
f@g = _fconcat(f, g);
series(ln(cos(y)), y = 0, 4);
```

$$x - 1 - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \frac{(x-1)^4}{4} + O\left((x-1)^5\right)$$

$$1 - \frac{y^2}{2} + \frac{y^4}{24} + O(y^6)$$

$$-\frac{y^2}{2} - \frac{y^4}{12} + O(y^6) = -\frac{y^2}{2} - \frac{y^4}{12} + O(y^6)$$

$$-\frac{y^2}{2} - \frac{y^4}{12} + O(y^6)$$

If the left argument is not of type `Series::Puiseux`, it is implicitly expanded around the limit point of the right argument before the composition:

```
f := series(sin(-x), x = 0);
sqrt(y) @ f = Series::Puiseux(sqrt(y), y) @ f;
```

$$-x + \frac{x^3}{6} - \frac{x^5}{120} + O(x^7)$$

$$\sqrt{-x} - \frac{(-x)^{5/2}}{12} + \frac{(-x)^{9/2}}{1440} + O\left(x^{13/2}\right) = \sqrt{-x} - \frac{(-x)^{5/2}}{12} + \frac{(-x)^{9/2}}{1440} + O\left(x^{13/2}\right)$$

If the right argument is not of type `Series::Puiseux`, it is implicitly expanded around the origin via the constructor before the composition:

```
f @ sqrt(y) = f @ Series::Puiseux(sqrt(y), y)
```

$$-\sqrt{y} + \frac{y^{3/2}}{6} - \frac{y^{5/2}}{120} + O\left(y^{7/2}\right) = -\sqrt{y} + \frac{y^{3/2}}{6} - \frac{y^{5/2}}{120} + O\left(y^{7/2}\right)$$

This may not work if the argument to be converted contains special mathematical functions, but you can explicitly expand it into a series via `series` in this case:

```
f @ tan(y)
```

FAIL

```
f @ series(tan(y), y = 0)
```

$$-y-\frac{y^3}{6}+\frac{y^5}{40}+O(y^7)$$

Mathematically, the composition of series expansions is not defined if the limit point of the right argument is not the expansion point of the left argument:

```
g := series(y^2 - 1, y = 0);
f @ g
```

$$-1+y^2+O(y^6)$$

FAIL

```
f @ (y^2 - 1)
```

FAIL

```
f @ series(y^2 - 1, y = 1, 4)
```

$$-2\,(y-1)-(y-1)^2+\frac{4\,(y-1)^3}{3}+2\,(y-1)^4+O((y-1)^5)$$

The method `revert` computes the inverse of a truncated series expansion with respect to composition. The expansion point of the inverse is the limit point of the input and vice versa:

```
f := series(ln(x), x = 1, 4);
revert(f) = series(exp(x), x = 0, 5)
```

$$x-1-\frac{(x-1)^2}{2}+\frac{(x-1)^3}{3}-\frac{(x-1)^4}{4}+O((x-1)^5)$$

$$1+x+\frac{x^2}{2}+\frac{x^3}{6}+\frac{x^4}{24}+O(x^5)=1+x+\frac{x^2}{2}+\frac{x^3}{6}+\frac{x^4}{24}+O(x^5)$$

```
f := series(cot(x), x = 0);
revert(f) = series(arccot(x), x = complexInfinity);
```

$$\frac{1}{x} - \frac{x}{3} - \frac{x^3}{45} + O(x^5)$$

$$\frac{1}{x} - \frac{1}{3\,x^3} + \frac{1}{5\,x^5} + O\left(\frac{1}{x^7}\right) = \frac{1}{x} - \frac{1}{3\,x^3} + \frac{1}{5\,x^5} + O\left(\frac{1}{x^7}\right)$$

```
f @ revert(f), revert(f) @ f
```

$$x + O\left(\frac{1}{x^5}\right), x + O(x^7)$$

If the series variable occurs in the coefficients or the type flag is 1, an error occurs:

```
f := series(ln(sin(x)), x);
g := series(arcsin(x + 1), x, 2);
```

$$\ln(x) - \frac{x^2}{6} - \frac{x^4}{180} + O(x^6)$$

$$\frac{\pi}{2} - \sqrt{2}\,\sqrt{-x} - \frac{\sqrt{2}\,(-x)^{3/2}}{12} + O\left(x^{5/2}\right)$$

```
revert(f)
```

```
Error: Cannot compute the functional inverse. [Series::Puiseux::revert]
```

```
revert(g)
```

```
Error: Cannot compute the functional inverse. [Series::Puiseux::revert]
```

Example 9

The methods `diff` and `int` implement term-by-term differentiation and integration:

```
f := series(ln(x), x = 1, 4);
g := diff(f, x);
series(1/x, x = 1, 4);
int(g, x);
```

$$x - 1 - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \frac{(x-1)^4}{4} + O((x-1)^5)$$

$$1 - (x-1) + (x-1)^2 - (x-1)^3 + O((x-1)^4)$$

$$1 - (x-1) + (x-1)^2 - (x-1)^3 + O((x-1)^4)$$

$$x - 1 - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \frac{(x-1)^4}{4} + O((x-1)^5)$$

If you specify a range of integration, then the result is an arithmetical expression plus a symbolic definite integral of the *O*-term:

```
int(f, x = 1..2);
```

$$\int_1^2 O((x-1)^5, x = 1)\, \mathrm{d}x + \frac{11}{30}$$

Example 10

Most special mathematical functions are overloaded for `Series::Puiseux`:

```
f := series(x/(1 - x), x, 4);
exp(f) = series(exp(x/(1 - x)), x, 4);
ln(f) = series(ln(x/(1 - x)), x, 4);
```

$$x + x^2 + x^3 + x^4 + O(x^5)$$

$$1 + x + \frac{3\,x^2}{2} + \frac{13\,x^3}{6} + O(x^4) = 1 + x + \frac{3\,x^2}{2} + \frac{13\,x^3}{6} + O(x^4)$$

$$\ln(x) + x + \frac{x^2}{2} + \frac{x^3}{3} + O(x^4) = \ln(x) + x + \frac{x^2}{2} + \frac{x^3}{3} + O(x^4)$$

If the system is unable to compute the composition, it returns a symbolic function call with evaluated arguments:

```
delete g:
g(f)
```

$$g(x + x^2 + x^3 + x^4 + O(x^5))$$

```
exp(series(x + 1/x, x = infinity, 5))
```

$$e^{x + \frac{1}{x} + O\left(\frac{1}{x^4}\right)}$$

In this case, you can try `series` to compute the composition:

```
series(exp(x + 1/x), x = infinity, 5)
```

$$e^x + \frac{e^x}{x} + \frac{e^x}{2\,x^2} + \frac{e^x}{6\,x^3} + \frac{e^x}{24\,x^4} + O\left(\frac{e^x}{x^5}\right)$$

Example 11

The system functions `Re`, `Im`, and `conjugate` work for all real series expansions:

```
f := series(exp(I*x), x, Real);
Re(f) = series(cos(x), x, Real);
Im(f) = series(sin(x), x, Real) + O(x^6);
conjugate(f) = series(exp(-I*x), x, Real);
```

$$1 + x\,\mathrm{i} - \frac{x^2}{2} - \frac{x^3\,\mathrm{i}}{6} + \frac{x^4}{24} + \frac{x^5\,\mathrm{i}}{120} + O(x^6)$$

$$1 - \frac{x^2}{2} + \frac{x^4}{24} + O(x^6) = 1 - \frac{x^2}{2} + \frac{x^4}{24} + O(x^6)$$

$$x - \frac{x^3}{6} + \frac{x^5}{120} + O(x^6) = x - \frac{x^3}{6} + \frac{x^5}{120} + O(x^6)$$

$$1 - x\,i - \frac{x^2}{2} + \frac{x^3\,i}{6} + \frac{x^4}{24} - \frac{x^5\,i}{120} + O(x^6) = 1 - x\,i - \frac{x^2}{2} + \frac{x^3\,i}{6} + \frac{x^4}{24} - \frac{x^5\,i}{120} + O(x^6)$$

Except in trivial cases, a symbolic function call is returned for an undirected expansion:

```
Re(series(PI, x));
Re(series(exp(I*x), x));
```

$$\pi + O(x^6)$$

$$\Re\left(1 + x\,i - \frac{x^2}{2} - \frac{x^3\,i}{6} + \frac{x^4}{24} + \frac{x^5\,i}{120} + O(x^6)\right)$$

Example 12

The method `contfrac` converts a series expansion into a continued fraction:

```
f := series(exp(x), x, 10);
contfrac(f);
```

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720} + \frac{x^7}{5040} + \frac{x^8}{40320} + \frac{x^9}{362880} + O(x^{10})$$

$$1 + \cfrac{x}{1 + \cfrac{x}{-2 + \cfrac{x}{-3 + \cfrac{x}{2 + \cfrac{x}{5 + \cfrac{x}{-2 + \cfrac{x}{-7 + \cfrac{x}{2 + \cfrac{x}{9 + O(x)}}}}}}}}}$$

```
g := series(tan(x), x = PI, 10);
contfrac(g);
```

$$x - \pi - \frac{(\pi - x)^3}{3} - \frac{2\,(\pi - x)^5}{15} - \frac{17\,(\pi - x)^7}{315} - \frac{62\,(\pi - x)^9}{2835} + O(-(\pi - x)^{11})$$

$$\cfrac{-\pi + x}{1 + \cfrac{(\pi - x)^2}{-3 + \cfrac{(\pi - x)^2}{5 + \cfrac{(\pi - x)^2}{-7 + \cfrac{(\pi - x)^2}{9 + O\left((\pi - x)^2\right)}}}}}$$

If the coefficients of a series expansion depend on the series variable, then so do the coefficients of the corresponding continued fraction:

```
h := series(ln(x + 1/x), x = infinity);
contfrac(h)
```

$$\ln(x) + \frac{1}{x^2} - \frac{1}{2\,x^4} + O\left(\frac{1}{x^6}\right)$$

$$\ln(x) + \cfrac{x^{-2}}{1 + \cfrac{x^{-2}}{2 + O\left(x^{-2}\right)}}$$

Example 13

For series expansions around the origin, the method `laplace`, overloading `laplace`, computes the Laplace transform term by term, if the second argument is the series variable. The result is a series expansion around infinity:

```
delete s:
f := series(exp(x), x);
g := laplace(f, x, s);
series(laplace(exp(x), x, s), s = infinity);
```

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + O(x^6)$$

$$\frac{1}{s} + \frac{1}{s^2} + \frac{1}{s^3} + \frac{1}{s^4} + \frac{1}{s^5} + \frac{1}{s^6} + O\left(\frac{1}{s^7}\right)$$

$$\frac{1}{s}+\frac{1}{s^2}+\frac{1}{s^3}+\frac{1}{s^4}+\frac{1}{s^5}+\frac{1}{s^6}+O\left(\frac{1}{s^7}\right)$$

Similarly, the method `ilaplace` computes the inverse Laplace transform term by term for series expansions around infinity, if the second argument is the series variable. The result is a series expansion around 0:

```
ilaplace(g, s, x)
```

$$1+x+\frac{x^2}{2}+\frac{x^3}{6}+\frac{x^4}{24}+\frac{x^5}{120}+O(x^6)$$

The Laplace transform and the inverse Laplace transform, respectively, of a series do not make sense for expansion points other than 0 or infinity, respectively, and in these cases a symbolic function call is returned:

```
laplace(series(ln(x), x = 1, 2), x, s);
```

$$\text{laplace}\left(x-1-\frac{(x-1)^2}{2}+O((x-1)^3), x, s\right)$$

If the second argument is not the series variable, then the coefficients are transformed:

```
h := series(sin(x*y), x = 1, 2);
laplace(h, y, s);
```

$$\sin(y)+y\cos(y)\,(x-1)+O((x-1)^2)$$

$$\frac{1}{s^2+1}-\left(\frac{1}{s^2+1}-\frac{2\,s^2}{(s^2+1)^2}\right)(x-1)+O((x-1)^2)$$

Example 14

When called with one argument, the method `coeff` returns the sequence of all coefficients of a series expansion:

```
f := series(tan(x), x);
```

```
coeff(f)
```

$$x + \frac{x^3}{3} + \frac{2\,x^5}{15} + O(x^7)$$

$$1,\ 0,\ \frac{1}{3},\ 0,\ \frac{2}{15}$$

```
g := series(1/(x - 1)^2, x = infinity);
coeff(g)
```

$$\frac{1}{x^2} + \frac{2}{x^3} + \frac{3}{x^4} + \frac{4}{x^5} + \frac{5}{x^6} + \frac{6}{x^7} + O\left(\frac{1}{x^8}\right)$$

$$1,\ 2,\ 3,\ 4,\ 5,\ 6$$

When called with two arguments, `coeff` returns an individual coefficient:

```
coeff(f, -1), coeff(f, 1), coeff(f, 2), coeff(f, 13/2);
```

$$0,\ 1,\ 0,\ 0$$

If the second argument exceeds the order of the error term, `coeff` returns `FAIL`:

```
coeff(f, 10)
```

$$\text{FAIL}$$

When the expansion point is `complexInfinity`, `coeff(s, n)` returns the coefficient of $\frac{1}{x^n}$, where `x` is the series variable of `s`:

```
coeff(g, 2), coeff(g, -3), coeff(g, -15/2)
```

$$0,\ 2,\ 0$$

Specifying the series variable as second or third argument, respectively, is optional:

```
coeff(f) = coeff(f, x);
```

```
coeff(f, 3) = coeff(f, x, 3)
```

$$\left(1,\ 0,\ \frac{1}{3},\ 0,\ \frac{2}{15}\right) = \left(1,\ 0,\ \frac{1}{3},\ 0,\ \frac{2}{15}\right)$$

$$\frac{1}{3} = \frac{1}{3}$$

For series expansions of type 1, the "coefficients" in general involve the series variable:

```
h := series(sin(sqrt(-x)), x);
coeff(h);
coeff(h, 3/2);
```

$$\sqrt{-x} - \frac{(-x)^{3/2}}{6} + \frac{(-x)^{5/2}}{120} + O\left(x^{7/2}\right)$$

$$\frac{\sqrt{-x}}{\sqrt{x}},\ 0,\ -\frac{(-x)^{3/2}}{6\,x^{3/2}},\ 0,\ \frac{(-x)^{5/2}}{120\,x^{5/2}}$$

$$-\frac{(-x)^{3/2}}{6\,x^{3/2}}$$

Example 15

The method `ldegree` returns the order of the leading term of a series expansion. When the expansion point is `complexInfinity` and the leading term is $\frac{1}{x^n}$, then this is n:

```
f := series(x*sin(sqrt(-x)), x);
g := series(cot(x), x = PI);
h := series(2*arccot(x), x = infinity);
```

$$-(-x)^{3/2} + \frac{(-x)^{5/2}}{6} - \frac{(-x)^{7/2}}{120} + O\left(x^{9/2}\right)$$

$$-\frac{1}{\pi - x} - \frac{x - \pi}{3} + \frac{(\pi - x)^3}{45} + O\left(-(\pi - x)^5\right)$$

$$\frac{2}{x} - \frac{2}{3\,x^3} + \frac{2}{5\,x^5} + O\left(\frac{1}{x^7}\right)$$

```
ldegree(f), ldegree(g), ldegree(h)
```

$$\frac{3}{2},\ -1,\ 1$$

The method `lcoeff` returns the coefficient of the leading term. For an expansion of type 1, it generally involves the series variable:

```
lcoeff(f), lcoeff(g), lcoeff(h)
```

$$-\frac{(-x)^{3/2}}{x^{3/2}},\ 1,\ 2$$

The method `lterm` returns the leading term itself:

```
lterm(f), lterm(g), lterm(h)
```

$$x^{3/2},\ -\frac{1}{\pi - x},\ \frac{1}{x}$$

Finally, the method `lmonomial` returns the whole summand:

```
lmonomial(f) = lcoeff(f)*lterm(f);
lmonomial(g) = lcoeff(g)*lterm(g);
lmonomial(h) = lcoeff(h)*lterm(h);
```

$$-(-x)^{3/2} = -(-x)^{3/2}$$

$$-\frac{1}{\pi - x} = -\frac{1}{\pi - x}$$

$$\frac{2}{x} = \frac{2}{x}$$

If the series expansion consists only of an O-term, all four methods return `FAIL`:

```
s := Series::Puiseux::zero(x, 6);
ldegree(s), lcoeff(s), lterm(s), lmonomial(s)
```

$$O(x^6)$$

$$\text{FAIL}, \text{FAIL}, \text{FAIL}, \text{FAIL}$$

Example 16

The methods `nthcoeff`, `nthmonomial`, and `nthterm` return the nth non-zero coefficient, monomial, or term, respectively, of a series expansion. In contrast to polynomials, they count from the term of lowest order on, i.e., the ordering is ascending by exponent for finite expansion points and descending by exponent when the expansion point is `complexInfinity`:

```
f := series(x*sin(sqrt(-x)), x);
g := series(cot(x), x = PI);
h := series(2*arccot(x), x = infinity);
```

$$-(-x)^{3/2} + \frac{(-x)^{5/2}}{6} - \frac{(-x)^{7/2}}{120} + O\left(x^{9/2}\right)$$

$$-\frac{1}{\pi - x} - \frac{x - \pi}{3} + \frac{(\pi - x)^3}{45} + O\left(-(\pi - x)^5\right)$$

$$\frac{2}{x} - \frac{2}{3\,x^3} + \frac{2}{5\,x^5} + O\left(\frac{1}{x^7}\right)$$

```
nthcoeff(f, 1) = lcoeff(f);
nthmonomial(g, 1) = lmonomial(g);
nthterm(h, 1) = lterm(h);
```

$$-\frac{(-x)^{3/2}}{x^{3/2}} = -\frac{(-x)^{3/2}}{x^{3/2}}$$

$$-\frac{1}{\pi - x} = -\frac{1}{\pi - x}$$

$$\frac{1}{x} = \frac{1}{x}$$

```
nthcoeff(f, 3), nthmonomial(f, 3), nthterm(f, 3);
nthcoeff(g, 3), nthmonomial(g, 3), nthterm(g, 3);
nthcoeff(h, 3), nthmonomial(h, 3), nthterm(h, 3);
```

$$-\frac{(-x)^{7/2}}{120\,x^{7/2}}, -\frac{(-x)^{7/2}}{120}, x^{7/2}$$

$$-\frac{1}{45}, \frac{(\pi - x)^3}{45}, -(\pi - x)^3$$

$$\frac{2}{5}, \frac{2}{5\,x^5}, \frac{1}{x^5}$$

If the second argument is not positive or exceeds the number of non-zero summands, all three methods return `FAIL`:

```
nthcoeff(f, -4), nthterm(g, 0), nthmonomial(h, 4)
```

FAIL, FAIL, FAIL

Example 17

We illustrate the difference between the ordering of terms in polynomials and series expansions. The ordering of the terms in a polynomial agrees with the ordering of the terms in a series expansion with expansion point `complexInfinity`:

```
f := poly(2*(x^2 + x)^3);
g := series(f, x = complexInfinity);
[lcoeff(f), lmonomial(f), lterm(f)];
[lcoeff(g), lmonomial(g), lterm(g)];
```

$$\mathrm{poly}(2\,x^6 + 6\,x^5 + 6\,x^4 + 2\,x^3, [x])$$

$$2\,x^6 + 6\,x^5 + 6\,x^4 + 2\,x^3 + O(1)$$

$$[2, \mathrm{poly}(2\,x^6, [x]), \mathrm{poly}(x^6, [x])]$$

$$[2, 2\,x^6, x^6]$$

```
[nthcoeff(f, 2), nthmonomial(f, 3), nthterm(f, 4)];
[nthcoeff(g, 2), nthmonomial(g, 3), nthterm(g, 4)];
```

$$[6, \mathrm{poly}(6\,x^4, [x]), \mathrm{poly}(x^3, [x])]$$

$$[6, 6\,x^4, x^3]$$

For finite expansion points, however, the ordering of the terms in a series expansion is the reverse of the ordering of the terms in the corresponding polynomial:

```
h := series(f, x = 0);
[lcoeff(h), lmonomial(h), lterm(h)];
[nthcoeff(h, 2), nthmonomial(h, 3), nthterm(h, 4)];
```

$$2\,x^3 + 6\,x^4 + 6\,x^5 + 2\,x^6 + O(x^9)$$

$$[2, 2\,x^3, x^3]$$

$$[6, 6\,x^5, x^6]$$

Example 18

The method `iszero` checks whether a series expansion has no non-zero summands apart from the *O*-term:

```
f := series(exp(x), x);
g := Series::Puiseux(0, x = 2, 4);
iszero(f), iszero(g)
```

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + O(x^6)$$

$$O((x-2)^4)$$

FALSE, TRUE

Example 19

The methods `convert` tries to convert an arbitrary object into an element of `Series::Puiseux`. If the input does not suggest an expansion point, `convert` uses the origin:

```
f := asympt(1/(x + 1), x = infinity);
g := sin(x)/(1 - x);
h := poly((x + 1)^10);
u := O((x - 1)^3, x = 1);
```

$$\frac{1}{x} - \frac{1}{x^2} + \frac{1}{x^3} - \frac{1}{x^4} + \frac{1}{x^5} - \frac{1}{x^6} + O\left(\frac{1}{x^7}\right)$$

$$-\frac{\sin(x)}{x-1}$$

$$\text{poly}(x^{10} + 10\,x^9 + 45\,x^8 + 120\,x^7 + 210\,x^6 + 252\,x^5 + 210\,x^4 + 120\,x^3 + 45\,x^2 + 10\,x + 1, [x])$$

$$O((x-1)^3, x = 1)$$

```
domtype(f), domtype(g), domtype(h), domtype(u)
```

Series::gseries, DOM_EXPR, DOM_POLY, O

```
F := Series::Puiseux::convert(f);
G := Series::Puiseux::convert(g);
H := Series::Puiseux::convert(h);
U := Series::Puiseux::convert(u);
```

$$\frac{1}{x} - \frac{1}{x^2} + \frac{1}{x^3} - \frac{1}{x^4} + \frac{1}{x^5} - \frac{1}{x^6} + O\left(\frac{1}{x^7}\right)$$

$$\sin(x) + x\sin(x) + x^2\sin(x) + x^3\sin(x) + x^4\sin(x) + x^5\sin(x) + O(x^6)$$

$$1 + 10x + 45x^2 + 120x^3 + 210x^4 + 252x^5 + 210x^6 + 120x^7 + 45x^8 + 10x^9 + x^{10} + O(x^{16})$$

$$O((x-1)^3)$$

`convert` returns `FAIL`, if it is unable to convert the input, e.g., because the input contains no or more than one indeterminate:

```
Series::Puiseux::convert(sin(1)),
Series::Puiseux::convert([1, y, 3])
```

FAIL, FAIL

The method `convert_to` tries to convert an element of `Series::Puiseux` into a specified type:

```
Series::Puiseux::convert_to(F, Series::gseries);
Series::Puiseux::convert_to(F, contfrac);
Series::Puiseux::convert_to(G, DOM_EXPR);
Series::Puiseux::convert_to(H, DOM_POLY);
Series::Puiseux::convert_to(H, O);
Series::Puiseux::convert_to(U, O);
```

$$\frac{1}{x} - \frac{1}{x^2} + \frac{1}{x^3} - \frac{1}{x^4} + \frac{1}{x^5} - \frac{1}{x^6} + O\left(\frac{1}{x^7}\right)$$

$$\frac{x^{-1}}{1+\frac{x^{-1}}{1+O\left(x^{-5}\right)}}$$

$$\sin(x)+x^2\sin(x)+x^3\sin(x)+x^4\sin(x)+x^5\sin(x)+x\sin(x)$$

$$\mathrm{poly}(x^{10}+10x^9+45x^8+120x^7+210x^6+252x^5+210x^4+120x^3+45x^2+10x+1,\ [x])$$

$$O(1)$$

$$O((x-1)^3,\ x=1)$$

`convert_to` returns `FAIL`, if it is unable to perform the requested conversion:

```
Series::Puiseux::convert_to(F, 0),
Series::Puiseux::convert_to(F, DOM_LIST)
```

$$\mathrm{FAIL},\ \mathrm{FAIL}$$

Example 20

The method `expr` converts an element of `Series::Puiseux` into an arithmetical expression, discarding the *O*-term. In general, the ordering of the summands is not preserved:

```
f := series(exp(x*y), x);
g := series(ln(x), x = 1, 3);
```

$$1+x\,y+\frac{x^2\,y^2}{2}+\frac{x^3\,y^3}{6}+\frac{x^4\,y^4}{24}+\frac{x^5\,y^5}{120}+O(x^6)$$

$$x-1-\frac{(x-1)^2}{2}+\frac{(x-1)^3}{3}+O((x-1)^4)$$

```
expr(f);
expr(g);
```

$$\frac{x^5\,y^5}{120}+\frac{x^4\,y^4}{24}+\frac{x^3\,y^3}{6}+\frac{x^2\,y^2}{2}+x\,y+1$$

$$x-\frac{(x-1)^2}{2}+\frac{(x-1)^3}{3}-1$$

The method `float` applies the system function `float` to all coefficients:

```
float(f);
float(g);
```

$$1.0+x\,y+0.5\,x^2\,y^2+0.1666666667\,x^3\,y^3+0.04166666667\,x^4\,y^4+0.008333333333\,x^5\,y^5+O(x^6)$$

$$1.0\,(x-1)-0.5\,(x-1)^2+0.3333333333\,(x-1)^3+O((x-1)^4)$$

Example 21

The methods `combine`, `expand`, and `normal` apply the corresponding system functions to all coefficients:

```
delete a, y:
f := series(y/(x + y^a), x, 4);
g := combine(f);
expand(g);
```

$$\frac{y}{y^a}-\frac{x\,y}{y^{2\,a}}+\frac{x^2\,y}{y^{3\,a}}-\frac{x^3\,y}{y^{4\,a}}+O(x^4)$$

$$y^{1-a}-x\,y^{1-2\,a}+x^2\,y^{1-3\,a}-x^3\,y^{1-4\,a}+O(x^4)$$

$$\frac{y}{y^a} - \frac{x\,y}{y^{2\,a}} + \frac{x^2\,y}{y^{3\,a}} - \frac{x^3\,y}{y^{4\,a}} + O(x^4)$$

For efficiency reasons, the arithmetical methods of `Series::Puiseux` usually do not perform any symbolic simplifications. Use `expand` or `normal` to simplify the results:

```
h := series(exp(x), x, 4)^a;
expand(h);
normal(h);
```

$$1 + a\,x + x^2\left(\frac{a}{2} + \frac{a\,(a-1)}{2}\right) + x^3\left(\frac{a}{6} + \frac{\left(\frac{a}{2} + \frac{a\,(a-1)}{2}\right)(a-2)}{3} + \frac{a\,(2\,a-1)}{6}\right) + O(x^4)$$

$$1 + a\,x + \frac{a^2\,x^2}{2} + \frac{a^3\,x^3}{6} + O(x^4)$$

$$1 + a\,x + \frac{a^2\,x^2}{2} + \frac{a^3\,x^3}{6} + O(x^4)$$

```
u := series(arctanh(x + y), x, 4);
normal(u);
```

$$\operatorname{arctanh}(y) - \frac{x\,\mathrm{i}}{y^2\,\mathrm{i} - \mathrm{i}} + \frac{x^2\,y}{(y^2-1)^2} - \frac{x^3\,(3\,y^2+1)}{3\,(y^2-1)^3} + O(x^4)$$

$$\operatorname{arctanh}(y) - \frac{x}{y^2-1} + \frac{x^2\,y}{(y^2-1)^2} - \frac{x^3\,(3\,y^2+1)}{3\,(y^2-1)^3} + O(x^4)$$

Besides normalizing the coefficients, the method `normal` also removes leading and trailing zeroes from the coefficient list:

```
v := Series::Puiseux::create(1, 3, 10,
                             [0, 1/2, 0, 5, 0, 0], x, 2);
coeff(v);
normal(v);
```

```
coeff(%);
```

$$\frac{(x-2)^4}{2} + 5\,(x-2)^6 + O((x-2)^{10})$$

$$0,\ \frac{1}{2},\ 0,\ 5,\ 0,\ 0$$

$$\frac{(x-2)^4}{2} + 5\,(x-2)^6 + O((x-2)^{10})$$

$$\frac{1}{2},\ 0,\ 5$$

The method `map` applies a given function to all coefficients. E.g., the system function `factor` is not overloaded for `Series::Puiseux`, but you can use `map` to express all coefficients in factored form:

```
map(u, factor);
```

$$\operatorname{arctanh}(y) - \frac{x}{(y-1)\,(y+1)} + \frac{y\,x^2}{(y-1)^2\,(y+1)^2} - \frac{(3\,y^2+1)\,x^3}{3\,(y-1)^3\,(y+1)^3} + O(x^4)$$

In the next example, we use `map` to multiply all coefficients of a series expansion by a constant:

```
w := series(exp(x), x, 3);
map(w, _mult, PI) = PI*w
```

$$1 + x + \frac{x^2}{2} + O(x^3)$$

$$\pi + \pi\,x + \frac{\pi\,x^2}{2} + O(x^3) = \pi + \pi\,x + \frac{\pi\,x^2}{2} + O(x^3)$$

For series expansions of type 1, `map` applies the function to all non-zero coefficients as returned by `coeff`:

```
z := series(sin(sqrt(-x)), x);
coeff(z);
map(z, cos);
```

$$\sqrt{-x} - \frac{(-x)^{3/2}}{6} + \frac{(-x)^{5/2}}{120} + O\left(x^{7/2}\right)$$

$$\frac{\sqrt{-x}}{\sqrt{x}},\ 0,\ -\frac{(-x)^{3/2}}{6\,x^{3/2}},\ 0,\ \frac{(-x)^{5/2}}{120\,x^{5/2}}$$

$$\sqrt{x}\,\cos\left(\frac{\sqrt{-x}}{\sqrt{x}}\right) + x^{3/2}\cos\left(\frac{(-x)^{3/2}}{6\,x^{3/2}}\right) + x^{5/2}\cos\left(\frac{(-x)^{5/2}}{120\,x^{5/2}}\right) + O\left(x^{7/2}\right)$$

Example 22

Three different methods can be used to substitute for the series variable: `_fconcat`, `func_call`, and `subs`. Suppose `f` is an element of `Series::Puiseux` and we want to substitute an expression `t` for the series variable `x`. Then `_fconcat` converts `t` into a series expansion around the origin via the constructor, computes the functional composition, and returns the result as an element of `Series::Puiseux`:

```
f := series(exp(x), x = 0, 5);
Series::Puiseux::_fconcat(f, y) = f @ y;
f @ (y^2 + y);
```

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + O(x^5)$$

$$1 + y + \frac{y^2}{2} + \frac{y^3}{6} + \frac{y^4}{24} + O(y^5) = 1 + y + \frac{y^2}{2} + \frac{y^3}{6} + \frac{y^4}{24} + O(y^5)$$

$$1 + y + \frac{3\,y^2}{2} + \frac{7\,y^3}{6} + \frac{25\,y^4}{24} + O(y^5)$$

The composition may fail if the limit point of `t` around the origin differs from the expansion point of `f` or if `t` contains special mathematical functions:

```
f @ (y + 1);
```

FAIL

```
f @ sin(y);
```

FAIL

Moreover, the composition does not work if the expression `t` is constant or contains more than one indeterminate:

```
f @ PI;
```

```
Error: Cannot compute composition. [Series::Puiseux::_fconcat]
```

```
f @ (x + y);
```

```
Error: Cannot compute composition. [Series::Puiseux::_fconcat]
```

You can enforce the composition by explicitly converting `t` into a series:

```
f @ series(y + 1, y = -1);
f @ series(sin(y), y = 0);
f @ series(x + y, x = -y);
```

$$1 + y + 1 + \frac{(y+1)^2}{2} + \frac{(y+1)^3}{6} + \frac{(y+1)^4}{24} + O((y+1)^5)$$

$$1 + y + \frac{y^2}{2} - \frac{y^4}{8} + O(y^5)$$

$$1 + x + y + \frac{(x+y)^2}{2} + \frac{(x+y)^3}{6} + \frac{(x+y)^4}{24} + O((x+y)^5)$$

Substitution with `func_call` always works. It discards the error term, `t` is substituted literally, and the result is an expression and not an object of type `Series::Puiseux`:

```
f(5) = Series::Puiseux::func_call(f, 5);
f(y) = Series::Puiseux::func_call(f, y);
```

$$\frac{523}{8} = \frac{523}{8}$$

$$\frac{y^4}{24} + \frac{y^3}{6} + \frac{y^2}{2} + y + 1 = \frac{y^4}{24} + \frac{y^3}{6} + \frac{y^2}{2} + y + 1$$

```
f(y^2 + y);
f(y + 1);
f(sin(y));
f(PI);
f(x + y);
```

$$y + \frac{(y^2+y)^2}{2} + \frac{(y^2+y)^3}{6} + \frac{(y^2+y)^4}{24} + y^2 + 1$$

$$y + \frac{(y+1)^2}{2} + \frac{(y+1)^3}{6} + \frac{(y+1)^4}{24} + 2$$

$$\frac{\sin(y)^4}{24} + \frac{\sin(y)^3}{6} + \frac{\sin(y)^2}{2} + \sin(y) + 1$$

$$\pi + \frac{\pi^2}{2} + \frac{\pi^3}{6} + \frac{\pi^4}{24} + 1$$

$$x + y + \frac{(x+y)^2}{2} + \frac{(x+y)^3}{6} + \frac{(x+y)^4}{24} + 1$$

Finally, if `subs` is used to substitute for the series variable, only very special substitutions are allowed (see the description of `subs` above for more details). Then a change of variable is performed, and the result is again of type `Series::Puiseux`:

```
subs(f, x = y^2 + y)
```

```
Error: Invalid substitution. [Series::Puiseux::subs]
```

```
subs(f, x = y + 1)
```

$$1 + y + 1 + \frac{(y+1)^2}{2} + \frac{(y+1)^3}{6} + \frac{(y+1)^4}{24} + O((y+1)^5)$$

```
subs(f, x = sin(y))
```

```
Error: Invalid substitution. [Series::Puiseux::subs]
```

```
subs(f, x = PI)
```

```
Error: The substitution is invalid. Exactly one indeterminate is expected. [Series::Pui
```

```
subs(f, x = x + y)
```

```
Error: The substitution is invalid. Exactly one indeterminate is expected. [Series::Pui
```

All three methods can handle the case where the series variable occurs in the coefficients:

```
s := series(ln(x^2 + x), x);
s @ (2*y);
s(2*y);
subs(s, x = 2*y);
```

$$\ln(x) + x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} + O(x^6)$$

$$\ln(2) + \ln(y) + 2\,y - 2\,y^2 + \frac{8\,y^3}{3} - 4\,y^4 + \frac{32\,y^5}{5} + O(y^6)$$

$$2\,y + \ln(2\,y) - 2\,y^2 + \frac{8\,y^3}{3} - 4\,y^4 + \frac{32\,y^5}{5}$$

$$\ln(2\,y) + 2\,y - 2\,y^2 + \frac{8\,y^3}{3} - 4\,y^4 + \frac{32\,y^5}{5} + O(y^6)$$

Of course, `subs` can also be used to substitute for other objects than the series variable in the coefficients and in the expansion point:

```
g := series(cos(x + y), x, 4);
h := series(1/x, x = y, 4);
```

$$\cos(y) - x\sin(y) - \frac{x^2\cos(y)}{2} + \frac{x^3\sin(y)}{6} + O(x^4)$$

$$\frac{1}{y} - \frac{x-y}{y^2} + \frac{(x-y)^2}{y^3} - \frac{(x-y)^3}{y^4} + O((x-y)^4)$$

```
subs(g, y = PI) = series(cos(x + PI), x, 4);
subs(h, y = 2) = series(1/x, x = 2, 4);
```

$$-1 + \frac{x^2}{2} + O(x^4) = -1 + \frac{x^2}{2} + O(x^4)$$

$$\frac{1}{2} - \frac{x-2}{4} + \frac{(x-2)^2}{8} - \frac{(x-2)^3}{16} + O((x-2)^4) = \frac{1}{2} - \frac{x-2}{4} + \frac{(x-2)^2}{8} - \frac{(x-2)^3}{16} + O((x-2)^4)$$

Even simultaneous substitutions are possible in the coefficients:

```
subs(g, [hold(sin) = cos, hold(cos) = sin, y = 2])
```

$$\sin(2) - x\cos(2) - \frac{x^2\sin(2)}{2} + \frac{x^3\cos(2)}{6} + O(x^4)$$

An error occurs, if the right hand side contains the series variable:

```
subs(h, y = x)
```

```
Error: The substitution is invalid. The right side must not contain the series variable
```

Example 23

The method `has` checks, whether an object occurs syntactically in the coefficients, the series variable, the expansion point, or the direction of an element of `Series::Puiseux`:

```
f := series(sin(x + 2*y), x = PI, 2);
has(f, x), has(f, y), has(f, PI), has(f, 2), has(f, Undirected);
has(f, hold(sin)), has(f, 3), has(f, sin(2*y)), has(f, x - PI);
```

$$-\sin(2\,y)+\cos(2\,y)\,(\pi-x)+O((\pi-x)^2)$$

TRUE, TRUE, TRUE, TRUE, TRUE

TRUE, FALSE, TRUE, FALSE

The last call returns `FALSE` since the expression `x - PI` occurs only in the screen output, but not in the internal representation of `f`.

```
g := series(sign(x), x, Right);
has(g, Right), has(g, Undirected);
```

$$1+O(x^6)$$

TRUE, FALSE

Example 24

The method `truncate` discards summands up to the given order:

```
f := series(x*sin(sqrt(x)), x);
Series::Puiseux::truncate(f, 10);
Series::Puiseux::truncate(f, 9/2);
Series::Puiseux::truncate(f, 7/2);
Series::Puiseux::truncate(f, 3);
Series::Puiseux::truncate(f, 3/2);
Series::Puiseux::truncate(f, 1);
```

$$x^{3/2} - \frac{x^{5/2}}{6} + \frac{x^{7/2}}{120} + O\left(x^{9/2}\right)$$

$$x^{3/2} - \frac{x^{5/2}}{6} + \frac{x^{7/2}}{120} + O\left(x^{9/2}\right)$$

$$x^{3/2} - \frac{x^{5/2}}{6} + \frac{x^{7/2}}{120} + O\left(x^{9/2}\right)$$

$$x^{3/2} - \frac{x^{5/2}}{6} + O\left(x^{7/2}\right)$$

$$x^{3/2} - \frac{x^{5/2}}{6} + O\left(x^{7/2}\right)$$

$$O\left(x^{3/2}\right)$$

$$O\left(x^{3/2}\right)$$

Parameters

f

An arithmetical expression representing a function in x

x

An identifier

x_0

The expansion point: an arithmetical expression. If not specified, the default expansion point 0 is used.

`order`

The number of terms to be computed: a nonnegative integer. The default order is given by the environment variable `ORDER` (default value 6).

`dir`

Either `Left`, `Right`, `Real`, or `Undirected`. This optional argument can be used to specify that the resulting expansion is possibly valid along the real line only. The default is `Undirected`, which means that the expansion is valid in a neighborhood of the expansion point in the complex plane.

Return Values

an object of domain type `Series::Puiseux`, or the value `FAIL`, if the `f` cannot be converted, e.g., if powers with non-rational exponents occur in `f`.

Function Calls

Calling an element of `Series::Puiseux` as a function discards the error term and substitutes the first argument for the series variable. See the description of the method `"func_call"` and “Example 22” on page 7-444.

Operations

`Series::Puiseux` implements the basic arithmetic of truncated series expansions. Use the ordinary arithmetical operators `+, -, *, /, ^`, and `@` for composition.

The arithmetical methods of `Series::Puiseux` usually do not perform any symbolic simplifications. Use `combine`, `expand`, or `normal` to request such simplifications explicitly.

See “Example 5” on page 7-418 and “Example 21” on page 7-441.

Special mathematical functions, such as `exp` or `sin`, are overloaded for elements of `Series::Puiseux`; cf. “Example 10” on page 7-427.

The system functions `coeff`, `lcoeff`, `nthcoeff`, `ldegree`, `lmonomial`, `nthmonomial`, `lterm`, and `nthterm` work on truncated series expansions. Note that in contrast to polynomials, coefficients, monomials, and terms are counted from the term of lowest order term on. Cf. “Example 17” on page 7-436.

Use the function `expr` to convert a series expansion to an arithmetical expression (as an element of a kernel domain).

Operands

A series of the domain type `Series::Puiseux` has the following seven operands:

1. a *type flag* $t \in \{0, 1\}$,
2. the *branching order* b, a positive integer,
3. an integer v such that $\frac{v}{b}$ is the order of the leading term,
4. an integer $e \geq v$ such that $\frac{e}{b}$ is the order of the error term,
5. a list of coefficients $l_1, \ldots, l_n$,
6. the *series variable* x and the *expansion point* x_0 in form of an equation $x = x_0$; the expansion point x_0 may be `complexInfinity` as well,
7. a *direction*, `Undirected`, `Real`, `Left`, or `Right`.

The type flag distinguishes between two different internal representations.

If $t = 0$, then the operands above represent the truncated series expansion

$$\sum_{1 \leq i \leq n} \left(l_i (x - x_0)^{\frac{(v+i-1)}{b}} + O\left((x - x_0)^{e/b} \right) \right)$$

.

If the expansion point x_0 is `complexInfinity`, then the operands represent the truncated expansion

$$\sum_{1 \leq i \leq n} \left(\frac{l_i}{x^{\frac{(i+v-1)}{b}}} + O\left(\frac{1}{x^{\frac{e}{b}}} \right) \right)$$

.

- A summand $l_i(x-x_0)^{\frac{(v+i-1)}{b}}$ (or $\frac{l_i}{x^{\frac{(i+v-1)}{b}}}$, respectively) is called a *monomial* of the expansion,
- the power $(x-x_0)^{\frac{(v+i-1)}{b}}$ (or $\frac{1}{x^{\frac{(v+i-1)}{b}}}$, respectively) is called a *term*,
- l_i is the corresponding *coefficient*, and
- the exponent $\frac{(v+i-1)}{b}$ is the *order* of the corresponding term or monomial.

If $t = 1$, then the operands above represent the expansion

$$\sum_{1 \le i \le n} \left(l_i + O\left((x-x_0)^{e/b}\right)\right)$$

.

In this case, the powers of x - x_0 are explicitly stored in the list, and l_i contains only terms of growth order $O\left((x-x_0)^{\frac{(v+i-1)}{b}}\right)$. The corresponding expansion for `x0 = complexInfinity` is

$$\sum_{1 \le i \le n} \left(l_i + O\left(\frac{1}{x^{\frac{e}{b}}}\right)\right)$$

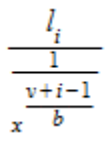
,

and l_i contains only terms of growth order $O\left(\frac{1}{x^{\frac{(v+i-1)}{b}}}\right)$.

The notions term and order are the same as for $t = 0$, a summand l_i is called a monomial, and the corresponding coefficient is

$$\frac{l_i}{(x-x_0)^{\frac{v+i-1}{b}}}$$

(or



,

respectively).

The latter type of representation serves for correct expansions around branch points. For example, if we want to expand $f(x) = \sqrt{x}$ around $x = 0$, then the truncated Puiseux series $\sqrt{-1}\,\sqrt{x} + O\left(x^{13/2}\right)$ does not approximate $f(x)$ in the lower half of the complex plane. With $t = 1$, the expansion $\sqrt{(-x)} + O\left(x^{13/2}\right)$, which approximates $f(x)$ also in the lower part of the complex plane near the origin, can be represented as an object of domain type `Series::Puiseux`.

The direction d has the same meaning as the parameter `dir` of the constructor. If d = *Undirected*, the operands above represent an expansion valid in some neighborhood of the expansion point in the complex plane. Usually, this is an open disc centered at `x0`. If $d \neq$ *Undirected* and x_0 represents a real number, this means that the expansion is valid for real values of x only. If d = *Left* or d = *Right*, then the expansion is valid for $x < x_0$ or $x > x_0$, respectively.

In the case `x_0 = complexInfinity`, and if d = *Undirected*, we have an expansion valid in the neighborhood of the north pole of the Riemann sphere, i.e., for all $x \in \mathbb{C}$ of sufficiently large absolute value. If d = *Left*, we have an expansion around the positive real infinity valid for sufficiently large real values of x. Similarly, if d = *Right*, we have an expansion around the negative real infinity valid for sufficiently large negative real values of x. Finally, if d = *Real*, the expansion is valid both around *infinity* and around $-\infty$.

Cf. “Example 2” on page 7-412.

Element Creation

Typically, objects of type `Series::Puiseux` are generated by calls to `series` or `taylor`.

Methods

Mathematical Methods

`conjugate` — Complex conjugation

```
conjugate(s)
```

This method overloads the system function `conjugate`. Cf. “Example 11” on page 7-428.

`contfrac` — Conversion into a continued fraction

```
contfrac(s)
```

This method overloads the system function `contfrac`. Cf. “Example 12” on page 7-429.

`diff` — Differentiation

```
diff(s, t)
```

This method overloads the system function `diff`. Cf. “Example 9” on page 7-426.

`_divide` — Division

```
_divide(s, t)
```

`_fconcat` — Functional composition

```
_fconcat(s, t)
```

If both `s` and `t` are of type `Series::Puiseux`, then the functional composition can only be defined if the limit point of t for values close to its expansion point is equal to the expansion point of s. Otherwise, an error occurs.

At least one of the arguments must be of type `Series::Puiseux`. If one of the arguments is not of this type, then it is converted into an element of `Series::Puiseux` via the constructor. If s is not of type `Series::Puiseux`, then it is converted into a

series expansion around the limit point of t. If t is not of type `Series::Puiseux`, then it is converted into a series expansion around 0. The implicit conversion is performed only if the corresponding expression contains exactly one free variable.

This method overloads the system function `_fconcat` for series expansions, i.e., you may use it in the form `s@t`. See "Example 8" on page 7-423 and "Example 22" on page 7-444.

`Im` — Imaginary part

```
Im(s)
```

This method overloads the system function `Im`. Cf. "Example 11" on page 7-428.

`int` — Integration

```
int(s, t | t = a .. b)
```

This method overloads the system function `int`. Cf. "Example 9" on page 7-426, and the help page of `int` for a description of further optional arguments.

`_invert` — Reciprocal of a series

```
_invert(s)
```

This method overloads the system function `_invert`, i.e., you may use it in the form `1/s`.

`ilaplace` — Inverse Laplace transform

```
ilaplace(s, u, v)
```

If `u` is not the series variable of `s`, then the coefficients of `s` are transformed, but not the expansion point. Otherwise, the expansion point of `s` must be infinity, `v` must be an identifier, and `s` is transformed term by term. The result is then a series expansion around $v = 0$.

This method overloads the function `ilaplace` for series expansions. Cf. "Example 13" on page 7-430.

`laplace` — Laplace transform

```
laplace(s, u, v)
```

If `u` is not the series variable of `s`, then the coefficients of `s` are transformed, but not the expansion point. Otherwise, the expansion point of `s` must be 0, the order of the leading term of `s` must be nonnegative, `v` must be an identifier, and `s` is transformed term by term. The result is then a series expansion around $v = \infty$.

This method overloads the function `laplace` for series expansions. Cf. "Example 13" on page 7-430.

`_mult` — Multiplication

```
_mult(s, t, , …)
```

Use the method `Series::Puiseux::scalmult` to multiply a series expansion `s` by a constant or a power of $x - x_0$.

This method overloads the system function `_mult` for series expansions, i.e., you may use it in the form `s*t*...`. Cf. "Example 5" on page 7-418.

`_negate` — Negation

```
_negate(s)
```

This method overloads the system function `_negate`, i.e., you may use it in the form `-s`.

`_plus` — Addition

```
_plus(s, t, , …)
```

This method overloads the system function `_plus` for series expansions, i.e., you may use it in the form `s+t+...`. Cf. "Example 5" on page 7-418.

`_power` — Exponentiation

```
_power(s, n)
```

If n is a rational number, the direction of `s` is `Undirected` or `Real`, and the leading coefficient of `s` is not positive, then the type flag of the result is 1 in general.

If n is not a rational number, then the leading summand of s must not contain the series variable. Otherwise, an error occurs.

This method overloads the system function `_power` for series expansions, i.e., you may use it in the form `s^n`. Cf. "Example 7" on page 7-421.

Re — Real part

```
Re(s)
```

This method overloads the system function `Re`. Cf. “Example 11” on page 7-428.

revert — Functional inversion

```
revert(s)
```

The expansion point of the inverse is the limit point of `s`.

This method overloads the system function `revert`. Cf. “Example 8” on page 7-423.

scalmult — Multiplication by a single monomial

```
Series::Puiseux::scalmult(s, a, k)
```

series — Serie expansion

```
series(s, y | y = y0, <order>, <dir>)
```

This method overloads the system function `series`.

_subtract — Subtraction

```
_subtract(s, t)
```

Access Methods

coeff — Extract coefficients

```
coeff(s, <x>, n)
```

```
coeff(s, <x>)
```

The second call returns the sequence of all coefficients of `s`, starting with the coefficient of lowest order. (This is the coefficient of the term with the highest exponent if x_0=`complexInfinity`.)

Specifying the variable `x` is optional; if it is present, it must coincide with the series variable of `s`.

This method overloads the system function `coeff`. Cf. “Example 14” on page 7-431.

`direction` — Direction of expansion

```
Series::Puiseux::direction(s)
```

`indet` — Serie variable

```
Series::Puiseux::indet(s)
```

`iszero` — Zero test

```
iszero(s)
```

This method overloads the system function `iszero`. Cf. “Example 18” on page 7-438.

`lcoeff` — Leading coefficient (of lowest order)

```
lcoeff(s)
```

This method overloads the system function `lcoeff`. See “Example 15” on page 7-433 and “Example 17” on page 7-436.

`ldegree` — Leading degree

```
ldegree(s)
```

This method overloads the system function `ldegree`. See “Example 2” on page 7-412 and “Example 15” on page 7-433.

`lmonomial` — Leading monomial (of lowest order)

```
lmonomial(s)
```

This method overloads the system function `lmonomial`. See “Example 15” on page 7-433 and “Example 17” on page 7-436.

`lterm` — Leading term (of lowest order)

```
lterm(s)
```

This method overloads the system function `lterm`. See “Example 15” on page 7-433 and “Example 17” on page 7-436.

`nthcoeff` — Extract coefficients

```
nthcoeff(s, n)
```

This method overloads the system function `nthcoeff`. See “Example 16” on page 7-435 and “Example 17” on page 7-436

`nthmonomial` — Extract monomials

```
nthmonomial(s, n)
```

This method overloads the system function `nthmonomial`. See “Example 16” on page 7-435 and “Example 17” on page 7-436.

`nthterm` — Extract terms

```
nthterm(s, n)
```

This method overloads the system function `nthterm`. See “Example 16” on page 7-435 and “Example 17” on page 7-436.

`order` — Order of the error term

```
Series::Puiseux::order(s)
```

`point` — Expansion point

```
Series::Puiseux::point(s)
```

Conversion Methods

`convert` — Convert any object into a series expansion

```
convert(f)
```

If no expansion point can be determined from `f`, the origin is used. Cf. “Example 19” on page 7-438.

`convert_to` — Convert a series expansion into another domain

```
convert_to(s, T)
```

Use the function `expr` to convert `s` into an object of a kernel domain.

`convert01` — Convert into a series expansion of type 1

```
Series::Puiseux::convert01(s)
```

`convert10` — Try to convert into a series expansion of type 0

```
Series::Puiseux::convert10(s)
```

For undirected expansions, the conversion is not possible in general, and then `s` is returned. However, you can enforce a conversion (with a not necessarily equivalent result) by using properties. Cf. “Example 3” on page 7-415.

`expr` — Convert a series expansion into an element of a kernel domain

```
expr(s)
```

This method overloads the system function `expr`. Cf. “Example 20” on page 7-440.

`float` — Convert numeric parts of the coefficients into floats

```
float(s)
```

This method overloads the system function `float`. Cf. “Example 20” on page 7-440.

Technical Methods

`combine` — Combine coefficients

```
combine(s)
```

This method overloads the system function `combine`; see the corresponding help page for further optional arguments. Cf. “Example 21” on page 7-441.

`const` — Convert a constant expression into a truncated series

```
Series::Puiseux::const(f, x | x = x0, n, <d>)
```

If the expansion point `x0` is omitted, `x0 = 0` is assumed. If the direction `d` is omitted, `d = Undirected` is assumed.

Use with care, since this function does not perform type checking. Cf. “Example 1” on page 7-409.

`create` — Syntactical constructor

```
Series::Puiseux::create(b, v, e, l, x, <x0>, <d>)
```

If the expansion point `x0` is omitted, `x0 = 0` is assumed. If the direction `d` is omitted, `d = Undirected` is assumed.

Use with care, since this function does not perform type checking. Cf. “Example 1” on page 7-409.

`expand` — Expand coefficients

```
expand(s)
```

This method overloads the system function `expand`; see the corresponding help page for further optional arguments. Cf. “Example 21” on page 7-441.

`func_call` — Evaluation at a point

```
Series::Puiseux::func_call(s, t)
```

You may also use this method in the form `s(t)`. Cf. “Example 22” on page 7-444.

`has` — Check whether an object occurs syntactically

```
has(s, t)
```

This method overloads the system function `has`. Cf. “Example 23” on page 7-449.

`map` — Apply a function to all non-zero coefficients

```
map(s, f, <arg1, , …>)
```

This method overloads the system function `map`. Cf. “Example 21” on page 7-441.

`normal` — Normal form

```
normal(s)
```

This method overloads the system function `normal`. Cf. “Example 21” on page 7-441.

one — Create a truncated series with constant term 1

```
Series::Puiseux::one(x, <x0>, n, <d>)
```

If the expansion point `x0` is omitted, `x0 = 0` is assumed. If the direction `d` is omitted, `d = Undirected` is assumed.

Use with care, since this function does not perform type checking. Cf. "Example 1" on page 7-409.

print — Pretty-print routine

```
print(s)
```

truncate — Truncate a series expansion

```
Series::Puiseux::truncate(s, n)
```

Cf. "Example 24" on page 7-449.

subs — Replace subexpressions

```
subs(s, old = new)
```

```
subs(s, [old1 = new1, old2 = new2, …])
```

If the series variable `x` of `s` does not occur in the left hand sides `old, old1, old2, ...`, then the substitution takes place in the coefficients and in the expansion point of `s`. The series variable must not occur in the right hand sides `new, new1, new2, ...`.

Note: In contrast to the usual behavior of `subs`, the result of the substitution is subjected to an additional evaluation.

In the second call, the series variable `x` of `s` must not occur anywhere in the substitution equations. In the first call, `x` is allowed to occur in `old` only if `old` equals `x`. In this case, a change of variable is performed, and `new` must be of the form

```
x0 + a*(b*y - c)^k
```

if `x0 <> complexInfinity` and

```
a*(b*y - c)^k
```

if `x0 = complexInfinity`, where

- `x0` is the expansion point of `s`
- `k` is a non-zero rational number
- `y` is an identifier, which may well be equal to `x`, and otherwise `y` must not occur in the coefficients of `s`.
- `a, b, c` are arithmetical expressions not involving `y`, with `a, b` being non-zero. If the direction of `s` is not `Undirected`, then `a` and `b` must represent real numbers.
- `c` is zero if either `x0 <> complexInfinity` and `k` is positive or `x0 = complexInfinity` and `k` is negative. In this case, the result of the substitution has expansion point `complexInfinity`.
- If `c` is non-zero, then the result of the substitution has expansion point `c/b`.

Use one of the methods `"_fconcat"` or `"func_call"` for more general substitutions for the series variable.

This method overloads the system function `subs`; Cf. “Example 22” on page 7-444.

`zero` — Create a truncated series with an error term only

```
Series::Puiseux::zero(x, <x0>, n, <d>)
```

If the expansion point `x0` is omitted, `x0 = 0` is assumed. If the direction `d` is omitted, `d = Undirected` is assumed.

Note: Although `Series::Puiseux::zero(x, n)` and `O(x^n)` are mathematically equivalent and are printed in the same way, they are different MuPAD objects. The former is an element of type `Series::Puiseux`, while the latter is an element of type `O`.

Use with care, since this function does not perform type checking. Cf. “Example 1” on page 7-409.

See Also

MuPAD Functions

`asympt` | `series`

MuPAD Domains
`Series::gseries`

Series::gseries

Generalized series expansions

Syntax

```
Series::gseries(f, x, <order>, <Left | Right>)
```

```
Series::gseries(f, x = a, <order>, <Left | Right>)
```

Description

`Series::gseries` is the domain of series expansions generalizing Taylor, Laurent and Puiseux expansions.

The call `Series::gseries(f, x)` computes a series expansion at the right hand side of $x = 0$.

The system functions `series` and `asympt` are the main application of this domain. The latter function only returns elements of this domain, whereas `series` can return an element of `Series::gseries` in cases, where a Puiseux series expansion does not exist.

There may be no need to explicitly create elements of this domain, but to work with the results of the mentioned system functions.

See the help page of the system function `asympt` for a detailed description of the parameters and examples for working with elements of the domain `Series::gseries`.

Note: Note that elements of `Series::gseries` only represents *directional* (real) series expansions.

Environment Interactions

The function is sensitive to the global variable `ORDER`, which determines the default number of terms of the expansion.

Parameters

`f`

An arithmetical expression

`x`

The series variable: an identifier

`a`

The expansion point: an arithmetical expression or ±`infinity`

`order`

The truncation order: a nonnegative integer

Options

`Left`

Compute a series expansion that is valid for real x smaller than a.

`Right`

Compute a series expansion that is valid for real x larger than a (the default case).

Return Values

Object of domain type `Series::gseries`, or the value `FAIL`.

Function Calls

Calling an element of `Series::gseries` as a function yields the object itself, regardless of the arguments. The arguments are *not* evaluated.

Operations

`Series::gseries` implements standard arithmetic of generalized series expansions. Use the ordinary arithmetical operators `+`, `-`, `*`, `/`, and `^`.

The system functions `coeff`, `lcoeff`, `nthcoeff`, `lterm`, `nthterm`, `lmonomial`, `nthmonomial`, and `ldegree` work on generalized series expansions. See the corresponding help pages of these functions for calling parameters. See the description of these methods below for further details.

The method `"indet"` returns the series variable of the series expansion, i.e., if `s` is an object of the domain `Series::gseries`, then `s::dom::indet(s)` returns the series variable.

The method `"point"` returns the expansion point of the series.

Use the function `expr` to convert a generalized series expansion into an arithmetical expression (as an element of a kernel domain).

Operands

A series of the domain type `Series::gseries` consists of four operands:

1. A list of pairs $[c_i, f_i]$. Each pair represents a *monomial* $c_i f_i$ of the series expansion, where the c_i are the *coefficients* and f_i the *terms* of `s`. The coefficients do not contain the series variable.

   This list can be empty, if the order of the expansion is zero.
2. An arithmetical expression `g` representing the *error term* of the form $O(g)$. It may be the integer 0, in which case the expansion is exact.
3. The *series variable* `x`.
4. The *expansion point* `a`.

Methods

Mathematical Methods

`_divide` — Divide two series expansions

```
_divide(s, t)
```

If the arguments are not of domain type `Series::gseries`, then they are converted into such objects. `FAIL` is returned, if one of these conversions fails.

This method overloads the function `_divide` for elements of `Series::gseries`, i.e., you may use it in the form `s/t`.

`_invert` — Multiplicative inverse of a series expansion

```
_invert(s)
```

This method overloads the function `_invert` for elements of `Series::gseries`, i.e., you may use it in the form `1/s`.

`_mult` — Multiply series expansions

```
_mult(s, t, …)
```

If both `s` and `t` are series expansions of the domain `Series::gseries`, then the result is a series expansion of the domain `Series::gseries`, too. Both series expansions must have the same series variable and expansion point, otherwise `FAIL` is returned.

If `s` or `t` is a series expansion of the domain `Series::Puiseux`, then it is converted into an object of `Series::gseries`. If this fails, then `FAIL` is returned. Otherwise, the product is computed and returned as an object of the domain `Series::gseries`.

If `s` is a series expansion and `t` is an arithmetical expression, then `t` is converted into a series expansion via the constructor `Series::gseries` (and vice versa).

Each argument of this method that is not of the domain type `Series::gseries` is converted into such an element, i.e., a generalized series expansion is computed. If this fails, then `FAIL` is returned.

This method overloads the function _mult for elements of Series::gseries, i.e., you may use it in the form s*t*....

_negate — Negative of a series expansion

```
_negate(s)
```

This method overloads the function _negate for elements of Series::gseries, i.e., you may use it in the form -s.

_plus — Add series expansions

```
_plus(s, t, …)
```

If both s and t are series expansions of the domain Series::gseries, then the result is a series expansion of the domain Series::gseries, too. Both series expansions must have the same series variable and expansion point, otherwise FAIL is returned.

If s or t is a series expansion of the domain Series::Puiseux, then it is converted into an object of Series::gseries. If this fails, then FAIL is returned. Otherwise, the sum is computed and returned as an object of the domain Series::gseries.

If s is a series expansion and t is an arithmetical expression, then t is converted into a series expansion via the constructor Series::gseries (and vice versa).

Each argument of this method that is not of the domain type Series::gseries is converted into such an element, i.e., a generalized series expansion is computed. If this fails, then FAIL is returned.

This method overloads the function _plus for elements of Series::gseries, i.e., you may use it in the form s+t+ ....

_power — Exponentiation of a series expansion

```
_power(s, n)
```

The exponent n must not involve the series variable of s. Otherwise, an error occurs.

If n is a positive integer, then repeated squaring is used for computing the nth power of s. Otherwise, the binomial theorem is applied after factoring out the leading monomial.

This method overloads the function _power for elements of Series::gseries, i.e., you may use it in the form s^n.

`_subtract` — Subtract two series expansions

```
_subtract(s, t)
```

If the arguments are not of domain type `Series::gseries`, then they are converted into such objects. `FAIL` is returned, if one of these conversions fails.

This method overloads the function `_subtract` for elements of `Series::gseries`, i.e., you may use it in the form `s-t`.

Access Methods

`coeff` — Extract coefficients

```
coeff(s, <n>)
```

This method overloads the function `coeff` for elements of `Series::gseries`.

`indet` — Serie variable

```
Series::gseries::indet(s)
```

Use the method `"point"` to get the expansion point of `s`.

`iszero` — Zero test

```
iszero(s)
```

This method overloads the function `iszero` for elements of `Series::gseries`.

`lcoeff` — Leading coefficient

```
lcoeff(s)
```

This method overloads the function `lcoeff` for elements of `Series::gseries`.

`ldegree` — Leading degree

```
ldegree(s)
```

This method overloads the function `ldegree` for elements of `Series::gseries`.

`lmonomial` — Leading monomial

```
lmonomial(s)
```

This method overloads the function `lmonomial` for elements of `Series::gseries`.

`lterm` — Leading term

```
lterm(s)
```

This method overloads the function `lterm` for elements of `Series::gseries`.

`nthcoeff` — Extract a coefficient

```
nthcoeff(s, n)
```

This method overloads the function `nthcoeff` for elements of `Series::gseries`.

`nthmonomial` — Extract a monomial

```
nthmonomial(s, n)
```

This method overloads the function `nthmonomial` for elements of `Series::gseries`.

`nthterm` — Extract a term

```
nthterm(s, n)
```

This method overloads the function `nthterm` for elements of `Series::gseries`.

`point` — Expansion point

```
Series::gseries::point(s)
```

Use the method `"indet"` to get the series variable of `s`.

Conversion Methods

`convert` — Convert an object into a generalized series expansion

```
Series::gseries::convert(x)
```

`convert_to` — Convert a generalized series expansion into other domains

```
Series::gseries::convert_to(s, T)
```

`T` might be the domain `DOM_POLY`, where the sum of monomials is considered as a polynomial in the indeterminates of the third operand of `s`.

If `T` is the domain `DOM_EXPR`, then the conversion is the same as implemented by the method `"expr"` (see below).

If `T` is the domain `Series::Puiseux`, then the system tries to convert `s` into a Puiseux series. If the conversion is not possible, `FAIL` is returned.

Use the function `expr` to convert `s` into an object of a kernel domain.

`create` — Create simple and fast a generalized series expansion

```
Series::gseries::create(list, errorTerm, x = a)
```

Note: This method should be used with caution, because no argument checking is performed. Use it to *create*, not to compute elements of `Series::gseries`.

`expr` — Convert a generalized series expansion into an element of a kernel domain

```
expr(s)
```

This method overloads the function `expr` for elements of `Series::gseries`.

`series` — Apply the function `series` to a generalized series expansion

```
series(s, x | x = x0, <order>, <dir>)
```

This method overloads the function `series` for elements of `Series::gseries`. See the corresponding help page for a description of the possible arguments.

Technical Methods

`combine` — Apply the function `combine` to all terms

```
combine(s, <target>)
```

This method overloads the system function `combine`. See the corresponding help page for a description of the optional argument `target`.

`has` — Check whether an object occurs syntactically

```
has(s, t)
```

This method overloads the system function `has`.

`map` — Map a function to the coefficients

```
map(s, func, …)
```

This method overloads the function `map` for elements of `Series::gseries`.

`print` — Pretty-print routine

```
print(s)
```

`subs` — Substitute into a generalized series expansion

```
subs(s, x = a, …)
```

This method overloads the function `subs` for elements of `Series::gseries`.

`TeX` — LaTeX formatting

```
Series::gseries::TeX(s)
```

This method is called by the system function `generate::TeX`.

See Also

MuPAD Domains
`Series::Puiseux`

export – Export Data

export::stl

export::stl

Export STL data

Syntax

```
export::stl(filename, [x, y, z], u = u_min .. u_max, v = v_min .. v_max, options)

export::stl(n, [x, y, z], u = u_min .. u_max, v = v_min .. v_max, options)

export::stl(filename, object_1, <object_2, …>, options)

export::stl(n, object_1, <object_2, …>, options)
```

Description

`export::stl` is used to create a triangulation of a parametrized surface and write the triangulation data in STL format to an external file.

STL files contain triangulation data of 3D surfaces. Each triangle is stored as a unit normal and three vertices. The normal and the vertices are specified by three coordinates each, so there is a total of 12 numbers stored for each triangle. Read the "Background" section of this help page for further details.

If the surface is closed, it is regarded as the boundary of a 3D solid. The normals of the triangles written into the STL file should point from the inside of the body to the outside.

Note: Note that the direction of the normals that `export::stl` writes into the STL file depend on the parametrization $x(u, v)$, $y(u, v)$, $z(u, v)$!

If `p1 = (x(u, v), y(u, v), z(u, v))`, `p2 = (x(u + du, v), y(u + du, v), z(u + du, v))`, `p3 = (x(u, v + dv), y(u, v + dv), z(u, v + dv))` are the corners of a triangle, the normal associated with this triangle is the cross product of the side `p2 - p1` times the side `p3 - p2`. The routine `export::stl` chooses neighboring values of the surface parameters with `du = (`u_{max}` - `u_{min}`)/(n_u - 1)` and `dv = (`v_{max}` - `v_{min}`)/(n_v - 1)`, respectively.

Note: Thus, if your parametrization is such that the cross product of the vectors $p_2 - p_1$ and $p_3 - p_2$ does not point to the outside of your body, you just need to let one of the parameters (u, say) run from u_{max} to u_{min} instead of from u_{min} to u_{max}. Just replace your call

```
export::stl(filename, [x,y,z], u = `u_{min}` .. `u_{max}`, v =
`v_{min}` .. `v_{max}`)
```

by

```
export::stl(filename, [x,y,z], u = `u_{max}` .. `u_{min}`, v =
`v_{min}` .. `v_{max}`).
```

Up to the irrelevant ordering in the STL file, the triangles generated by these calls are the same apart from the direction of the normal associated with each triangle.

If the file is specified by a character string, the corresponding file is opened and closed, automatically.

As an alternative to specifying the file by a string, the user may open the file herself via `fopen` in `Write` mode and pass the file descriptor returned by `fopen` to `export::stl`. If binary data are to be written to the file, make sure that it is opened with the `Raw`, i.e., call `fopen(filename, Write, Raw)`.

Note: Note that `export::stl` does not close the file automatically if it is specified by a file descriptor. It remains open after `export::stl` has finished its job. The file needs to be closed explicitly by the user using `fclose`.

If the file is specified by a character string, the name may correspond to an absolute or a relative path name. In particular, the environment variable `WRITEPATH` is taken into account. The details on the help page of `fopen` hold for `export::stl`, too.

Note: With the option `Append`, the file is first opened for reading and, after reading of the data in the file, opened for writing. If no absolute pathname is used to specify the file, make sure that the environment variables `READPATH` and `WRITEPATH` point to the same folder. Alternatively, it is a good idea to place the file in the same folder as the MuPAD notebook which you are currently using. If this notebook is saved on

the disk of your computer, the absolute path is available as the environment variable NOTEBOOKPATH. Thus, specifying a file named "myfile.stl", say, by the absolute path name NOTEBOOKPATH."myfile.stl" ensures that the file is found in the same folder as your notebook.

Text files generated with the option Text or the equivalent Ascii can be opened and read with any text editor. However, binary files generated with the option Bin or the equivalent options Binary or Raw are faster to create and to process.

The file generated by export::stl can be read and visualized in MuPAD using the plot primitive plot::SurfaceSTL.

If the file name given ends in ".gz", export::stl writes a compressed file which can be read by any program supporting gzip compression.

Environment Interactions

The function is sensitive to the environment variable DIGITS which determines the numerical working precision. The current value of DIGITS determines the number of significant decimal digits with which the STL data are written to the specified file. (This holds for text files. In binary STL files all numerical values have a precision of about 7 decimal digits.) For the internal computation of the data by MuPAD, the value of DIGITS is temporarily increased by 10 to minimize round-off effects.

The STL data generated by export::stl are written to the specified file.

Examples

Example 1

We generate a sphere given by the following parametrization:

```
x:= cos(u)*sin(v):
y:= sin(u)*sin(v):
z:= cos(v):
```

We call export::stl to generate the STL data and write them into a file named "sphere.stl". The file is to be generated in the same directory as the current MuPAD

notebook that we are using. Hence, we specify an absolute path name for the file using the path of the current notebook. If this notebook was saved to the disk of your computer, this path is available in the environment variable NOTEBOOKPATH:

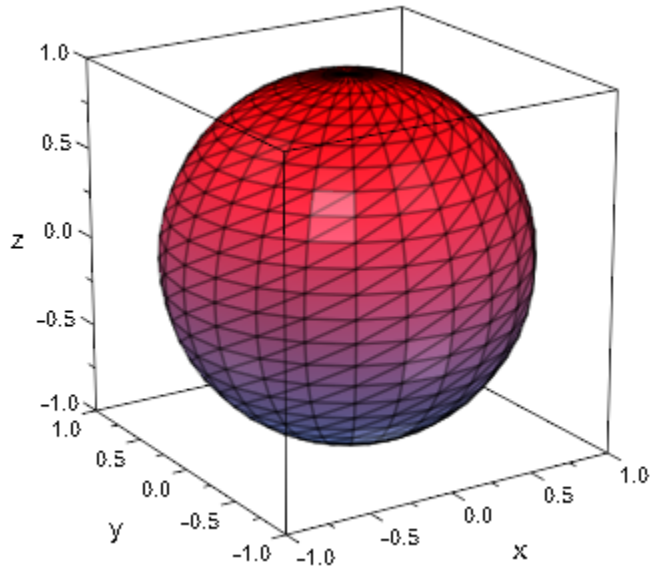
```
filename:= NOTEBOOKPATH."sphere.stl":
export::stl(filename, [x, y, z], u = 0 .. 2*PI, v = 0 .. PI, Text)
```

Since the file was created in Text format, it can be opened with any text editor. It should look like this:

```
solid MuPADtoSTL1
 facet normal -0.06540070486 -0.008610166138 -0.9978219344
  outer loop
   vertex 100.0 100.0 300.0
   vertex 112.607862 103.3782664 298.7167292
   vertex 113.0526192 100.0 298.7167292
  endloop
 endfacet
 facet normal -0.1950260058 -0.02567566076 -0.9804619409
  outer loop
   vertex 113.0526192 100.0 298.7167292
   vertex 112.607862 103.3782664 298.7167292
   vertex 125.0 106.6987298 294.8888739
  endloop
 endfacet

 ...

endsolid MuPADtoSTL1
```

We reimport the STL data and visualize the surface using plot::SurfaceSTL:

```
plot(plot::SurfaceSTL(filename, MeshVisible))
```



We reduce the number of significant output digits to a reasonable size. Further, we specify a mesh size and request a specific output box:

```
DIGITS:= 7:
export::stl(filename, [x, y, z], u = 0..2*PI, v = 0..PI,
            Mesh = [10, 10],
            OutputBox = [-100..100, -100..100, -100..100],
            Text):
```

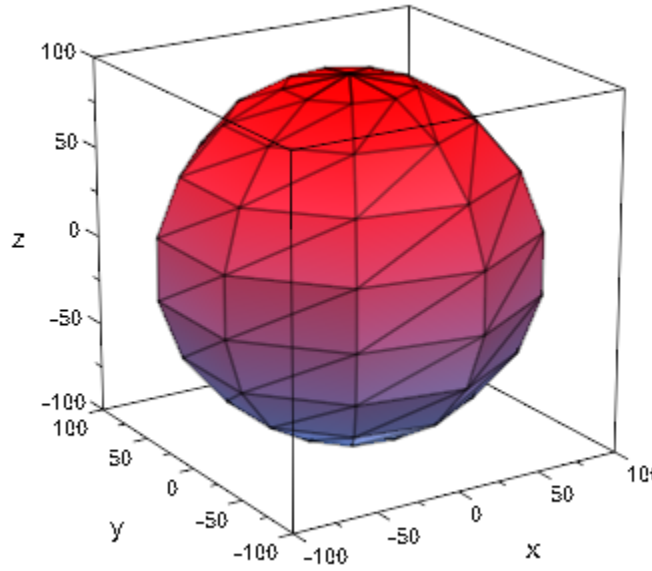
The file now should look like this:

```
solid MuPADtoSTL2
 facet normal -0.1733024 -0.06307691 -0.9828467
  outer loop
   vertex -3.10912 0.000000002143114 100.0
   vertex 24.32249 22.66816 93.96926
   vertex 32.7003 0.000000002143114 93.96926
  endloop
 endfacet

 ...
```

```
endsolid MuPADtoSTL2
```

We visualize the new content of the file:

```
plot(plot::SurfaceSTL(filename, MeshVisible))
```




```
delete x, y, z, filename, DIGITS:
```

Example 2

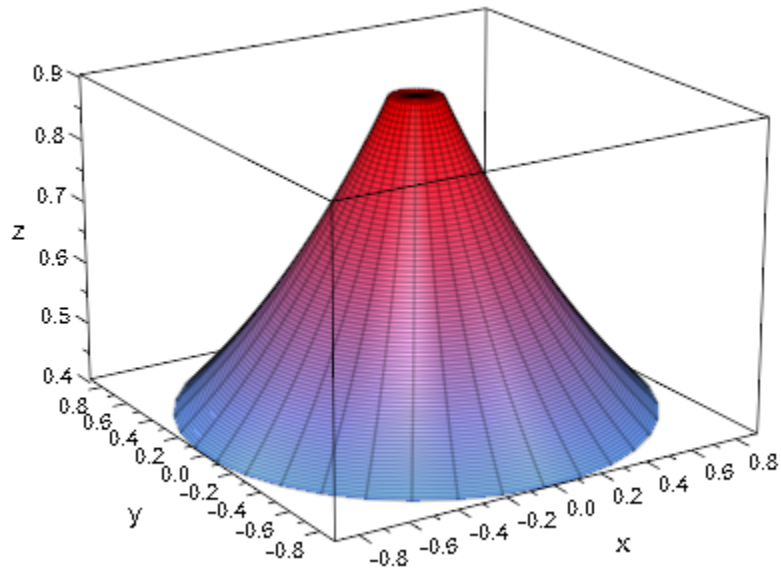
We specify the parametrization of the surface by a mixture of expressions and procedures:

```
x:= piecewise([0.1 < u < 0.9, u*cos(v)], [Otherwise, 0]):
y:= (u, v) -> piecewise([0.1 < u < 0.9, u*sin(v)], [Otherwise, 0]):
z:= (u, v) -> if u <= 0.1 then exp(-0.1)
              elif u < 0.9 then exp(-u)
              else exp(-0.9)
```

```
                end_if:
```

This is the surface that we wish to export to STL:

```
plot(plot::Surface([x, y, z], u = 0..1, v = 0..2*PI,
                                    Mesh = [100, 36])):
```

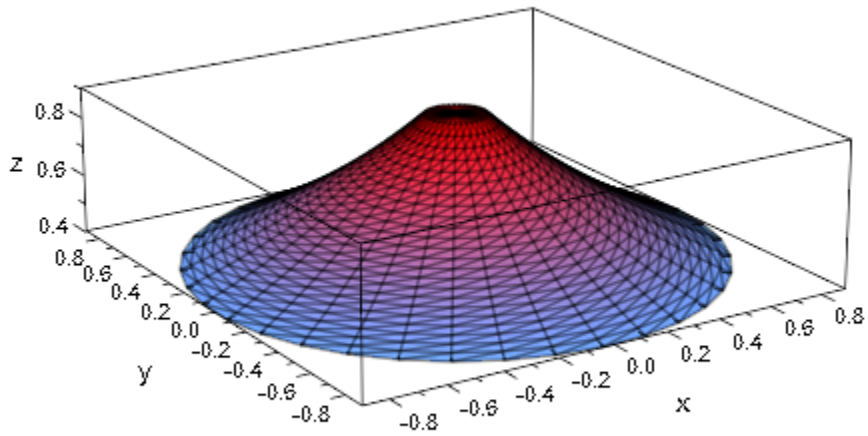


We assume that there is no external file “sample.stl”. We create it by opening it in `Write` mode in the same directory as the current MuPAD notebook that we are using. Hence, we specify an absolute path name for the file using the path of the current notebook. If this notebook was saved to the disk drive of your computer, this path is available in the environment variable `NOTEBOOKPATH`. The file descriptor `n` returned by `fopen` is passed to `export::stl`:

```
filename:= NOTEBOOKPATH."sample.stl":
DIGITS:= 7:
export::stl(filename, [x, y, z], u = 0..1, v = 0..2*PI,
                                        Mesh = [30, 36])
```

We reimport the STL data and visualize the surface using `plot::SurfaceSTL`:

```
plot(plot::SurfaceSTL(filename, MeshVisible))
```

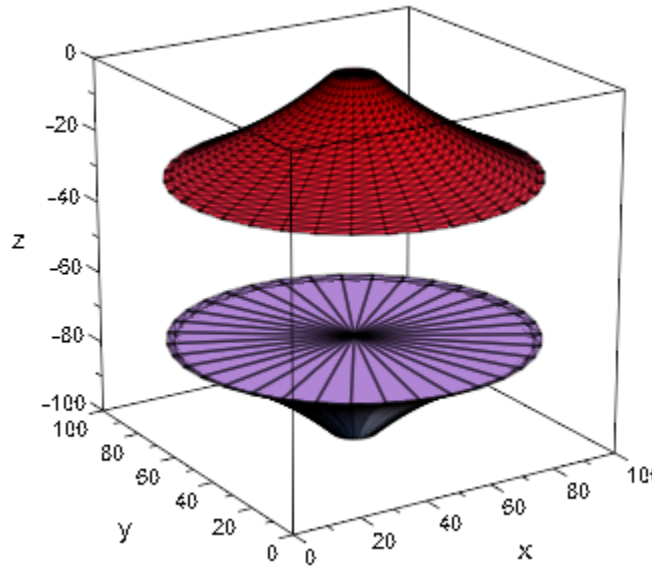


We can append a further surface to the file using the option `Append`:

```
export::stl(filename, [x, y, -z], u = 0..1, v = 0..2*PI,
            Mesh = [30, 36],
            OutputBox = [0..100, 0..100, -100..0],
            Append)
```

We visualize the new content of the file via `plot::SurfaceSTL`:

```
plot(plot::SurfaceSTL(filename, MeshVisible))
```

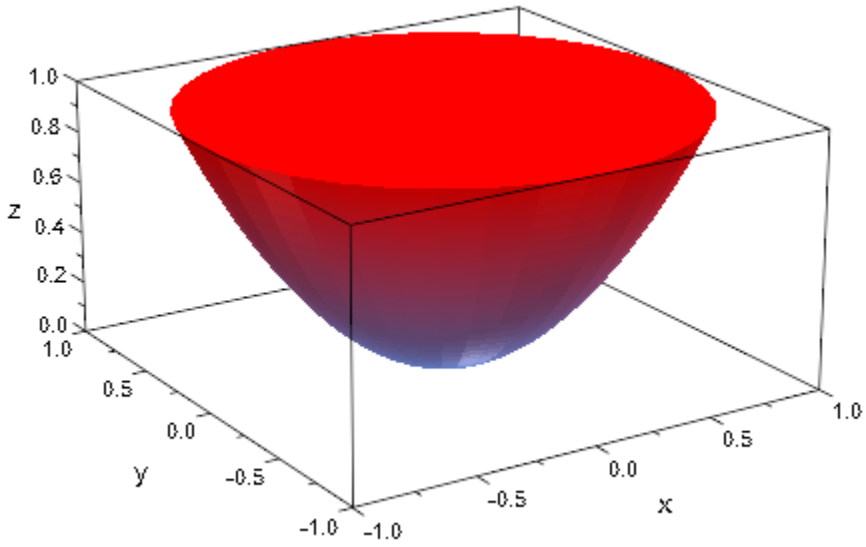



```
delete x, y, z, filename, DIGITS:
```

Example 3

We wish to create a closed surface consisting of a “bowl” with a “lid”.

```
bowl:= [u*cos(v), u*sin(v), u^2], u = 0 .. 1, v = 0 .. 2*PI:
lid:=  [u*cos(v), u*sin(v),  1 ], u = 0 .. 1, v = 0 .. 2*PI:

filename:= NOTEBOOKPATH."sample.stl":
DIGITS:= 7:
export::stl(filename, bowl, Mesh = [30, 36]):
export::stl(filename, lid, Mesh = [30, 36], Append):
plot(plot::SurfaceSTL(filename), Scaling = Constrained):
```



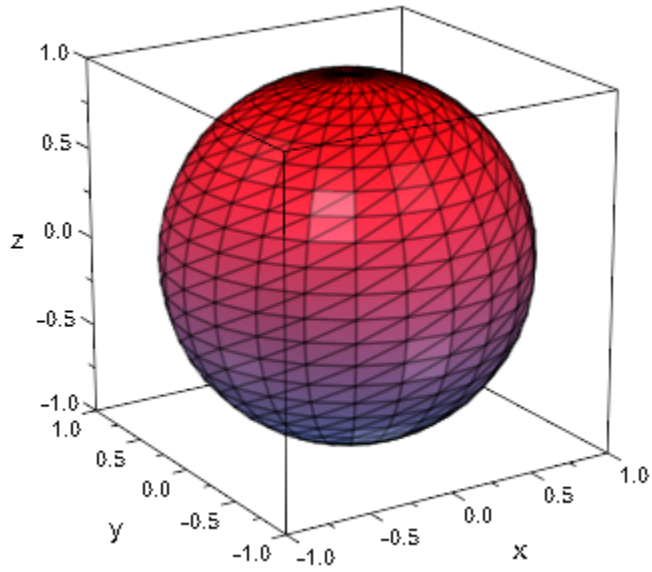

```
delete filename, DIGITS:
```

Example 4

We demonstrate the options `Scaling` = `Constrained` and `Scaling` = `Unconstrained`. With `Scaling` = `Constrained`, the coordinates given by the parametrization `x, y, z` are scaled by the same factor to fit the surface into the output box. Here, we create a sphere of radius 1. The output box is not a cube: the range for the `z` coordinate is notably larger than for `x` and `y`. Nevertheless, the sphere stays a sphere when using `Scaling` = `Constrained`. However, the output box is not completely filled by the sphere:

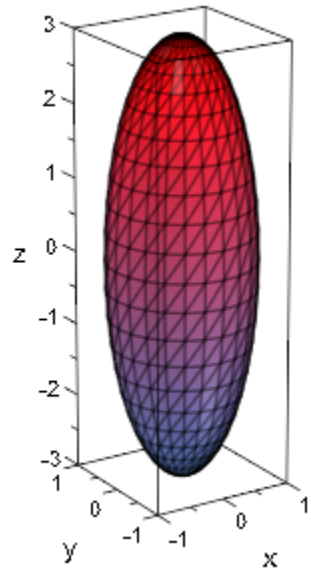
```
x:= cos(u)*sin(v):
y:= sin(u)*sin(v):
z:= cos(v):
DIGITS:= 7:
filename:= NOTEBOOKPATH."sphere.stl":
export::stl(filename, [x, y, z], u = 0 .. 2*PI, v = 0 .. PI,
            OutputBox = [-1 .. 1, -1 .. 1, -3 .. 3],
            Scaling = Constrained):
```

```
plot(plot::SurfaceSTL(filename,
                      Scaling = Constrained,
                      MeshVisible))
```



With `Scaling` = `Unconstrained`, the sphere is deformed to an ellipsoid filling the output box:

```
export::stl(filename, [x, y, z], u = 0..2*PI, v = 0..PI,
            OutputBox = [-1..1, -1..1, -3..3],
            Scaling = Unconstrained):

plot(plot::SurfaceSTL(filename,
                      Scaling = Constrained,
                      MeshVisible))
```




```
delete x, y, z, filename, DIGITS:
```

Parameters

`filename`

A file name: a non-empty character string

`n`

A file descriptor provided by `fopen`: a positive integer

$\texttt{object}_1$, $\texttt{object}_2$, …

3D graphical objects of the `plot` library

`x, y, z`

The coordinate functions: arithmetical expressions or `piecewise` objects depending on the surface parameters u and v. Alternatively, procedures that accept 2 input parameters u, v and return a numerical value when the input parameters are numerical.

u

The first surface parameter: an identifier or an indexed identifier.

u_{min} .. u_{max}

The range for the parameter u: u_{min}, u_{max} must be numerical real values.

v

The second surface parameter: an identifier or an indexed identifier.

v_{min} .. v_{max}

The range for the parameter v: v_{min}, v_{max} must be numerical real values.

Options

Mesh

Option, specified as `Mesh = [nu, nv]`

Sets the mesh size: the integer n_u determines, how many equidistant points in the `u` direction are used to sample the parametrization `x, y, z` numerically. Correspondingly, the integer n_v determines, how many equidistant points in the `v` direction are used. Thus, a regular mesh of (n_u - 1) (n_v - 1) rectangles is used. Each rectangle is split into 2 triangles, resulting in a triangulation consisting of 2 (n_u - 1) (n_v - 1) triangles. The default is `Mesh = [25, 25]`.

OutputBox

Option, specified as `OutputBox = [xmin .. xmax, ymin .. ymax, zmin .. zmax]`

By default, the coordinates of the mesh points defining the STL object are written into the file as provided by the parametrization of the surface. Thus, if several objects are written into the file via the option `Append`, the position of the objects in space is transparent and can be controlled by the user via a suitable parametrization. However, many devices such as Rapid Prototyping tools with which the STL file shall be processed, impose severe restrictions on the data in the STL file. E.g., the original STL specification requires that the x, y, z coordinates of the mesh points are positive. Many devices require

that the coordinates must lie in a prescribed range (between 0 and 100, say). The option `OutputBox` provides a simple mean to shift and scale the coordinates given by the parametrization to a prescribed range.

The option `OutputBox = [ xmin.. xmax, ymin.. ymax, zmin.. zmax]` sets the output box defined by numerical values `xmin, …, zmax`. The mathematical coordinates `x(u, v), y(u, v), z(u, v)` with `u`, `v` ranging from `umin` to `umax` and from `vmin` to `vmax`, respectively, are shifted and scaled such that the output coordinates written to the STL file range between the values `xmin` and `xmax`, `ymin` and `ymax`, `zmin` and `zmax`.

Note: If several objects are written to the file via the option `Append`, only the very last call of `export::stl` should bear the option `OutputBox`!

This last call shifts and scales all coordinates of all surfaces inside the file such that the entire scene of objects fits into the output box. The relative size and positions of the objects are preserved.

See “Example 3” on page 8-10.

This option is rather expensive since all data in the STL file need to be modified!

This option is not available if the file was opened outside `export::stl` and passed by a file descriptor `n`.

`Scaling`

Option, specified as `Scaling = Unconstrained` or `Scaling = Constrained`

With `Scaling = Unconstrained`, the surface is scaled by different factors in the x, y, and z direction, such that it fills the output box set by the option `OutputBox - [`x_{min}` .. `x_{max}`, `y_{min}` .. `y_{max}`, `z_{min}` .. `z_{max}`]` . Thus, the output coordinates of a sphere define an ellipsoid with diameters given by the side lengths of the output box. This is the default setting.

With `Scaling = Constrained`, the surface is scaled by the same factor in the x, y, and z direction such that it fits into the output box set by the option `OutputBox = [`x_{min}` .. `x_{max}`, `y_{min}` .. `y_{max}`, `z_{min}` .. `z_{max}`]` . A sphere will remain a sphere even if the sides of the output box have different lengths.

This option is ignored if not used in conjunction with the `OutputBox` option.

`Ascii`, `Bin`, `Binary`, `Raw`, `Text`

With the synonymous flags `Bin`, `Binary`, or `Raw`, respectively, the STL file is created as a binary file. If a binary file is specified by a file descriptor `n`, make sure that is was opened by the command `n:= fopen(filename, Write, Raw)`. With the synonymous flags `Text` and `Ascii`, repectively, the STL file is created as a text file. The default is `Bin`.

`Append`

With this flag, the STL data of the surface are appended to an existing STL file named "filename". If no such file exists, it is created and processed as without `Append`. This option is not available if the file was opened outside `export::stl` and passed by a file descriptor `n`.

Return Values

`null()` object.

Algorithms

There are two storage formats available for STL files, which are ASCII and BINARY. ASCII files are human-readable while BINARY files are smaller and faster to process. Both ASCII as well as BINARY files can be generated by `export::stl`. A typical ASCII STL file looks like this:

```
solid sample
 facet normal -4.470293E-02 7.003503E-01 -7.123981E-01
  outer loop
   vertex -2.812284E+00 2.298693E+01 0.000000E+00
   vertex -2.812284E+00 2.296699E+01 -1.960784E-02
   vertex -3.124760E+00 2.296699E+01 0.000000E+00
  endloop
 endfacet
 ...
endsolid sample
```

STL BINARY files have the following format:

```
Bytes  Type   Description
80     ASCII  header, no data significance
4      uint   number of facets in file

4      float  normal  x - start of facet
4      float  normal  y
4      float  normal  z
4      float  vertex1 x
4      float  vertex1 y
4      float  vertex1 z
4      float  vertex2 x
4      float  vertex2 y
4      float  vertex2 z
4      float  vertex3 x
4      float  vertex3 y
4      float  vertex3 z
2      byte   not used  - end of facet
       ...
```

Facet orientation: The facets define the surface of a 3D object. As such, each facet is part of the boundary between the interior and the exterior of the object. The orientation of the facets (which way is "out" and which way is "in") is specified redundantly in two ways which should be consistent. First, the direction of the normal is outward. Second, which is most commonly used nowadays, the facet vertices are listed in counter-clockwise order when looking at the object from the outside (right-hand rule).

Vertex-to-vertex rule: Each triangle must share two vertices with each of its adjacent triangles. In other words, a vertex of one triangle cannot lie on the side of another.

Axes: The format specifies that all vertex coordinates must be strictly positive numbers. However, it seems that – with a few exceptions – most software used today (MuPAD included) allow negative coordinates as well.

Units: The STL file does not contain any scale information; the coordinates may be interpreted in arbitrary units.

Further details about the STL file format are available in the web, e.g., at:

- www.ennex.com/fabbers/StL.asp,
- www.math.iastate.edu/burkardt/data/stl/stl.html and
- rpdrc.ic.polyu.edu.hk/content/stl/stl_introduction.htm.

Collections of STL sample files can be found in the web, e.g., at:

- www.wohlersassociates.com/Software-for-Rapid-Prototyping.html and
- www.cs.duke.edu/~edels/Tubes.

Information about rapid prototyping technologies is available in the web, e.g., at:

www.cs.hut.fi/~ado/rp/rp.html.

See Also

MuPAD Functions
`fclose` | `fopen` | `READPATH` | `WRITEPATH`

MuPAD Graphical Primitives
`plot::SurfaceSTL`

fp – Functional Programming

fp::apply
fp::bottom
fp::curry
fp::expr_unapply
fp::fixargs
fp::fixedpt
fp::fold
fp::nest
fp::nestvals
fp::unapply

fp::apply

Apply function to arguments

Syntax

```
fp::apply(f, <e, …>)
```

Description

`fp::apply(f,a)` returns `f(a)`.

`fp::apply` applies the function `f` to the arguments given by `e`, ...

Environment Interactions

Same side effects as when calling `f(e,...)` directly.

Examples

Example 1

Apply the function `f` to `x` and `y`:

```
fp::apply(f, x, y)
```

$$f(x, y)$$

Example 2

Apply the functions of the first list to the arguments given by the second list:

```
zip([sin, cos], [x, y], fp::apply)
```

$[\sin(x), \cos(y)]$

Parameters

`f`

Function

`e`

Object used as argument

Return Values

Result of the function call `f(e,...)`.

fp::bottom

Function that never returns

Syntax

```
fp::bottom()
```

Description

`fp::bottom()` never returns because it raises an error.

Environment Interactions

Raises an error in any case.

Examples

Example 1

Calling `fp::bottom` is equivalent to calling `error` with a fixed error string:

```
fp::bottom()
```

```
Error: The bottom is reached. [fp::bottom]
```

`fp::bottom` is used to indicate the bottom of a recursion inside a `traperror` call. In most cases, programs not using `fp::bottom` will be more readable.

Return Values

This function never returns.

fp::curry

Curry an n-ary function

Syntax

```
fp::curry(f, <n>)
```

Description

`fp::curry(f)` returns the higher-order function $x \to (y \to f(x, y))$.

`fp::curry` returns the curried version of the `n`-ary function `f`. If no arity `n` is given, then the function is assumend to be binary.

If `n` is smaller than 2 then `f` is returned. Otherwise, given a n-ary function f, `fp::curry` returns the function $x_1 \to ((x_2, \ldots, x_n) \to f(x_1, \ldots, x_n))$

Examples

Example 1

Create curried versions of binary and 3-nary functions:

```
cf := fp::curry(f):
cf(x)(y)
```

$$f(x, y)$$

```
cg := fp::curry(g, 3):
cg(x)(y)(z)
```

$$g(x, y, z)$$

Example 2

A curried version of `_plus` may be used to create a function which increments its argument by `1`:

```
inc := fp::curry(_plus)(1):
inc(x)
```

$$x + 1$$

Parameters

`f`

n-ary function

`n`

Nonnegative integer

Return Values

Unary higher-order function.

fp::expr_unapply

Create a functional expression from an expression

Syntax

```
fp::expr_unapply(e, <x, …>)
```

Description

`fp::expr_unapply(e,x)` tries to interpret the expression `e` as a function in `x` and to return a functional expression computing that function.

`fp::expr_unapply` views the expression `e` as a function in the indeterminates `x,...` and tries to return a functional expression computing that function. If `fp::expr_unapply` cannot find a functional expression `FAIL` is returned.

If no indeterminates are given, any indeterminates of `e` found by `indets` are used.

Examples

Example 1

Get the functional expression computing `sin(x)`:

```
fp::expr_unapply(sin(x), x)
```

```
     sin
```

```
fp::expr_unapply(sin(x[1]), x[1])
```

```
     sin
```

Example 2

Get the functional expression computing `sin(x)^2+cos(x)^2`:

```
fp::expr_unapply(sin(x)^2 + cos(x)^2)
```

$$\sin^2 + \cos^2$$

Parameters

e

Expression

x

Identifier or indexed identifier

Return Values

Functional expression or `FAIL`.

See Also

MuPAD Functions
`fp::unapply`

fp::fixargs

Create function by fixing all but one argument

Syntax

```
fp::fixargs(f, n, <e, …>)
```

Description

`fp::fixargs(f,1,y)` returns the function $x \to f(x, y)$.

`fp::fixargs` returns an unary function, defined by fixing all but the `n`-th argument of the function `f` to the values given by `e...`.

Thus, given a m-ary function f and m - 1 values e_1, …, e_{m-1}, `fp::fixargs` returns the function

$$x \to f\left(e_1, \ldots, e_{n-1}, x, e_n, \ldots, e_{m-1}\right)$$

Examples

Example 1

Fix the first and third argument of `f` to `x1` and `x3`:

```
fp::fixargs(f, 2, x1, x3)(y)
```

$$f(\mathrm{x1}, y, \mathrm{x3})$$

Example 2

Create a function which increments its argument by one:

```
inc := fp::fixargs(_plus, 1, 1):
inc(x)
```

$$x+1$$

Example 3

Create a function which tests the identifier `x` for a type:

```
type_of_x := fp::fixargs(testtype, 2, x):
map([DOM_INT, DOM_IDENT], type_of_x)
```

$$[\text{FALSE}, \text{TRUE}]$$

Parameters

`f`

Function

`n`

Positive integer defining free argument

`e`

Object used as fixed argument

Return Values

Unary function.

fp::fixedpt

Returns fixed point of a function

Syntax

```
fp::fixedpt(f)
```

Description

`fp::fixedpt(f)` returns the fixed point of the unary function `f`.

`fp::fixedpt` is implemented as the *Y* combinator which is defined as follows:

$$Y: f \to g(f)(g(f))$$

where the function *g* is defined as

$$g: f \to h \to x \to f(h(h))(x)$$

Examples

Example 1

A function computing the Fibonacci numbers is created as a fixed point:

```
fb2 := (f,n) -> if n <= 2 then 1 else f(n-1) + f(n-2) end:
fib := fp::fixedpt(fp::curry(fb2)):
fib(i) $ i=1..9
```

```
1, 1, 2, 3, 5, 8, 13, 21, 34
```

Parameters

f

Unary function

Return Values

Unary function.

fp::fold

Create function which iterates over sequences

Syntax

```
fp::fold(f, <e, …>)
```

Description

`fp::fold` returns a function which repeatedly applies `f` to sequences of arguments, where the expressions `e...` are used as starting values.

Thus, given the function f and the starting values e_1, …, e_n, `fp::fold` returns the function which is defined by

$$(x_1, x_2, \ldots, x_m) \to f(x_m, \ldots f(x_2, f(x_1, e_1, \ldots, e_n))\ldots)$$

for any positive integer m. If the argument sequence is void (i.e. $m = 0$) the function simply returns the sequence (e_1, …, e_n).

Examples

Example 1

A call to `fp::fold` returns a function, which accepts an arbitrary number of arguments:

```
fp::fold(f, x)(y1, y2, y3)
```

$$f(\text{y3}, f(\text{y2}, f(\text{y1}, x)))$$

Example 2

The function `pset` returns the power set of the set given by its arguments:

```
addelem := (x,y) -> y union map(y, _union, {x}):
pset := fp::fold(addelem, {{}}):
pset(a,b,c)
```

$$\{\emptyset, \{c\}, \{a\}, \{b\}, \{b, c\}, \{a, b\}, \{a, c\}, \{a, b, c\}\}$$

Parameters

f

Function

e

Object used as starting value

Return Values

Function.

fp::nest

Repeated composition of function

Syntax

```
fp::nest(f, n)
```

Description

`fp::nest(f,n)` returns the `n`-fold repeated composition of the function `f`.

Thus, given the function `f`, `fp::nest` returns the identity function `id` if `n` is 0 and otherwise the function

$$f(f(\ldots f(x)\ldots))$$

`n`-fold repeated.

Note that `fp::nest` is obsolete, one should use the `@@` operator or its functional form `_fnest` instead. It is only supported for compatibility with former versions of MuPAD.

Examples

Example 1

Apply the 3-fold repeated composition of `f` to `x`:

```
fp::nest(f, 3)(x)
```

$$f(f(f(x)))$$

Example 2

Numerically finding a fixed point of the function `cos` by repeated application:

```
p :=fp::nest(cos, 100)(1.0):
p, cos(p)
```

```
0.7390851332, 0.7390851332
```

Parameters

f

Function

n

Nonnegative integer

Return Values

Function.

See Also

MuPAD Functions

`_fconcat` | `_fnest` | `fp::nestvals`

fp::nestvals

Repeated composition returning intermediate values

Syntax

```
fp::nestvals(f, n)
```

Description

`fp::nestvals(f, n)` returns a function which applies the function `fn`-fold repeatedly to its argument and returns the intermediate $n + 1$ values as a list.

Thus `fp::nestvals` returns the function $[x, f(x), \ldots, f(f(\ldots f(x)\ldots))]$

The function returned is equivalent to `[_fnest(f,i) $i=0..n]`, but more efficient.

Examples

Example 1

Apply `f` 3 times nested to `x`:

```
fp::nestvals(f, 3)(x)
```

$$[x, f(x), f(f(x)), f(f(f(x)))]$$

Example 2

Apply `cos` 4 times nested to `1.0` and return the result and intermediate values:

```
fp::nestvals(cos, 4)(1.0)
```

$$[1.0, 0.5403023059, 0.8575532158, 0.6542897905, 0.7934803587]$$

Parameters

f

Function

n

Nonnegative integer

Return Values

Function.

See Also

MuPAD Functions
`_fconcat` | `_fnest`

fp::unapply

Create a procedure from an expression

Syntax

```
fp::unapply(e, <x, …>)
```

Description

`fp::unapply` views the expression `e` as a function in the indeterminates `x,...` and returns a procedure computing that function.

If no indeterminates are given, any indeterminates of `e` found by `indets` are used.

Examples

Example 1

Get the procedure computing `sin(x)^2+cos(y)^2`:

```
s := fp::unapply(sin(x)^2 + cos(y)^2, x, y)
```

$$(x, y) \to \cos(y)^2 + \sin(x)^2$$

Parameters

e

Expression

x

Identifier or indexed identifier

Return Values

Procedure.

Overloaded By

e

See Also

MuPAD Functions
`fp::expr_unapply`

generate – Generate Input to Other Programs

generate::C
generate::fortran
generate::MATLAB
generate::MathML
generate::optimize
generate::Simscape
generate::TeX

generate::C

Generate C formatted string

Syntax

```
generate::C(e, <NoWarning>)
```

Description

`generate::C(e)` generates C code for the MuPAD expression `e`.

`generate::C` returns a C formatted string representing an expression, equation, list of equations or a matrix.

An equation represents an assignment in C code. The type of the assignment is `double`.

When generating C code for a matrix, the generator assigns only nonzero elements. See “Example 3” on page 10-3.

To print an output string to a file, use the `fprint` function. Use the printing option `Unquoted` to remove quotation marks and to expand special characters like line breaks and tabs.

Use the `generate::optimize` function to optimize the MuPAD code before converting it to C code. See “Example 5” on page 10-4.

The `NoWarning` option lets you suppress warnings. See “Example 6” on page 10-4.

Examples

Example 1

The code generator converts a list of equations to a sequence of assignments:

```
generate::C([x1 = y2^2*(y1 + sin(z)), x2 = tan(x1^4)]):
```

```
print(Unquoted, %)
```

```
  x1 = (y2*y2)*(y1+sin(z));
  x2 = tan(x1*x1*x1*x1);
```

Example 2

MuPAD matrix and array indexing differs from C array indexing. By default, MuPAD array indices start with 1, and C array indices start with 0. To create the code compatible with the default indexing in C, the `generate::C` function decrements each index by one:

```
A:= matrix([[1,2],[3,4]]):
generate::C(A)."\n".
  generate::C(hold(Determinante = A[1,1]*A[2,2] - A[1,2]*A[2,1])):
print(Unquoted, %)
```

```
  A[0][0] = 1.0;
  A[0][1] = 2.0;
  A[1][0] = 3.0;
  A[1][1] = 4.0;

  Determinante = A[0][0]*A[1][1]-A[0][1]*A[1][0];
```

Example 3

Generated C code does not include assignments for zero elements of a matrix:

```
A:= matrix([[1, 0, 0],[0, 0, 1]]):
print(Unquoted, generate::C(A))
```

```
  A[0][0] = 1.0;
  A[1][2] = 1.0;
```

Example 4

If the first index of an array is not 1, the `generate::C` function issues a warning:

```
A:= array(1..2, 2..3, [[1,2],[3,4]]):
print(Unquoted, generate::C(A))
```

```
Warning: The array index 'A[1..2, 2..3]' is out of range 1..n. [DOM_ARRAY::CF]


  A[0][1] = 1.0;
  A[0][2] = 2.0;
  A[1][1] = 3.0;
  A[1][2] = 4.0;
```

Example 5

The `generate::C` function does not optimize your code:

```
print(Unquoted,
      generate::C([x = a + b, y = (a + b)^2])):


  x = a+b;
  y = pow(a+b,2.0);
```

You can use the `generate::optimize` function before converting your MuPAD expression to C code. For example, this function can reduce the number of operations by finding common subexpressions:

```
print(Unquoted,
      generate::C(
          generate::optimize([x = a + b, y = (a + b)^2])
      )):


  x = a+b;
  y = x*x;
```

Example 6

By default, the `generate::C` function can issue warnings:

```
print(Unquoted, generate::C(f(x)))


Warning: Function 'f' is not verified to be a valid C function.


  t0 = f(x);
```

If you started using `generate::C` recently, the warnings can help you identify the potential issues in the converted code. If you want to suppress warnings, use the `NoWarning` option:

```
print(Unquoted, generate::C(f(x), NoWarning))
```

```
  t0 = f(x);
```

Parameters

e

An expression, equation, list of equations, or a matrix

Options

`NoWarning`

Suppress warnings.

Return Values

`generate::C` returns a string containing C code.

See Also

MuPAD Functions

`fprint` | `generate::optimize` | `print`

generate::fortran

Generate Fortran formatted string

Syntax

```
generate::fortran(e, <NoWarning>, <Version = "versionName">)
```

Description

`generate::fortran(e)` generates Fortran code for the MuPAD expression `e`.

`generate::fortran` returns a Fortran formatted string representing an expression, equation, list of equations, or a matrix.

An equation represents an assignment in Fortran code. The type of the assignment is `double`.

When generating Fortran code for a matrix, the generator assigns only nonzero elements. See “Example 2” on page 10-7.

To print an output string to a file, use the `fprint` function. To remove quotation marks and to expand special characters like line breaks and tabs, use the printing option `Unquoted`.

Use the `generate::optimize` function to optimize the MuPAD code before converting it to Fortran code. See “Example 4” on page 10-7.

The `NoWarning` option lets you suppress warnings. See “Example 5” on page 10-8.

The `Version` option specifies the target version of the Fortran compiler that `generate::fortran` uses to generate code. The options are `Fortran77` (default), `Fortran90`, and `Fortran95`. See “Example 6” on page 10-9.

Examples

Example 1

The code generator converts a list of equations to a sequence of assignments.

```
generate::fortran([x[1] = y[2 + i]^2*(y[1] + sin(z)),
                   x[2] = tan(x[1]^4)]):
print(Unquoted,%)
```

```
      x(1) = (sin(z)+y(1))*y(i+2)**2
      x(2) = tan(x(1)**4)
```

Example 2

Generated Fortran code does not include assignments for zero elements of a matrix.

```
A:= matrix([[1, 0, 0],[0, 0, 1]]):
print(Unquoted, generate::fortran(A))
```

```
      A(1,1) = 1.0D0
      A(2,3) = 1.0D0
```

Example 3

If the first index of an array is not 1, then the `generate::fortran` function issues a warning.

```
A:= array(1..2, 2..3, [[1,2],[3,4]]):
print(Unquoted, generate::fortran(A))
```

```
Warning: The array index 'A[1..2, 2..3]' is out of range 1..n. [DOM_ARRAY::CF]
```

```
      A(1,2) = 1.0D0
      A(1,3) = 2.0D0
      A(2,2) = 3.0D0
      A(2,3) = 4.0D0
```

Example 4

The `generate::fortran` function does not optimize your code.

```
print(Unquoted,
      generate::fortran([x = a + b, y = (a + b)^2])):
```

```
      x = a+b
      y = (a+b)**2
```

You can use the `generate::optimize` function before converting your MuPAD expression to Fortran code. For example, this function can reduce the number of operations by finding common subexpressions.

```
print(Unquoted,
      generate::fortran(
          generate::optimize([x = a + b, y = (a + b)^2])
      )):


      x = a+b
      y = x**2
```

Example 5

By default, the `generate::fortran` function can issue warnings.

```
print(Unquoted, generate::fortran(gamma(x)))


Warning: Function 'gamma' requires a Fortran2008 compiler.


      t0 = gamma(x)
```

Warnings help identify potential issues in converted code. To suppress warnings, use the `NoWarning` option.

```
print(Unquoted, generate::fortran(gamma(x), NoWarning))


      t0 = gamma(x)
```

If the warning specifies that the compiler required is either Fortran90 or Fortran95, then you can suppress the warning by specifying the correct compiler version using `Version`. For example, the `ceiling` function requires Fortran90 instead of the default Fortran77.

```
generate::fortran(ceil(x))
```

```
Warning: Function 'ceiling' requires a Fortran90 compiler.
```

```
"    t0 = ceiling(x) "
```

Specify `Version` as `Fortran90`. The `generate::fortran` function does not issue a warning.

```
generate::fortran(ceil(x), Version = "Fortran90")
```

```
"    t0 = ceiling(x) "
```

Example 6

By default, the `generate::fortran` function uses the target Fortran version `Fortran77` to generate code. To specify `Fortran90` or `Fortran95` as the target version, use the `Version` option.

Generate output for the Fortran90 compiler by specifying the `Version` option as `Fortran90`.

```
f := expand((x+1)^20):
fcode90 := generate::fortran(f, Version = "Fortran90"):
print(Unquoted, fcode90)
```

```
      t0 = x*2.0D1+x**2*1.9D2+x**3*1.14D3+x**4*4.845D3+x**5*1.5504D4+x**&
     &6*3.876D4+x**7*7.752D4+x**8*1.2597D5+x**9*1.6796D5+x**10*1.84756D5&
     &+x**11*1.6796D5+x**12*1.2597D5+x**13*7.752D4+x**14*3.876D4+x**15*1&
     &.5504D4+x**16*4.845D3+x**17*1.14D3+x**18*1.9D2+x**19*2.0D1+x**20+1&
     &.0D0
```

The code formatting for multiline statements in `Fortran90` differs from the formatting in the default target of `Fortran77`.

Parameters

e

An expression, equation, list of equations, or a matrix

Options

`NoWarning`

Suppress warnings.

`Version`

Specify the Fortran compiler version. The default version is `Fortran77`. The `Version` values are `Fortran77`, `Fortran90`, and `Fortran95`. For example, `generate::fortran(..., Version = "Fortran90")` uses compiler version `Fortran90`.

Return Values

`generate::fortran` returns a string containing Fortran code.

See Also

MuPAD Functions
`fprint` | `generate::optimize` | `print`

generate::MATLAB

Generate MATLAB formatted string

Syntax

```
generate::MATLAB(e, <NoWarning>)
```

Description

`generate::MATLAB(e)` generates MATLAB code for the MuPAD expression `e`.

`generate::MATLAB` returns a MATLAB formatted string representing an expression, equation, list of equations or a matrix.

`generate::MATLAB` assumes that the type of converted data is `double`. See “Example 1” on page 10-12.

An equation represents an assignment in MATLAB code. See “Example 4” on page 10-13.

When generating MATLAB code for a matrix, the generator produces a matrix of zeros, and then it substitutes nonzero elements. See “Example 2” on page 10-12.

Use the `generate::optimize` function to optimize the MuPAD code before converting it to the MATLAB syntax. See “Example 5” on page 10-13.

To display generated MATLAB code on screen, use the `print` function. Use the printing option `Unquoted` to remove quotation marks and to expand special characters like line breaks and tabs. If a generated code line is longer than the `TEXTWIDTH` setting, the `print` function breaks that line into several shorter lines. The inserted line continuation character (\) is not valid in MATLAB. To avoid inserting line continuation characters, increase the `TEXTWIDTH` setting or use the `fprint` function to write generated code to a file.

`generate::MATLAB` does not create a MATLAB function. You can print an output string to a file using the `fprint` function with the `Unquoted` option. See “Example 6” on page 10-14.

Working from the MATLAB workspace you can create a MATLAB function containing your expression. To call the MuPAD expression from the MATLAB workspace, use `evalin` or `feval` functions. See “Create MATLAB Functions from MuPAD Expressions”.

If you work with the Simulink® products, you can copy the generated code and paste it into a Simulink block. Also, you can call the MuPAD expression from the MATLAB workspace using `evalin` or `feval` functions. Working from the MATLAB workspace you can automatically create a Simulink block containing your expression. See “Create MATLAB Function Blocks from MuPAD Expressions”.

The `NoWarning` option lets you suppress warnings. See “Example 7” on page 10-14.

Examples

Example 1

By default, MATLAB stores all numeric values as double-precision floating-point. In accordance with the default MATLAB data type, `generate::MATLAB` converts the elements of expressions, equations, and matrices to the `double` format:

```
print(Unquoted, generate::MATLAB(x^2 + y/3 + 1/6))
```

```
  t0 = y*(1.0/3.0)+x^2+1.0/6.0;
```

Example 2

The generator produces a matrix of zeros, and then it replaces nonzero elements:

```
A:= matrix([[1, 0, 0],[0, 0, 1]]):
print(Unquoted, generate::MATLAB(A))
```

```
  A = zeros(2,3);
  A(1,1) = 1.0;
  A(2,3) = 1.0;
```

Example 3

If the first index of an array is not 1, the `generate::MATLAB` function issues a warning:

```
A:= array(1..2, 2..3, [[1,2],[3,4]]):
print(Unquoted, generate::MATLAB(A))
```

```
Warning: The array index 'A[1..2, 2..3]' is out of range 1..n. [DOM_ARRAY::CF]
```

```
  A = zeros(2,2);
  A(1,2) = 1.0;
  A(1,3) = 2.0;
  A(2,2) = 3.0;
  A(2,3) = 4.0;
```

Example 4

When generating MATLAB code from equations, you get assignments instead of equations. For example, generate MATLAB code for the following list of equations:

```
f := generate::MATLAB([x = exp(t*s), y = sin(t)*cos(s)]):
print(Unquoted, f)
```

```
  x = exp(s*t);
  y = cos(s)*sin(t);
```

Example 5

The `generate::MATLAB` function does not optimize your code:

```
print(Unquoted,
      generate::MATLAB([x = a + b, y = (a + b)^2])):
```

```
  x = a+b;
  y = (a+b)^2;
```

You can use the `generate::optimize` function before converting your MuPAD code to MATLAB syntax. For example, this function can reduce the number of operations by finding common subexpressions:

```
f := generate::optimize([x = a + b, y = (a + b)^2]):
print(Unquoted,
      generate::MATLAB(f)):
```

```
  x = a+b;
  y = x^2;
```

Example 6

To create a file with a MATLAB formatted string representing a symbolic expression, use the `fprint` function:

```
A:= matrix([[1, 0, 0],[0, 0, 1]]):
fprint(Unquoted, Text, "matrixA.m", generate::MATLAB(A))
```

If the file matrixA.m already exists, `fprint` replaces the existing MATLAB code with the converted symbolic expression. You can open and edit the resulting file.

Example 7

By default, the `generate::MATLAB` function can issue warnings:

```
print(Unquoted, generate::MATLAB(g(x)))
```

```
Warning: Function 'g' is not verified to be a valid MATLAB function.
```

```
  t0 = g(x);
```

If you started using `generate::MATLAB` recently, the warnings can help you identify the potential issues in the converted code. If you want to suppress warnings, use the `NoWarning` option:

```
print(Unquoted, generate::MATLAB(g(x), NoWarning))
```

```
  t0 = g(x);
```

Parameters

e

An expression, equation, list of equations, or a matrix

Options

`NoWarning`

Suppress warnings.

Return Values

`generate::MATLAB` returns a string containing MATLAB code.

See Also

MuPAD Functions

`fprint` | `generate::optimize` | `generate::Simscape` | `print`

generate::MathML

Generate MathML from expressions

Syntax

```
generate::MathML(expr, options)
```

Description

`generate::MathML(expr)` returns the MathML code representing `expr`. To print this code to a file, use `fprint`.

Examples

Example 1

Generate the MathML code from the following expression. Use `hold` to prevent evaluation of the integral. By default, `generate::MathML` returns both Presentation and Content MathML, and includes annotations.

```
generate::MathML(hold(int)(exp(x^2)/x, x))
```

```
<math xmlns='http://www.w3.org/1998/Math/MathML'>
  <semantics>
    <mrow>
      <mo form='prefix'>&Integral;</mo>
      <mrow/>
      <mfrac>
        <msup>
          <mo>&ee;</mo>
          <msup>
            <mi>x</mi>
            <mn>2</mn>
          </msup>
        </msup>
```

```
        <mi>x</mi>
      </mfrac>
      <mo form='infix'>&DifferentialD;</mo>
      <mi>x</mi>
    </mrow>
    <annotation-xml encoding='MathML-Content'>
      <apply>
        <int/>
        <bvar>
          <ci>x</ci>
        </bvar>
        <apply>
          <divide/>
          <apply>
            <exp/>
            <apply>
              <power/>
              <ci>x</ci>
              <cn type='integer'>2</cn>
            </apply>
          </apply>
          <ci>x</ci>
        </apply>
      </apply>
    </annotation-xml>
    <annotation encoding='MuPAD'>
      int(exp(x^2)/x, x)
    </annotation>
  </semantics>
</math>
```

Example 2

Show only the Presentation MathML part of the output by setting `Content` to `FALSE`.

```
generate::MathML(hold(int)(exp(x^2)/x, x),
                         Content = FALSE)
```

```
<math xmlns='http://www.w3.org/1998/Math/MathML'>
  <semantics>
    <mrow>
      <mo form='prefix'>&Integral;</mo>
      <mrow/>
```

```
      <mfrac>
        <msup>
          <mo>&ee;</mo>
          <msup>
            <mi>x</mi>
            <mn>2</mn>
          </msup>
        </msup>
        <mi>x</mi>
      </mfrac>
      <mo form='infix'>&DifferentialD;</mo>
      <mi>x</mi>
    </mrow>
    <annotation encoding='MuPAD'>
      int(exp(x^2)/x, x)
    </annotation>
  </semantics>
</math>
```

Show only the Content MathML part of the output by setting `Presentation` to `FALSE`.

```
generate::MathML(hold(int)(exp(x^2)/x, x),
                   Presentation = FALSE)
```

```
<math xmlns='http://www.w3.org/1998/Math/MathML'>
  <semantics>
    <apply>
      <int/>
      <bvar>
        <ci>x</ci>
      </bvar>
      <apply>
        <divide/>
        <apply>
          <exp/>
          <apply>
            <power/>
            <ci>x</ci>
            <cn type='integer'>2</cn>
          </apply>
        </apply>
        <ci>x</ci>
      </apply>
    </apply>
```

```
    <annotation encoding='MuPAD'>
      int(exp(x^2)/x, x)
    </annotation>
  </semantics>
</math>
```

Suppress annotations by setting `Annotation` to `FALSE`.

```
generate::MathML(hold(int)(exp(x^2)/x, x),
                      Annotation = FALSE)


<math xmlns='http://www.w3.org/1998/Math/MathML'>
  <semantics>
    <mrow>
      <mo form='prefix'>&Integral;</mo>
      <mrow/>
      <mfrac>
        <msup>
          <mo>&ee;</mo>
          <msup>
            <mi>x</mi>
            <mn>2</mn>
          </msup>
        </msup>
        <mi>x</mi>
      </mfrac>
      <mo form='infix'>&DifferentialD;</mo>
      <mi>x</mi>
    </mrow>
    <annotation-xml encoding='MathML-Content'>
      <apply>
        <int/>
        <bvar>
          <ci>x</ci>
        </bvar>
        <apply>
          <divide/>
          <apply>
            <exp/>
            <apply>
              <power/>
              <ci>x</ci>
              <cn type='integer'>2</cn>
            </apply>
```

```
          </apply>
          <ci>x</ci>
        </apply>
      </apply>
    </annotation-xml>
  </semantics>
</math>
```

Example 3

Generate MathML code from the following expression and write the result to `filename.mathml` by using `fprint`.

```
fprint(Text, "filename.mathml",
       generate::MathML(hold(int)(exp(x^2)/x, x))):
```

Parameters

`expr`

Arithmetical expression

Options

`Annotation`

Option, specified as `Annotation = FALSE`.

Suppresses the output of annotations.

`Content`

Option, specified as `Content = FALSE`.

Suppresses the MathML Content part of the output.

`Presentation`

Option, specified as `Presentation = FALSE`.

Suppresses the MathML Presentation part of the output.

Return Values

`generate::MathML` returns an object containing MathML code.

Overloaded By

expr

See Also

MuPAD Functions
`fprint` | `print`

generate::optimize

Generate optimized code

Syntax

```
generate::optimize(r)
```

Description

`generate::optimize(r)` returns a sequence of equations representing an "optimized computation sequence" for the input expression r. Each equation in the sequence corresponds to an assignment of a subexpression of the input expression to a "temporary variable." Common subexpressions are computed only once, thus reducing the total operation count.

The number of operations, namely additions (or subtractions), multiplications (or divisions) and in particular functions calls of the output is usually lower than the number of such operations of the input. This facility is useful for code generation.

Examples

Example 1

In this first example, we show the effects of optimization for a simple expression:

```
generate::optimize(cos(x^2) + x^2*sin(x^2) + x^4)
```

$$[t2 = x^2,\ t1 = \cos(t2) + t2\ \sin(t2) + t2^2]$$

The "blind" computation of the input expression requires 7 multiplications, 2 additions and 2 function calls. The optimized version introduces a "temporary variable" `t2` storing the subexpression `x^2` that is used to compute the final result `t1`. This reduces the total cost to 3 multiplications, 2 additions and 2 function calls, albeit using 1 extra assignment to the temporary variable `t2`.

Example 2

Here we repeat the exercise of the first example but with an array of expressions:

```
generate::optimize(array(1..2, 1..2, [[x^3, x^2],[x^2, x^4]]))
```

$$\left[t2 = x^2,\ t1 = \begin{pmatrix} t2\,x & t2 \\ t2 & t2^2 \end{pmatrix}\right]$$

The original input requires 6 multiplications. The optimized version needs only 3 multiplications and 1 extra assignment.

Example 3

We optimize a list of equations representing a computation sequence for 3 variables `t`, `C[1]`, `C[2]`:

```
generate::optimize([t = u, C[1] = t*(u - w)^2, C[2] = 2*(u - w)^3])
```

$$\left[t = u,\ t1 = u - w,\ t2 = t1^2,\ C_1 = t\,t2,\ C_2 = 2\,t1\,t2\right]$$

The original computation requires 5 multiplications and 2 subtractions. The optimized version needs 4 multiplications and 1 subtraction.

Note that since these examples involve small expressions, the computational savings are slight. In the case of very large expressions, optimization can yield a considerable dividend.

Parameters

r

An expression, array or list of equations

Return Values

List of equations.

Algorithms

A number of FORTRAN compilers provide optimizers. However, they use algorithms of complexity $O(n^2)$ and $O(n^3)$ where n is the size of the input expressions. For large amounts of code, these algorithms may "break." MuPAD provides a reasonably good scalar (as in non-vectorized and non-parallelized) optimizer which is limited to common subexpression optimization and using binary powering for integer powers. It uses hashing of expressions so that given a sub-expression, it can determine in constant time if this subexpression has already occurred. This results in an overall efficiency which is of lower complexity namely, $O(n)$ i.e. linear in the size of the input expressions to be optimized, Hence overall efficiency is not compromised by very large expressions. This does mean that not all possible optimizations are made but nonetheless a number of reductions including the exploitation of some symmetries are possible.

It should be understood that "optimization" is meant in the sense of compiler optimization. The end-result rarely corresponds to the absolute irreducible minimum number of operations – or as in the case of FORTRAN code generation, the absolute minimum of floating-point operations (FLOPS). Achieving this limit can be extremely difficult if not impossible especially for large computational sequences. Nonetheless, in a number of real-life instances, the MuPAD optimizer can yield a very useful result. Additionally, MuPAD provides symbolic manipulation tools such as `factor` which can yield additional reduction in operation costs.

In many cases of optimization, it is most often a matter of how best to pose the problem so as to fully exploit every possible symmetry or useful natural property of the given problem.

generate::Simscape

Generate Simscape equation

Syntax

```
generate::Simscape(e, <NoWarning>)
```

Description

`generate::Simscape(e)` generates Simscape™ code for the MuPAD expression `e`.

Simscape software extends the Simulink product line with tools for modeling and simulating multidomain physical systems, such as those with mechanical, hydraulic, pneumatic, thermal, and electrical components. Unlike other Simulink blocks, which represent mathematical operations or operate on signals, Simscape blocks represent physical components or relationships directly. With Simscape blocks, you build a model of a system just as you would assemble a physical system. For more information about Simscape software, see “Simscape”.

You can extend the Simscape modeling environment by creating custom components. When you define a component, use the equation section of the component file to establish the mathematical relationships among a component's variables, parameters, inputs, outputs, time, and the time derivatives of each of these entities. MuPAD and Simscape software let you perform symbolic computations and use the results of these computations in the equation section. The `generate::Simscape` function translates the results of symbolic computations to Simscape language equations.

`generate::Simscape` returns a Simscape formatted string representing an expression, equation, list of expressions or equations, or a matrix.

`generate::Simscape` converts the identifier `t` to the variable `time` in the resulting Simscape code. However, the name `t` of a function call does not change during conversion. See “Example 1” on page 10-27 and “Example 2” on page 10-27.

`generate::Simscape` converts any derivative with respect to the variable `t` to the Simscape notation `x.der`, where `x` is the time-dependent variable. See “Example 3” on page 10-28.

`generate::Simscape` assumes that the type of converted data is `double`. See “Example 4” on page 10-28.

When generating Simscape code for a matrix, the generator produces a dense matrix. See “Example 5” on page 10-28.

Use the `generate::optimize` function to optimize the MuPAD code before converting it to the Simscape syntax. See “Example 6” on page 10-29.

`generate::Simscape` converts piecewise expressions to Simscape code by using the `if` statements. See “Example 7” on page 10-29.

The equation section of a Simscape component file supports a limited number of functions. For details and the list of supported functions, see Simscape equations. If a symbolic equation contains the functions that are not available in the equation section of a Simscape component file, `generate::Simscape` cannot correctly convert these equations to Simscape equations. Such expressions do not trigger an error. The following types of expressions are prone to invalid conversion:

- Expressions with infinities
- Expressions that contain programming structures, such as loops, coditional statements (except for the `if` statement), and `map` function calls
- Expressions that contain intervals, sets, and lists

To display generated Simscape code on screen, use the `print` function. To remove quotation marks and to expand special characters like line breaks and tabs, use the printing option `Unquoted`. If a generated code line is longer than the `TEXTWIDTH` setting, the `print` function breaks that line into several shorter lines. The inserted line continuation character (\) is not valid in Simscape. To avoid inserting line continuation characters, increase the `TEXTWIDTH` setting or use the `fprint` function to write generated code to a file.

To write generated Simscape code to a file, use the `fprint` function with the `Unquoted` option. See “Example 8” on page 10-30.

The `NoWarning` option lets you suppress warnings. See “Example 2” on page 10-27.

Examples

Example 1

The `generate::Simscape` function replaces all instances of the MuPAD identifier `t` with the variable `time`. For example, convert the following equation to the Simscape equation:

```
e := A*sin(w*t) + B*cos(w*t) = 0:
print(Unquoted, generate::Simscape(e))
```

```
  B*cos(time*w)+A*sin(time*w) == 0.0;
```

Example 2

The `generate::Simscape` function does not change the function name `t` in function calls:

```
print(Unquoted, generate::Simscape([t(), t(0), t(x)]))
```

```
Warning: Function 't' is not verified to be a valid Simscape function.

Warning: Function 't' is not verified to be a valid Simscape function.

Warning: Function 't' is not verified to be a valid Simscape function.

  t();
  t(0.0);
  t(x);
```

This example produces a few identical warnings. If you started using `generate::Simscape` recently, warnings can help you identify potential issues in the converted code. If you want to suppress warnings, use the `NoWarning` option:

```
print(Unquoted, generate::Simscape([t(), t(0), t(x)], NoWarning))
```

```
t();
t(0.0);
t(x);
```

Example 3

When generating Simscape code, the `generate::Simscape` function converts the derivatives with respect to the variable `t` to the Simscape notation `x.der`. Here `x` is the time-dependent variable. For example, generate the Simscape code for the equation `e` that has two time-dependent variables:

```
e := x'(t) + diff(y(t), t) + 2*x + 5 = 0:
print(Unquoted, generate::Simscape(e))
```

```
x*2.0+x.der+y.der+5.0 == 0.0;
```

Example 4

By default, Simscape stores all numeric values as double-precision floating-point values. In accordance with this default data type, `generate::Simscape` converts the elements of expressions, equations, and matrices to the double format:

```
print(Unquoted, generate::Simscape(x^2 + y/3 + 1/6))
```

```
y*(1.0/3.0)+x^2+1.0/6.0;
```

Example 5

`generate::Simscape` can generate Simscape code for a MuPAD matrix. In contrast to `generate::MATLAB` (which produces sparse matrices), the Simscape code generator produces dense matrices:

```
A:= matrix([[1, 0, 0],[0, 0, 1]]):
print(Unquoted, generate::Simscape(A))
```

```
[
1.0 0.0 0.0
0.0 0.0 1.0
];
```

Example 6

The `generate::Simscape` function does not optimize your code:

```
print(Unquoted,
      generate::Simscape([x = a + b, y = (a + b)^2])):
```

```
  x == a+b;
  y == (a+b)^2;
```

You can use the `generate::optimize` function before converting your MuPAD code to Simscape syntax. For example, this function can reduce the number of operations by finding common subexpressions:

```
print(Unquoted,
      generate::Simscape(
          generate::optimize([x = a + b, y = (a + b)^2])
      )):
```

```
  x == a+b;
  y == x^2;
```

Example 7

The `generate::Simscape` function also accepts piecewise expressions. The function uses `if` statements when generating Simscape code for piecewise expressions. For example, the Fourier transform of the following expression is a piecewise function:

```
FT := fourier(exp(-abs(x)*abs(t))*sin(t)/t, t, s)
```

$$\begin{cases} \arctan\left(\frac{s+1}{|x|}\right) - \arctan\left(\frac{s-1}{|x|}\right) & \text{if } x \neq 0 \end{cases}$$

`generate::Simscape` converts this result to a valid Simscape expression:

```
print(Unquoted, generate::Simscape(FT))
```

```
  if (x ~= 0.0)
    -atan((s-1.0)/abs(x))+atan((s+1.0)/abs(x));
```

```
else
  NaN;
end
```

Example 8

To create a text file with a Simscape formatted string representing a symbolic expression, use the `fprint` command:

```
e := x'(t) + 2*x + 5 = 0:
fprint(Unquoted, Text, "eqn.txt", generate::Simscape(e))
```

If the file `eqn.txt` already exists, `fprint` replaces the existing Simscape code with the converted symbolic expression. You can open and edit the resulting text file.

Parameters

`e`

An expression, equation, list of equations, or a matrix

Options

`NoWarning`

Suppress warnings.

Return Values

`generate::Simscape` returns a `string` containing Simscape code. In case of invalid conversion, the returned value is an arbitrary string.

See Also

MuPAD Functions

`fprint` | `generate::MATLAB` | `generate::optimize` | `print`

generate::TeX

Generate formatted string from expressions

Syntax

```
generate::TeX(e)
```

Description

`generate::TeX(e)` returns a TeX formatted string representing `e`. This string may be printed to a file using `fprint`. Use the printing option `Unquoted` to remove quotes and to expand special characters like newlines and tabs.

The output string may be used in the math-mode of TeX. Note that `generate::TeX` doesn't break large formulas into smaller ones.

Examples

Example 1

`generate::TeX` generates a string containing the TeX code:

```
generate::TeX(hold(int)(exp(x^2)/x, x))
```

```
"\int \frac{\mathrm{e}^{x^2}}{x} \,\mathrm{d} x"
```

Use `print` with option `Unquoted` to get a more readable output:

```
print(Unquoted, generate::TeX(hold(int)(exp(x^2)/x, x)))
```

```
\int \frac{\mathrm{e}^{x^2}}{x} \,\mathrm{d} x
```

Example 2

This example shows how to write a `"TeX"`-method for a domain. The domain elements represent open intervals. The `"TeX"`-method makes recursive use of `generate::TeX` in order to TeX-format its operands and concatenates the resulting strings to a new string containing the TeX output of the interval.

```
Interval := newDomain("Interval"):
Interval::TeX :=
  e -> "\\left]".generate::TeX(extop(e, 1)).
       ", ".generate::TeX(extop(e, 2))."\\right[":
print(Unquoted,
      generate::TeX(new(Interval, 1, x^(a+2)))):
```

```
\left]1, x^{a + 2}\right[
```

Parameters

e

An arithmetical expression

Return Values

`generate::TeX` returns a string containing TeX code.

Overloaded By

e

Algorithms

The TeX packages amsmath and amssymb are needed.

A domain overloading `generate::TeX` has to provide a function as its `"TeX"`-slot which translates its elements into a TeX formatted string. This function may use `generate::TeX` recursively. See “Example 2” on page 10-32.

See Also

MuPAD Functions
`fprint` | `print`

Graph – Graph Theory

Graph::inDegree
Graph::isConnected
Graph::isDirected
Graph::isEdge
Graph::isVertex
Graph::longestPath
Graph::maxFlow
Graph::minCost
Graph::minCut
Graph::minimumSpanningTree
Graph
Graph::outDegree
Graph::plotBipartiteGraph
Graph::plotCircleGraph
Graph::plotGridGraph
Graph::printEdgeCostInformation
Graph::printEdgeDescInformation
Graph::printEdgeInformation
Graph::printEdgeWeightInformation
Graph::printGraphInformation
Graph::printVertexInformation
Graph::removeEdge
Graph::removeVertex
Graph::residualGraph
Graph::revert
Graph::setEdgeCosts
Graph::setEdgeDescriptions
Graph::setEdgeWeights
Graph::setVertexWeights
Graph::shortestPathAllPairs
Graph::shortestPathSingleSource
Graph::stronglyConnectedComponents
Graph::topSort

Graph::addEdges

Adds one or several edges to a graph

Syntax

```
Graph::addEdges(G, Edge)

Graph::addEdges(G, Edge, <EdgeWeights = ew>, <EdgeCosts = ec>, <EdgeDescriptions = ed>)
```

Description

`Graph::addEdges` adds one or several edges to an already existing Graph. An edge is represented by a list containing two vertices of the graph. A warning is raised if one of the specified edges does already exist in the graph.

`Graph::addEdges(G,Edge)` adds the edge(s) `Edge` to the graph `G`. The two vertices of each edge must be vertices in the given graph. Otherwise an error is raised. If an edge is specified that already exists, a warning will be printed that this edge is not used.

With `Graph::addEdges(G, Edge,EdgeWeights=ew,EdgeCosts=ec, EdgeDescriptions=ed)` the weight, cost and description of each edge can be set to every edge additionally. If these specifications are missing, the default value `0(=None)` is assumed. If a specification is used it has to hold exactly the same number of values as there are edges. Otherwise an error will be raised.

Note: The value `None` can be used in the specification lists for every edge that is not to be specified explicitly.

Examples

Example 1

First, an undirected graph with two vertices and no edges is created. Then two edges are added:

```
G := Graph([a, b, c, d], []):
Graph::printEdgeInformation(G):
G := Graph::addEdges(G, [[a, b], [c, d]]):
Graph::printEdgeInformation(G)
```

```
No edges.
```

```
Edges existing in the graph:
----------------------------
[a, b], [c, d], [b, a], [d, c]
```

As you can see, `[b,a]` and `[d,c]` were inserted automatically.

```
G2 := Graph::addEdges(G, [[a,d]]):Graph::getEdges(G2)
```

$$[[a, d], [d, a], [a, b], [c, d], [b, a], [d, c]]$$

Now, what happens if an edge is inserted that already exists in the graph?

```
G := Graph::addEdges(G, [[d, c]])
```

```
Warning: The following edges were not used for operation: [[d, c]]. [Graph::selectEdge
```

$$\text{Graph}(\ldots)$$

Suppose, we try to insert an edge with a vertex not existing in the graph:

```
G := Graph::addEdges(G, [[a, 5]])
```

```
Error: One or more edges contain vertices that are not in list '[5]'. [Graph::addEdges
```

Now let's see what happens when a directed graph is created:

```
G := Graph([a, b, c, d], [], Directed):
G := Graph::addEdges(G, [[a, b], [b, c], [c, d]],
                     EdgeWeights = [2/8, -5, PI],
                     EdgeCosts = [30, -40, None]):
Graph::printGraphInformation(G)
```

```
Vertices: [a, b, c, d]
```

```
Edges: [[a, b], [b, c], [c, d]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: [a, b] = 1/4, [b, c] = -5, [c, d] = PI (other existing edges\
 have no weight)
Edge costs: [a, b] = 30, [b, c] = -40 (other existing edges have costs zer\
o)
Adjacency list (out): a = [b], b = [c], c = [d], d = []
Adjacency list (in): a = [], b = [a], c = [b], d = [c]
Graph is directed.
```

Have a close look at the Edge costs line. The edge `[c,d]` is not mentioned explicitly due to the value `None`:

```
G2 := Graph::addEdges(G, [[a, b], [a, d]], EdgeWeights=[10, 20],
                      EdgeCosts = [80, 90],
                      EdgeDescriptions = ["First way", "Second way"]):
Graph::printGraphInformation(G2)
```

```
Warning: The following edges were not used for operation: [[a, b]]. [Graph::selectEdge]
```

```
Vertices: [a, b, c, d]
Edges: [[a, d], [a, b], [b, c], [c, d]]
Vertex weights: no vertex weights.
Edge descriptions: [a, d] = "Second way"
Edge weights: [a, b] = 1/4, [b, c] = -5, [c, d] = PI, [a, d] = 20 (other e\
xisting edges have no weight)
Edge costs: [a, b] = 30, [b, c] = -40, [a, d] = 90 (other existing edges h\
ave costs zero)
Adjacency list (out): a = [b, d], b = [c], c = [d], d = []
Adjacency list (in): a = [], b = [a], c = [b], d = [a, c]
Graph is directed.
```

If an edge has specifications, but exist already in the graph, the specifications will not change. (see Information for edge `[a, b]` above)

Parameters

G

`Graph`

`Edge`

List of one or more edges

`ew, ec`

Lists of numbers

`ed`

List of texts

`b`

Boolean value

Options

`EdgeWeights`

The weight(s) of the new edge(s). Default is 0.

`EdgeCosts`

The cost(s) of the new edge(s). Default is 0.

`EdgeDescriptions`

The description(s) for the new edge(s). Default is no text.

Return Values

`Graph` with the correct edges inserted.

Graph::addVertices

Adds one or several vertices to a graph

Syntax

```
Graph::addVertices(G, Vertex, <VertexWeights = vw>)
```

Description

`Graph::addVertices` adds one or several vertices to an already existing graph. A vertex is represented by an arbitrary expression. A warning is raised if one of the specified vertices does already exist in the graph.

`Graph::addVertices(G, Vertex)` adds the vertices in `Vertex` to the graph `G`. If a vertex is specified that already exists, a warning will be printed that this vertex (and it's vertex weight) is not used.

With `Graph::addVertices(G, Vertex,VertexWeights=vw)` the weight can be set to every vertex additionally. If these specifications are missing, the default value `0(=None)` is assumed. If a specification is used it has to hold exactly the same number of values as there are vertices. Otherwise an error will be raised.

Note: The value `None` can be used in the specification lists for every edge that is not to be specified explicitly.

Examples

Example 1

First, an undirected graph with two vertices and no edges is created. Then two vertices are added:

```
G := Graph([a, b, c, d], []):
Graph::printVertexInformation(G):
```

```
G := Graph::addVertices(G, [e, f]):
Graph::printVertexInformation(G)
```

```
Vertices existing in the graph:
------------------------------
Vertex a has weight None
Vertex b has weight None
Vertex c has weight None
Vertex d has weight None
```

```
Vertices existing in the graph:
------------------------------
Vertex a has weight None
Vertex b has weight None
Vertex c has weight None
Vertex d has weight None
Vertex e has weight None
Vertex f has weight None
```

No weights were specified, so every vertex has weight `None`. In the algorithms default-values will be used accordingly.

Now, what happens if a vertex is inserted that already exists in the graph?

```
G2 := Graph::addVertices(G, [a, g], VertexWeights=[10, 100]):
Graph::printVertexInformation(G2)
```

```
Warning: The following vertices already exist: [a]. [Graph::selectVertex]
```

```
Vertices existing in the graph:
------------------------------
Vertex a has weight None
Vertex b has weight None
Vertex c has weight None
Vertex d has weight None
Vertex e has weight None
Vertex f has weight None
Vertex g has weight 100
```

If a vertex weight is specified for a vertex already existing, it will not be changed (see Vertex `a` above)

Parameters

`G`

`Graph`

`Vertex`

List of one or more vertices

`vw`

Lists of numbers

Options

`VertexWeights`

The weight(s) of the new vertex/vertices. Default is 0.

Return Values

`Graph` with the correct vertices inserted.

Graph::admissibleFlow

Checks a flow for admissibility in a Graph

Syntax

```
Graph::admissibleFlow(G, f)
```

Description

`Graph::admissibleFlow(G, f)` checks if the flow `f` is admissible in the Graph `G` according to its vertices and their capacities.

`Graph::admissibleFlow` checks whether a given flow is an admissible flow in the specified graph. A flow in a graph is a table `t`, where `t[[i,j]]` gives the number of units flowing from vertex `i` to vertex `j`. `Graph::admissibleFlow` returns `TRUE` if the flow is admissible. Otherwise `FALSE` is returned.

`Graph::admissibleFlow` does not check whether the flow is admissible, if a flow from vertex `i` to vertex `j` is allowed to pass through other vertices. See “Example 2” on page 11-11.

Examples

Example 1

In a cyclic graph with default capacities (1), the flow with one unit flowing from each vertex to its successor is certainly admissible:

```
G1 := Graph::createCircleGraph([v1, v2, v3, v4]):
Graph::admissibleFlow(G1, table([v1, v2] = 1,
                                [v2, v3] = 1,
                                [v3, v4] = 1,
                                [v4, v1] = 1))
```

```
    TRUE
```

Example 2

The flow must be specified in whole. `Graph::admissibleFlow` does not include “hops”, like skipping vertices in the path:

```
Graph::admissibleFlow(Graph::createCircleGraph([v1, v2, v3]),
                                    table([v1, v3] = 1))
```

```
    FALSE
```

Parameters

`G`

Graph

`f`

The flow, specified in a table

Return Values

Either TRUE or FALSE

Graph::bipartite

Finds out if a graph is bipartite.

Syntax

```
Graph::bipartite(G, <Bool | Lists>)
```

Description

`Graph::bipartite(G)` finds out whether G is bipartite or not.

`Graph::bipartite(G, Sets)`: If G is bipartite, then a list containing two lists will be returned. Each of the lists contains the vertices belonging to the set. If G is not bipartite, then `FAIL` will be returned instead of any list.

`Graph::bipartite(G, Bool)` offers the same result as `Graph::bipartite(G)`. If G is bipartite, then `TRUE` will be returned, otherwise `FALSE`.

Examples

Example 1

A small graph containing 3 vertices with 2 edges connecting them is created:

```
G := Graph([a, b, c], [[a, b], [b, c]]):
Graph::bipartite(G, Lists);
Graph::bipartite(G, Bool)
```

$$[[b], [a, c]]$$

$$\text{TRUE}$$

Two lists with vertices are shown. Another word for bipartite is two-colorable. This means that the graph above can be colored with only two colors so that no two vertices

have the same color if connected with an edge. The bottom output could also be accomplished without using the parameter `Bool`:

```
Graph::bipartite(G)
```

```
    TRUE
```

The following example shows what happens when a graph is not bipartite (an edge is added to connect the vertices a and c):

```
G2 := Graph::addEdges(G, [[a, c]]):
Graph::bipartite(G2, Lists);
Graph::bipartite(G2, Bool)
```

```
    FAIL
```

```
    FALSE
```

Parameters

`G`

`Graph`

Options

`Lists`

If Lists is stated the return value will be a list of two lists containing the (sorted) vertices belonging to each set, or `FAIL`.

`Bool`

If Bool is stated the return value will be either `TRUE` or `FALSE`. This is the default.

Return Values

Depending on the options either a boolean value or list-sets will be returned.

Graph::breadthFirstSearch

Makes a breadth first Search in a graph.

Syntax

```
Graph::breadthFirstSearch(G, <StartVertex = v>)
```

Description

`Graph::breadthFirstSearch` traverses through a graph via breadth first search. The output shows the first time of identification and the predecessor of each vertex. If a vertex is a single vertex with no predecessor its predecessor is *infinity*.

`Graph::breadthFirstSearch(G, StartVertex = v)` traverses through a graph via breadth first search starting from vertex `v`. The output shows the first time of identification and the predecessor of each vertex. If a vertex is a single vertex with no predecessor its predecessor is *infinity*.

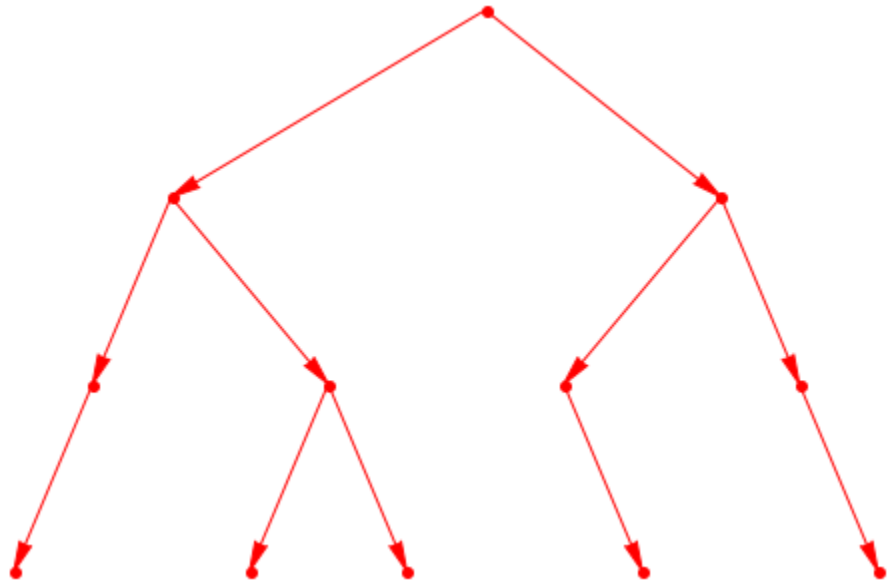
Examples

Example 1

A typical tree is created and drawn for a better understanding of the algorithm:

```
G := Graph([a, b, c, d, e, f, g, h, i, j, k, l],
           [[a, b], [a, c], [b, d], [b, e], [c, f], [c, g],
            [d, h], [e, i], [e, j], [f, k], [g, l]],
           Directed):
plot(
  Graph::plotGridGraph(G, VerticesPerLine = [12, 12, 12, 12],
    VertexOrder = [
  None, None, None, None, None, None,
  a,    None, None, None, None, None,
  None, None, b,    None, None, None,
  None, None, None, c,    None, None,
  None, d,    None, None, e,    None,
  None, f,    None, None, g,    None,
```

```
  h,    None, None, i,    None, j,
  None, None, k,    None, None, l
    ]
  )
)
```



Now we call breadthFirstSearch to find out the starting times and predecessors

```
Graph::breadthFirstSearch(G)
```

a	1	a	0	a	a
b	2	b	1	b	a
c	3	c	1	c	a
d	4	d	2	d	b
e	5	e	2	e	b
f	6	f	2	f	c
g	7	g	2	g	c
h	8	h	3	h	d
i	9	i	3	i	e
j	10	j	3	j	e
k	11	k	3	k	f
l	12	l	3	l	g

Vertex `a` is dicovered first, then vertex b and so on. The right table shows the predecessor of every vertex. The backtracking from a single vertex is therefore really simple. `a` as the first vertex discovered in its component can not be backtracked any further. The distance of each vertex in its component can be read in the middle table. Root-vertices always have the value `0` (they `are` the roots).

Example 2

What happens now, if there exist a vertex that has no connection to any other vertex. The upper example is taken and a single vertex is added without changing anything else. Then a breadth first search is invoked on the graph:

```
G := Graph([a, b, c, d, e, f, g, h, i, j, k, l],
           [[a, b], [a, c], [b, d], [b, e], [c, f], [c, g],
            [d, h], [e, i], [e, j], [f, k], [g, l]],
           Directed):

G2 := Graph::addVertices(G, [m]):
Graph::breadthFirstSearch(G2, StartVertex = [a])
```

a	1	a	0	a	a
b	2	b	1	b	a
c	3	c	1	c	a
d	4	d	2	d	b
e	5	e	2	e	b
f	6	f	2	f	c
g	7	g	2	g	c
h	8	h	3	h	d
i	9	i	3	i	e
j	10	j	3	j	e
k	11	k	3	k	f
l	12	l	3	l	g
m	13	m	0	m	∞

The newly inserted vertex `m` has no predecessor. The predecessor therefore holds the value *infinity*.

Example 3

If we start somewhere in the graph without knowing the root of the DAG, the results are of course different:

```
G := Graph([a, b, c, d, e, f, g, h, i, j, k, l],
           [[a, b], [a, c], [b, d], [b, e], [c, f], [c, g],
            [d, h], [e, i], [e, j], [f, k], [g, l]],
           Directed):

Graph::breadthFirstSearch(G, StartVertex = [c])
```

$$\begin{bmatrix} a & 6 \\ b & 7 \\ c & 1 \\ d & 8 \\ e & 9 \\ f & 2 \\ g & 3 \\ h & 10 \\ i & 11 \\ j & 12 \\ k & 4 \\ l & 5 \end{bmatrix}, \begin{bmatrix} a & 0 \\ b & 1 \\ c & 0 \\ d & 2 \\ e & 2 \\ f & 1 \\ g & 1 \\ h & 3 \\ i & 3 \\ j & 3 \\ k & 2 \\ l & 2 \end{bmatrix}, \begin{bmatrix} a & a \\ b & a \\ c & c \\ d & b \\ e & b \\ f & c \\ g & c \\ h & d \\ i & e \\ j & e \\ k & f \\ l & g \end{bmatrix}$$

The predecessor of `c` is `c`, but if we look at the graph it should be `a`. This is nevertheless not quite correct. Breadth first search takes the given vertex and uses this as the root of the graph (no in-vertices!). This explains also why the next call shows a *infinity* as predecessor to l.

Parameters

`G`

`Graph`

`v`

List containing one vertex.

Options

`StartVertex`

Defines a vertex from which to start the breadth first traversal.

Return Values

List containing three tables. The first table holds the timestamp of the discovery. The second the distance to the root-vertex. The last table holds the predecessor vertices.

Graph::checkForVertices

Checks if all vertices in edges really exist.

Syntax

```
Graph::checkForVertices(Edge, Vertex)
```

Description

`Graph::checkForVertices(Edge, Vertex)` checks if all vertices out of Edge are in Vertex.

Examples

Example 1

What vertices are within the stated edges, but not in the vertex list?

```
Graph::checkForVertices([[a, b], [1, 2]], [a, 2])
```

$$[b, 1]$$

Neither b nor 1 were in the second list. a was in the first edge and 2 in the second.

Example 2

A more complex example. The second list contains a vertex that does not exist in the graph at all. For the checking it does not matter though. Every vertex NOT in the second list is to be returned. In the end it does not matter if the vertex-list contains vertices that are not existent, because only existing vertices are returned.

```
G := Graph::createCompleteGraph(10):
Graph::checkForVertices(Graph::getEdges(G), [1, 2, 3, 11])
```

```
[4, 5, 6, 7, 8, 9, 10]
```

Parameters

`Edge`

List of one or more Edges

`Vertex`

List of one or more vertices

Return Values

List with the vertices out of the Edges that were not stated in Vertex.

Graph::chromaticNumber

Chromatic number of a graph

Syntax

```
Graph::chromaticNumber(G)
```

Description

`Graph::chromaticNumber(G)` returns the chromatic number of the graph `G`. The chromatic number of a graph is defined to be the number of colors necessary to color it such that no two adjacent vertices have the same color.

Examples

Example 1

We compute the chromatic number of the complete graph with 5 vertices; it must be 5 since any two vertices are adjacent:

```
Graph::chromaticNumber(Graph::createCompleteGraph(5))
```

```
    5
```

Parameters

`G`

An undirected graph

Return Values

Positive integer

Algorithms

Internally, the chromatic polynomial is used to compute the chromatic number.

See Also

MuPAD Functions

`Graph::chromaticPolynomial`

Graph::chromaticPolynomial

Calculates a chromatic polynomial

Syntax

```
Graph::chromaticPolynomial(G, x)
```

Description

`Graph::chromaticPolynomial(G, x)` returns the chromatic polynomial of the graph `G`. Evaluating the result at $x = n$, for any integer n, gives the number of possible ways to color the graph `G` using n colors such that no two adjacent vertices have the same color.

`G` must be an undirected graph: if an edge goes from a to$b$, another edge must go from b to a, for any two vertices$a$, b.

Examples

Example 1

We compute the chromatic polynomial of the complete graph with 5 vertices:

```
f:= Graph::chromaticPolynomial(Graph::createCompleteGraph(5), x)
```

$$\mathrm{poly}(x^5 - 10\,x^4 + 35\,x^3 - 50\,x^2 + 24\,x, [x])$$

There are 240 ways to color a complete graph with 5 vertices, since this is the number of bijective mappings between the set of colors and the set of vertices:

```
f(5)
```

$$120$$

```
delete f:
```

Example 2

Now let us delete one edge from a complete graph:

```
G:= Graph::createCompleteGraph(5):
G:= Graph::removeEdge(G, [[2, 3]]):
G:= Graph::removeEdge(G, [[3, 2]])
```

```
Graph(...)
```

Now there are some additional possible colourings: vertices 2 and 3 may now have the same color, in five different ways; in each case, there must be one of the four remaining colors that does not occur at all. In each of the 20 cases, we are left with 3 vertices that form a complete graph and 3 colors, such that there are 6 colourings. Altogether this gives us 120 additional colourings:

```
Graph::chromaticPolynomial(G, x)(5)
```

```
240
```

Parameters

G

An undirected graph

x

An identifier

Return Values

polynomial

Algorithms

Computing the chromatic polynomial of a graph with n vertices reduces to computing two chromatic polynomials of graphs with n - 1 vertices. The running time is hence roughly 2^n.

References

See Birkhoff and Lewis, *Chromatic Polynomials*, Trans. AMS, Vol. 60, p.355–451, 1946.

Graph::contract

Contracts vertices

Syntax

```
Graph::contract(G, VertexTable)
```

Description

`Graph::contract(G, VertexTable)` contracts the vertices for each entry in the table.

Note: The graph to be contracted must not have edge weights, costs or descriptions. If it has, an error will be raised.

Note: If `VertexTable` contains vertices not in `G`, these will be skipped.

Examples

Example 1

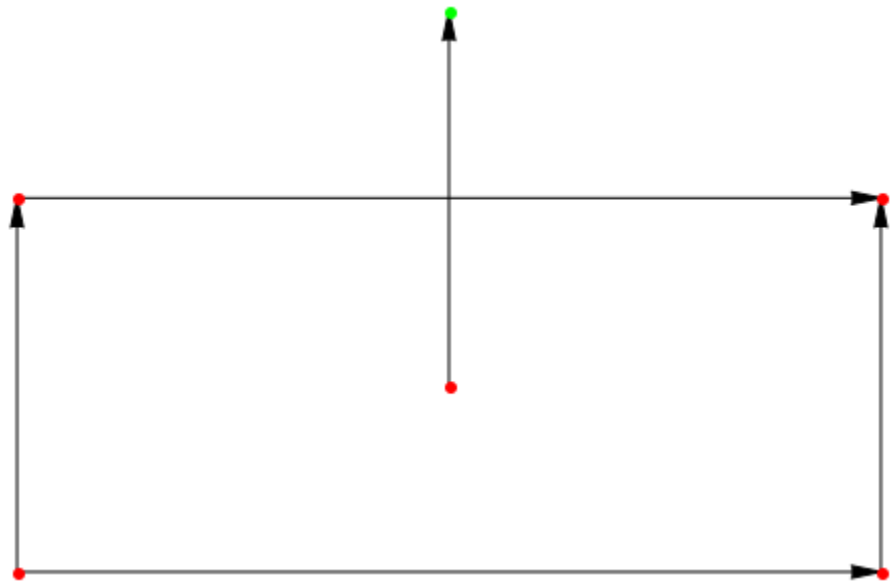
A simple example to see how a contraction is done.

```
ConG := Graph([a, b, c, d, e, f],
              [[a, c], [d, a], [f, c], [d, f], [e, b]],
              Directed):
Graph::printGraphInformation(ConG)
```

```
Vertices: [a, b, c, d, e, f]
Edges: [[a, c], [d, a], [d, f], [e, b], [f, c]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
```

```
Adjacency list (out): a = [c], b = [], c = [], d = [a, f], e = [b], f = [c]
Adjacency list (in): a = [d], b = [e], c = [a, f], d = [], e = [], f = [d]
Graph is directed.
```

```
plot(
  Graph::plotGridGraph(ConG,
       VerticesPerLine = 3,
       VertexOrder = [None, b,    None,
                      a,    None, c,
                      None, e,    None,
                      d,    None, f],
       EdgeColor = RGB::Black))
```



```
t := table(A = [a, b, c],
           B = [d, f])
```

A	$[a, b, c]$
B	$[d, f]$

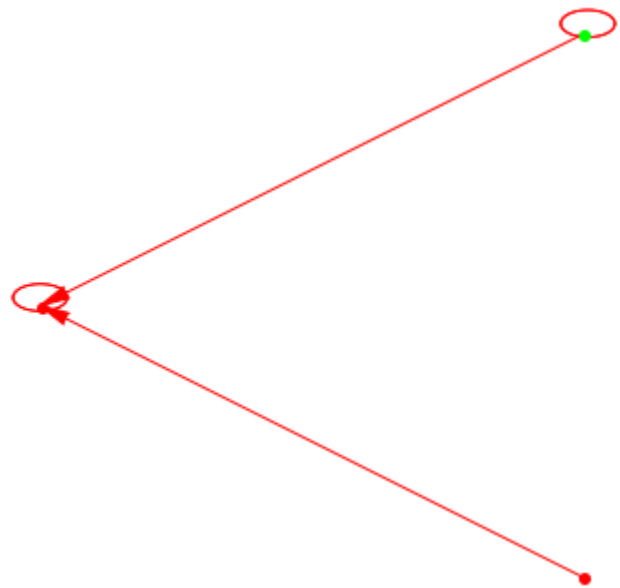
```
newG := Graph::contract(ConG, t):
```

```
Graph::printGraphInformation(newG)
```

```
Vertices: [A, B, e]
Edges: [[A, A], [B, A], [B, B], [e, A]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): e = [A], A = [A], B = [A, B]
Adjacency list (in): e = [], A = [A, B, e], B = [B]
Graph is directed.
```

Since vertices *a*, *b*, *c* were contracted to vertex *A*, edge [a,c] was removed and edge [A, A] was created. Vertices *d*, *f* took care of the deletion of edges [*d*, *a*], [*d*, *f*], [*f*, *c*]. Instead edges [*B*, *A*] and[*B*, *B*] were created. In the end edge [*e*, *b*] was changed to [*e*, *A*] since vertex *b* does not exist any longer because it was replaced by *A*.

```
plot(Graph::plotGridGraph(newG,
          VerticesPerLine = 2,
          VertexOrder = [None, B,
                         A,    None,
                         None, e]))
```



Example 2

`Graph::contract` ignores vertices not in the graph:

```
Con2 := Graph([], []):
t := table(A = [a, b, c], B = [d, f]):
Graph::printGraphInformation(Graph::contract(Con2, t))
```

```
Vertices: no vertices.
Edges: []
Adjacency list (out): no edges.
Adjacency list (in): no edges.
Graph is undirected.
```

Parameters

`G`

`Graph`

`VertexTable`

A table with the name of the new vertex on the left side and a list of vertices to contract on the right side.

Return Values

`Graph` consisting of the new vertices and edges.

Graph::convertSSQ

Converts a Graph into a single source single sink Graph

Syntax

```
Graph::convertSSQ(G, q, s)
```

Description

`Graph::convertSSQ(G, q, s)` converts the graph `G` into a directed single source single sink graph. The specified vertices `q` and `s` are added to the graph. It is an error if they are already predefined. Otherwise they are connected to the other vertices of the graph in the following way:

A new edge `[q,i]` is added for every vertex `i` with a positive weight. A new edge `[i,s]` is added for every vertex `i` with a negative weight. The capacities of these edges are in each case the weight of node `i`. The edge weights are zero.

Examples

Example 1

A testexample to show the transformation.

```
V  := [1, 2, 3, 4]:
Vw := [4, 0, 0, -4]:
Ed := [[1, 2], [1, 3], [2, 3], [2, 4], [3, 4]]:
Ec := [2, 2, 1, 3, 1]:
Ew := [4, 2, 2, 3, 5]:
G1 := Graph(V, Ed, VertexWeights = Vw,
            EdgeWeights = Ew,EdgeCosts = Ec):
G2 := Graph::convertSSQ(G1, [q], [s]):
Graph::printGraphInformation(G2)
```

```
Vertices: [1, 2, 3, 4, q, s]
Edges: [[1, 2], [1, 3], [2, 1], [2, 3], [2, 4], [3, 1], [3, 2], [3, 4], [4\
, 2], [4, 3], [4, s], [q, 1]]
Vertex weights: 1 = 0, 2 = 0, 3 = 0, 4 = 0, q = 4, s = -4 (other existing \
vertices have no weight)
Edge descriptions: no edge descriptions.
Edge weights: [1, 2] = 4, [1, 3] = 2, [2, 3] = 2, [2, 4] = 3, [3, 4] = 5, \
[2, 1] = 4, [3, 1] = 2, [3, 2] = 2, [4, 2] = 3, [4, 3] = 5, [q, 1] = 4, [4\
, s] = 4 (other existing edges have no weight)
Edge costs: [1, 2] = 2, [1, 3] = 2, [2, 3] = 1, [2, 4] = 3, [3, 4] = 1, [2\
, 1] = 2, [3, 1] = 2, [3, 2] = 1, [4, 2] = 3, [4, 3] = 1, [q, 1] = 0, [4, \
s] = 0 (other existing edges have costs zero)
Adjacency list (out): 1 = [2, 3], 2 = [1, 3, 4], 3 = [1, 2, 4], 4 = [2, 3,\
 s], q = [1], s = []
Adjacency list (in): 1 = [2, 3, q], 2 = [1, 3, 4], 3 = [1, 2, 4], 4 = [2, \
3], q = [], s = [4]
Graph is directed.
```

The former undirected graph was transformed into a directed one!

Parameters

q, s

Vertices not predefined in the Graph

G

A Graph

Return Values

Directed augmented Graph

Algorithms

Both, Bellman and Dijkstra expect a Graph without negative circles. Only Dijkstra may return erroneous results when negative edges (either weights or costs) are specified.

The Bellman algorithm originated from: Ahuja, Magnanti, Orlin: Dom::Graph Flows, Prentice-Hall, 1993 Section 5.4

Graph::createCircleGraph

Generates a circle Graph

Syntax

```
Graph::createCircleGraph(L, <Directed | Undirected>)
```

```
Graph::createCircleGraph(N, <Directed | Undirected>)
```

Description

`Graph::createCircleGraph(L)` generates a circle Graph

`Graph::createCircleGraph([v1,...,vn])` generates a new graph which is the cycle `[v1,v2]`, `[v2,v3]`, ..., `[vn,v1]`. The values for the edge weights, edge costs and vertex weights can be set manually via `Graph::setEdgeWeights`, `Graph::setEdgeCosts` and `Graph::setVertexWeights`

`Graph::createCircleGraph(3)` generates a new graph which is the cycle `[1,2]`, `[2,3]`, `[3,1]`. The values for the edge weights, edge capacities and vertex weights can be set manually via `Graph::setEdgeWeights`, `Graph::setEdgeCosts` and `Graph::setVertexWeights`

`Graph::createCircleGraph(3, Undirected)` generates a new graph which includes the vertices `[1,2]`, `[2,3]`, `[3,1][2,1]`, `[3,2]`, `[1,3]`.

Examples

Example 1

A (directed) circle graph with four vertices:

```
G1 := Graph::createCircleGraph(4):
Graph::printGraphInformation(G1)
```

```
Vertices: [1, 2, 3, 4]
Edges: [[1, 2], [2, 3], [3, 4], [4, 1]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [2], 2 = [3], 3 = [4], 4 = [1]
Adjacency list (in): 1 = [4], 2 = [1], 3 = [2], 4 = [3]
Graph is directed.
```

Example 2

The same graph but this time with parameter `Undirected`:

```
G2 := Graph::createCircleGraph(4, Undirected):
Graph::printGraphInformation(G2)
```

```
Vertices: [1, 2, 3, 4]
Edges: [[1, 2], [1, 4], [2, 1], [2, 3], [3, 2], [3, 4], [4, 1], [4, 3]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [2, 4], 2 = [1, 3], 3 = [2, 4], 4 = [1, 3]
Adjacency list (in): 1 = [2, 4], 2 = [1, 3], 3 = [2, 4], 4 = [1, 3]
Graph is undirected.
```

Example 3

The circle graph with predefined vertices:

```
G3 := Graph::createCircleGraph([a, b, c, d, 4, 5, 6]):
Graph::printGraphInformation(G3)
```

```
Vertices: [4, 5, 6, a, b, c, d]
Edges: [[4, 5], [5, 6], [6, a], [a, b], [b, c], [c, d], [d, 4]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): a = [b], b = [c], c = [d], d = [4], 4 = [5], 5 = [6]\
```

```
, 6 = [a]
Adjacency list (in): a = [6], b = [a], c = [b], d = [c], 4 = [d], 5 = [4],\
 6 = [5]
Graph is directed.
```

Parameters

`L`

List of vertices

`N`

A positive Integer

Options

`Directed`

The Graph is created as a directed graph. `Default`.

`Undirected`

The Graph is created as an undirected graph.

Return Values

Graph

Graph::createCompleteGraph

Generates a complete graph

Syntax

```
Graph::createCompleteGraph(n)
```

Description

`Graph::createCompleteGraph(n)` generates the complete Graph with `n` vertices. A complete graph has a connection between each pair of vertices (except to itself).

The vertices of the generated graph are labeled with the numbers `1` to `n`.

Examples

Example 1

The complete Graph with three vertices has 3 2 = 6 edges:

```
G := Graph::createCompleteGraph(3):
Graph::printGraphInformation(G)
```

```
Vertices: [1, 2, 3]
Edges: [[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [2, 3], 2 = [1, 3], 3 = [1, 2]
Adjacency list (in): 1 = [2, 3], 2 = [1, 3], 3 = [1, 2]
Graph is undirected.
```

Parameters

n

A positive integer

Return Values

Graph

Graph::createGraphFromMatrix

Transfers a squared matrix into a directed graph

Syntax

```
Graph::createGraphFromMatrix(M)
```

Description

`Graph::createGraphFromMatrix(M)` generates a directed Graph where each $m_{i,j}$ in the matrix defines an edge from$i$ to j. The value of the cell defines the weight of the resulting edge.

The vertices of the generated graph are labeled with the numbers `1` to `n`, where n defines the column/row-dimension of the matrix. Since the matrix has to be squared, n stays the same.

Examples

Example 1

A matrix is defined and the resulting squared matrix is transfered into a Graph.

```
a := matrix([[3, 2, 4], [2, 3, 6], [4, 8, 3]]);
G := Graph::createGraphFromMatrix(a):
Graph::printGraphInformation(G)
```

$$\begin{pmatrix} 3 & 2 & 4 \\ 2 & 3 & 6 \\ 4 & 8 & 3 \end{pmatrix}$$

```
Vertices: [1, 2, 3]
Edges: [[1, 1], [1, 2], [1, 3], [2, 1], [2, 2], [2, 3], [3, 1], [3, 2], [3\
, 3]]
```

```
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: [1, 1] = 3, [1, 2] = 2, [1, 3] = 4, [2, 1] = 2, [2, 2] = 3, \
[2, 3] = 6, [3, 1] = 4, [3, 2] = 8, [3, 3] = 3 (other existing edges have \
no weight)
Edge costs: no edge costs.
Adjacency list (out): 1 = [1, 2, 3], 2 = [1, 2, 3], 3 = [1, 2, 3]
Adjacency list (in): 1 = [1, 2, 3], 2 = [1, 2, 3], 3 = [1, 2, 3]
Graph is directed.
```

Parameters

M

A matrix

Return Values

Graph

Graph::createRandomEdgeWeights

Sets random weights to edges

Syntax

```
Graph::createRandomEdgeWeights(G, r, <Dom::Integer | Dom::Real>)
```

Description

`Graph::createRandomEdgeWeights(G, x..y)` creates random integer edge weights within the range `x..y`.

`Graph::createRandomEdgeWeights(G, x..y, Dom::Integer)` does exactly the same.

`Graph::createRandomEdgeWeights(G, x..y, Dom::Real)` creates random real edge weights within the range `x..y`.

Note: Already existing edge weights will be changed, too!

Examples

Example 1

Creating edge weights for a small cyclic graph. First, it has no specified weights:

```
G1 := Graph::createCircleGraph(5):
Graph::printGraphInformation(G1)
```

```
Vertices: [1, 2, 3, 4, 5]
Edges: [[1, 2], [2, 3], [3, 4], [4, 5], [5, 1]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [2], 2 = [3], 3 = [4], 4 = [5], 5 = [1]
```

```
Adjacency list (in): 1 = [5], 2 = [1], 3 = [2], 4 = [3], 5 = [4]
Graph is directed.
```

Now the weights are set within the range `-100..50` (your output may differ due random assignment):

```
G2 := Graph::createRandomEdgeWeights(G1, -100..50):
Graph::printGraphInformation(G2)
```

```
Vertices: [1, 2, 3, 4, 5]
Edges: [[1, 2], [2, 3], [3, 4], [4, 5], [5, 1]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: [1, 2] = 47, [2, 3] = -12, [3, 4] = 28, [4, 5] = 1, [5, 1] =\
 -36 (other existing edges have no weight)
Edge costs: no edge costs.
Adjacency list (out): 1 = [2], 2 = [3], 3 = [4], 4 = [5], 5 = [1]
Adjacency list (in): 1 = [5], 2 = [1], 3 = [2], 4 = [3], 5 = [4]
Graph is directed.
```

If the weights should be of type Real it can be set optionally:

```
G2 := Graph::createRandomEdgeWeights(G1, -100..50, Dom::Real):
Graph::printGraphInformation(G2)
```

```
Vertices: [1, 2, 3, 4, 5]
Edges: [[1, 2], [2, 3], [3, 4], [4, 5], [5, 1]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: [1, 2] = -67.72964183, [2, 3] = -10.16896282, [3, 4] = -72.8\
4684348, [4, 5] = -61.00518722, [5, 1] = 18.2662729 (other existing edges \
have no weight)
Edge costs: no edge costs.
Adjacency list (out): 1 = [2], 2 = [3], 3 = [4], 4 = [5], 5 = [1]
Adjacency list (in): 1 = [5], 2 = [1], 3 = [2], 4 = [3], 5 = [4]
Graph is directed.
```

Parameters

G

A graph

r

A range

Return Values

Graph

Graph::createRandomEdgeCosts

Sets random costs to edges

Syntax

```
Graph::createRandomEdgeCosts(G, r, <Dom::Integer | Dom::Real>)
```

Description

`Graph::createRandomEdgeCosts(G, x..y)` creates random edge weights of type Integer within the range `x..y`.

`Graph::createRandomEdgeCosts(G, x..y, Dom::Integer)` does exactly the same.

`Graph::createRandomEdgeCosts(G, x..y, Dom::Real)` creates random edge weights of type Real within the range x..y.

Note: Already existing edge costs will be changed, too!

Examples

Example 1

Creating edge weights for a small cyclic graph. First, it has no specified weights:

```
G1 := Graph::createCircleGraph(5):
Graph::printGraphInformation(G1)
```

```
Vertices: [1, 2, 3, 4, 5]
Edges: [[1, 2], [2, 3], [3, 4], [4, 5], [5, 1]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [2], 2 = [3], 3 = [4], 4 = [5], 5 = [1]
```

```
Adjacency list (in): 1 = [5], 2 = [1], 3 = [2], 4 = [3], 5 = [4]
Graph is directed.
```

Now the costss are set within the range `-100..50` (your output may differ due random assignment):

```
G2 := Graph::createRandomEdgeCosts(G1, -100..50):
Graph::printGraphInformation(G2)
```

```
Vertices: [1, 2, 3, 4, 5]
Edges: [[1, 2], [2, 3], [3, 4], [4, 5], [5, 1]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: [1, 2] = 47, [2, 3] = -12, [3, 4] = 28, [4, 5] = 1, [5, 1] = -\
36 (other existing edges have costs zero)
Adjacency list (out): 1 = [2], 2 = [3], 3 = [4], 4 = [5], 5 = [1]
Adjacency list (in): 1 = [5], 2 = [1], 3 = [2], 4 = [3], 5 = [4]
Graph is directed.
```

If the costs should be of type Real it can be set optionally:

```
G2 := Graph::createRandomEdgeCosts(G1, -100..50, Dom::Real):
Graph::printGraphInformation(G2)
```

```
Vertices: [1, 2, 3, 4, 5]
Edges: [[1, 2], [2, 3], [3, 4], [4, 5], [5, 1]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: [1, 2] = -67.72964183, [2, 3] = -10.16896282, [3, 4] = -72.846\
84348, [4, 5] = -61.00518722, [5, 1] = 18.2662729 (other existing edges ha\
ve costs zero)
Adjacency list (out): 1 = [2], 2 = [3], 3 = [4], 4 = [5], 5 = [1]
Adjacency list (in): 1 = [5], 2 = [1], 3 = [2], 4 = [3], 5 = [4]
Graph is directed.
```

Parameters

G

A graph

r

A range

Return Values

Graph

Graph::createRandomGraph

Generates a random graph.

Syntax

```
Graph::createRandomGraph(VertexNr, EdgeNr, <Directed | Undirected>)
```

Description

`Graph::createRandomGraph` generates a random graph.

`Graph::createRandomGraph(VertexNr, EdgeNr)` generates a random graph with VertexNr vertices and EdgeNr edges.

Note: If the number EdgeNr is too great (i.e. $\text{EdgeNr} \geq \frac{\text{VertexNr}\,(\text{VertexNr}-1)}{2}$), a complete graph will be created.

`Graph::createRandomGraph(VertexNr, EdgeNr, Undirected)` generates a random graph with VertexNr vertices and 2 *EdgeNr* edges is created. This is due to the fact that no odd number of undirected edges could be created otherwise.

Examples

Example 1

The following graph was created `randomly`, meaning that your results will most probably differ:

```
G := Graph::createRandomGraph(5,6):
Graph::printGraphInformation(G)
```

```
Vertices: [1, 2, 3, 4, 5]
```

```
Edges: [[1, 4], [2, 1], [2, 4], [2, 5], [3, 1], [3, 4]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [4], 2 = [1, 4, 5], 3 = [1, 4], 4 = [], 5 = []
Adjacency list (in): 1 = [2, 3], 2 = [], 3 = [], 4 = [1, 2, 3], 5 = [2]
Graph is directed.
```

Example 2

The same number of vertices, but this time the edges are undirected (and therefore the number of Edges is (2 *EdgeNr*)). As you can clearly see, the edges differ from the edges created above:

```
G := Graph::createRandomGraph(5, 6, Undirected):
Graph::printGraphInformation(G)
```

```
Vertices: [1, 2, 3, 4, 5]
Edges: [[1, 4], [1, 5], [2, 3], [2, 4], [2, 5], [3, 2], [4, 1], [4, 2], [4\
, 5], [5, 1], [5, 2], [5, 4]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [4, 5], 2 = [3, 4, 5], 3 = [2], 4 = [1, 2, 5], 5\
 = [1, 2, 4]
Adjacency list (in): 1 = [4, 5], 2 = [3, 4, 5], 3 = [2], 4 = [1, 2, 5], 5 \
= [1, 2, 4]
Graph is undirected.
```

Example 3

If the number of edges to be created extends the possible limit ($\frac{\text{vertices}\,(\text{vertices}-1)}{2}$), a complete graph will be returned:

```
G := Graph::createRandomGraph(3, 6, Undirected):
Graph::printGraphInformation(G)
```

```
Warning: Cannot produce the required number of edges. Creating a complete graph instead
```

```
Vertices: [1, 2, 3]
Edges: [[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [2, 3], 2 = [1, 3], 3 = [1, 2]
Adjacency list (in): 1 = [2, 3], 2 = [1, 3], 3 = [1, 2]
Graph is undirected.
```

Parameters

`VertexNr`

Positive integer

`EdgeNr`

Positive integer

Options

`Directed`

If `Directed` is stated, a directed Graph is created `Default`

`Undirected`

If `Undirected` is stated, an undirected Graph is created.

Return Values

`Graph`

Graph::createRandomVertexWeights

Sets random weights to vertices

Syntax

```
Graph::createRandomVertexWeights(G, r, <Int | Real>)
```

Description

`Graph::createRandomVertexWeights(G, x..y)` creates random vertex weights of type Integer within the range x..y.

`Graph::createRandomVertexWeights(G, x..y, Real)` creates random vertex weights of type Real within the range x..y.

Note: Already existing vertex weights will be changed, too!

Examples

Example 1

Creating vertex weights for a small cyclic graph. First, it has no specified weights:

```
G1 := Graph::createCircleGraph(5):
Graph::printGraphInformation(G1)
```

```
Vertices: [1, 2, 3, 4, 5]
Edges: [[1, 2], [2, 3], [3, 4], [4, 5], [5, 1]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [2], 2 = [3], 3 = [4], 4 = [5], 5 = [1]
```

```
Adjacency list (in): 1 = [5], 2 = [1], 3 = [2], 4 = [3], 5 = [4]
Graph is directed.
```

Now the weights are set within the range `-100..50` (your output may differ due random assignment):

```
G2 := Graph::createRandomVertexWeights(G1, -100..50):
Graph::printGraphInformation(G2)
```

```
Vertices: [1, 2, 3, 4, 5]
Edges: [[1, 2], [2, 3], [3, 4], [4, 5], [5, 1]]
Vertex weights: 1 = 47, 2 = -12, 3 = 28, 4 = 1, 5 = -36 (other existing ve\
rtices have no weight)
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [2], 2 = [3], 3 = [4], 4 = [5], 5 = [1]
Adjacency list (in): 1 = [5], 2 = [1], 3 = [2], 4 = [3], 5 = [4]
Graph is directed.
```

If the weights should be of type Real it can be set optionally:

```
G2 := Graph::createRandomVertexWeights(G1, -100..50, Real):
Graph::printGraphInformation(G2)
```

```
Vertices: [1, 2, 3, 4, 5]
Edges: [[1, 2], [2, 3], [3, 4], [4, 5], [5, 1]]
Vertex weights: 1 = -67.72964183, 2 = -10.16896282, 3 = -72.84684348, 4 = \
-61.00518722, 5 = 18.2662729 (other existing vertices have no weight)
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [2], 2 = [3], 3 = [4], 4 = [5], 5 = [1]
Adjacency list (in): 1 = [5], 2 = [1], 3 = [2], 4 = [3], 5 = [4]
Graph is directed.
```

Parameters

G

A graph

`r`

A range

Options

`Int`

If stated, the weights are only of type integer. (Default)

`Real`

If stated, the weights are only of type real.

Return Values

Graph

Graph::depthFirstSearch

Makes a depth first Search in a graph.

Syntax

```
Graph::depthFirstSearch(G, <StartVertex = v>)
```

Description

`Graph::depthFirstSearch` traverses through a graph via depth first search. The output shows the first time of identification, the finishing time and the predecessor of each vertex. If a vertex is a single vertex with no predecessor its predecessor is *infinity*.

`Graph::depthFirstSearch(G, StartVertex=v)` traverses through a graph via depth first search starting from vertex `v`. The output shows the first time of identification, the finishing time and the predecessor of each vertex. If a vertex is a single vertex with no predecessor its predecessor is *infinity*.

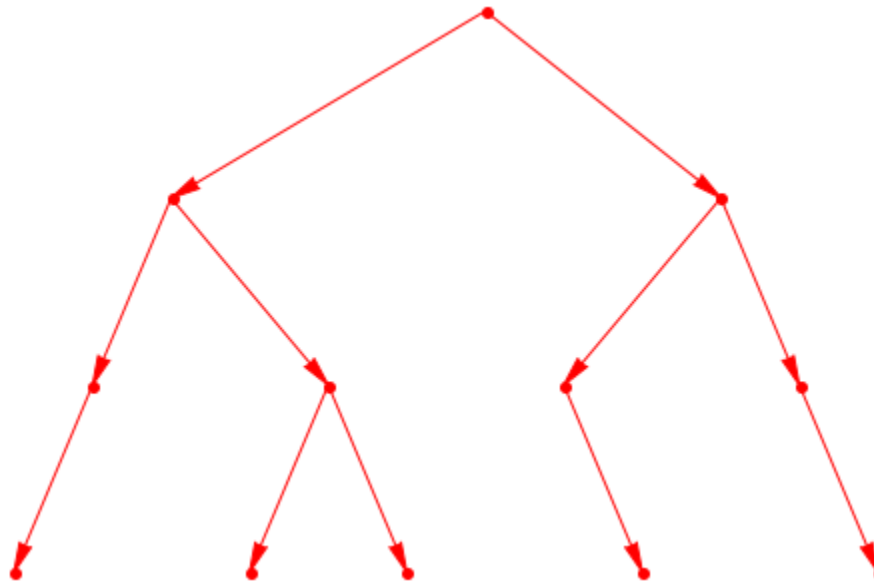
Examples

Example 1

A typical tree is created and drawn for a better understanding of the algorithm.

```
G := Graph([a, b, c, d, e, f, g, h, i, j, k, l],
           [[a, b], [a, c], [b, d], [b, e], [c, f], [c, g],
            [d, h], [e, i], [e, j], [f, k], [g, l]],
           Directed):
plot(
  Graph::plotGridGraph(G, VerticesPerLine = [12, 12, 12, 12],
    VertexOrder = [
  None, None, None, None, None, None,
  a,    None, None, None, None, None,
  None, None, b,    None, None, None,
```

```
  None, None, None, c,    None, None,
  None, d,    None, None, e,    None,
  None, f,    None, None, g,    None,
  h,    None, None, i,    None, j,
  None, None, k,    None, None, l
    ]
  )
)
```



Now we call `Graph::depthFirstSearch` to find out the starting times, the finishing times and the predecessors of each vertex:

```
Graph::depthFirstSearch(G)
```

a	1	a	24	a	a
b	2	b	13	b	a
c	14	c	23	c	a
d	3	d	6	d	b
e	7	e	12	e	b
f	15,	f	18,	f	c
g	19	g	22	g	c
h	4	h	5	h	d
i	8	i	9	i	e
j	10	j	11	j	e
k	16	k	17	k	f
l	20	l	21	l	g

Vertex a is dicovered first, then vertex b and so on. The table in the middle shows the finishing times. h for example has the finishing time of 5, meaning that vertices a, b, c, d and h itself were visited before it was recognized that h is a leaf (finishing time = starting time + 1). The right table shows the predecessor of every vertex. The backtacking from a single vertex is therefore really simple. a as the first vertex discovered in its component can not be backtracked any further.

Example 2

What happens now, if there exist a vertex that has no connection to any other vertex. The upper example is taken and a single vertex is added without changing anything else. Then a depth first search is invoked on the graph:

```
G := Graph([a, b, c, d, e, f, g, h, i, j, k, l],
           [[a, b], [a, c], [b, d], [b, e], [c, f], [c, g],
            [d, h], [e, i], [e, j], [f, k], [g, l]],
           Directed):

G2 := Graph::addVertices(G, [m]):
Graph::depthFirstSearch(G2, StartVertex = [a])
```

a	1	a	24	a	a
b	2	b	13	b	a
c	14	c	23	c	a
d	3	d	6	d	b
e	7	e	12	e	b
f	15	f	18	f	c
g	19	g	22	g	c
h	4	h	5	h	d
i	8	i	9	i	e
j	10	j	11	j	e
k	16	k	17	k	f
l	20	l	21	l	g
m	25	m	26	m	∞

The newly inserted vertex `m` has no predecessor. The predecessor holds therefore the value *infinity*.

Example 3

If we start somewhere in the graph without knowing the root of the DAG, the results are of course different:

```
G := Graph([a, b, c, d, e, f, g, h, i, j, k, l],
           [[a, b], [a, c], [b, d], [b, e], [c, f], [c, g],
            [d, h], [e, i], [e, j], [f, k], [g, l]],
           Directed):

Graph::depthFirstSearch(G, StartVertex = [c])
```

a	11	a	24	a	a
b	12	b	23	b	a
c	1	c	10	c	c
d	13	d	16	d	b
e	17	e	22	e	b
f	2	f	5	f	c
g	6	g	9	g	c
h	14	h	15	h	d
i	18	i	19	i	e
j	20	j	21	j	e
k	3	k	4	k	f
l	7	l	8	l	g

The predecessor of `c` is `c`, but if we look at the graph it should be `a`. This is nevertheless not quite correct. Breadth first search takes the given vertex and uses this as the root of the graph (no in-vertices!). This explains also why the next call shows a *infinity* as predecessor to l:

```
Graph::depthFirstSearch(G, StartVertex = [l])
```

a	3	a	24	a	a
b	4	b	15	b	a
c	16	c	23	c	a
d	5	d	8	d	b
e	9	e	14	e	b
f	17,	f	20,	f	c
g	21	g	22	g	c
h	6	h	7	h	d
i	10	i	11	i	e
j	12	j	13	j	e
k	18	k	19	k	f
l	1	l	2	l	∞

Parameters

`G`

`Graph`

`v`

List containing one vertex.

Options

`StartVertex`

Defines a vertex from which to start the depth first traversal.

Return Values

List containing three tables. The first table holds the first identification timestamp of each vertex, the second the finishing timestamp and the third the predecessor vertex.

Graph::getAdjacentEdgesEntering

Returns the incident edges.

Syntax

```
Graph::getAdjacentEdgesEntering(G, Vertex)
```

Description

`Graph::getAdjacentEdgesEntering(G, Vertex)` returns a list with vertices `v1..vn`, where `[v1, Vertex] .. [vn, Vertex]` are incident (incoming) Edges to `Vertex`.

Examples

Example 1

First, a complete graph is defined:

```
G1 := Graph::createCompleteGraph(5):
Graph::printGraphInformation(G1)
```

```
Vertices: [1, 2, 3, 4, 5]
Edges: [[1, 2], [1, 3], [1, 4], [1, 5], [2, 1], [2, 3], [2, 4], [2, 5], [3\
, 1], [3, 2], [3, 4], [3, 5], [4, 1], [4, 2], [4, 3], [4, 5], [5, 1], [5, \
2], [5, 3], [5, 4]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [2, 3, 4, 5], 2 = [1, 3, 4, 5], 3 = [1, 2, 4, 5]\
, 4 = [1, 2, 3, 5], 5 = [1, 2, 3, 4]
Adjacency list (in): 1 = [2, 3, 4, 5], 2 = [1, 3, 4, 5], 3 = [1, 2, 4, 5],\
 4 = [1, 2, 3, 5], 5 = [1, 2, 3, 4]
Graph is undirected.
```

Now we get the vertices that form all incident edges `[2, 1] .. [5, 1]`:

```
Graph::getAdjacentEdgesEntering(G1, [1])
```

```
[2, 3, 4, 5]
```

Now we get the vertices that form all incident edges `[1, 5] .. [4, 5]`:

```
Graph::getAdjacentEdgesEntering(G1, [5])
```

```
[1, 2, 3, 4]
```

Parameters

`G`

A graph

`Vertex`

One vertex of `G`.

Return Values

List

Graph::getAdjacentEdgesLeaving

Returns the adjacent edges.

Syntax

```
Graph::getAdjacentEdgesLeaving(G, Vertex)
```

Description

`Graph::getAdjacentEdgesLeaving(G, Vertex)` returns a list with vertices `v1..vn`, where `[Vertex, v1] .. [Vertex, vn]` are adjacent (outgoing) Edges to `Vertex`.

Examples

Example 1

First, a complete graph is defined:

```
G1 := Graph::createCompleteGraph(5):
Graph::printGraphInformation(G1)
```

```
Vertices: [1, 2, 3, 4, 5]
Edges: [[1, 2], [1, 3], [1, 4], [1, 5], [2, 1], [2, 3], [2, 4], [2, 5], [3\
, 1], [3, 2], [3, 4], [3, 5], [4, 1], [4, 2], [4, 3], [4, 5], [5, 1], [5, \
2], [5, 3], [5, 4]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [2, 3, 4, 5], 2 = [1, 3, 4, 5], 3 = [1, 2, 4, 5]\
, 4 = [1, 2, 3, 5], 5 = [1, 2, 3, 4]
Adjacency list (in): 1 = [2, 3, 4, 5], 2 = [1, 3, 4, 5], 3 = [1, 2, 4, 5],\
 4 = [1, 2, 3, 5], 5 = [1, 2, 3, 4]
Graph is undirected.
```

Now we get the vertices that form all adjacent edges `[1, 2] .. [1, 5]`:

```
Graph::getAdjacentEdgesLeaving(G1, [1])
```

```
[2, 3, 4, 5]
```

Now we get the vertices that form all adjacent edges `[2, 1] .. [2, 5]`:

```
Graph::getAdjacentEdgesLeaving(G1, [2])
```

```
[1, 3, 4, 5]
```

Parameters

`G`

A graph

`Vertex`

One vertex of `G`.

Return Values

List

Graph::getBestAdjacentEdge

Returns the "best" incident edges.

Syntax

```
Graph::getBestAdjacentEdge(G, Vertex, Vertices, <Min | Max>, <Weights | Costs>)
```

Description

`Graph::getBestAdjacentEdge(G, Vertex)` returns the best incident edge according to specified attributes.

`Graph::getBestAdjacentEdge(G, Vertex, Vertices)` returns a vertex `v` out of `Vertices`. The best edge is (Vertex, v) according to the specifications.

Examples

Example 1

Let us create a graph and find out the edge with the least weight:

```
G1 := Graph([1, a, 3, 4], [[1, a], [1, 3], [1, 4]],
            EdgeWeights = [10, 20, 30],
            EdgeCosts = [30, 20, 10]):
Graph::getBestAdjacentEdge(G1, [1], Graph::getVertices(G1)),
Graph::getBestAdjacentEdge(G1, [1], Graph::getVertices(G1),
                                              Min, Weights)
```

$$a, a$$

The result shows that edge [1, a] has the least weight. It also shows that `Min` and `Weights` are the defaults if omitted. Next, we want to know the edge with maximum weight:

```
Graph::getBestAdjacentEdge(G1, [1], Graph::getVertices(G1), Max)
```

```
    4
```

The vertex with maximum weight is edge [1,4]. Now we have a look at the costs. The minimum cost edge can be found with:

```
Graph::getBestAdjacentEdge(G1, [1], Graph::getVertices(G1), Costs)
```

```
    4
```

So the vertex with maximum weight is also the edge with minimum costs. Finally let us search for the edge with maximum costs:

```
Graph::getBestAdjacentEdge(G1, [1], Graph::getVertices(G1),
                                                  Costs, Max)
```

```
    a
```

Parameters

`G`

A graph

`Vertex`

One vertex of `G`.

`Vertices`

Vertices in `G`.

Options

`Min`

If stated, the edge with the minimum attribute will be found. (Default)

`Max`

If stated, the edge with the maximum attribute will be found.

`Weights`

If stated, edge weights will be used for comparison. (Default)

`Costs`

If stated, edge costs will be used for comparison.

Return Values

Vertex

Graph::getEdgeCosts

Returns a table with the edge costs.

Syntax

```
Graph::getEdgeCosts(G)
```

Description

`Graph::getEdgeCosts(G)` returns a table with the edge costs of the graph `G`. Thus `Graph::getEdgeCosts(G)` returns the costs of all edges in `G`.

Note: Costs will most probably only be defined, if transportation problems occur.

Note: If FAIL is returned, no costs werde defined (this way both, network and graph algorithms handle this situation correct.)

Examples

Example 1

First lets define a graph without edge costs:

```
G1 := Graph::createCircleGraph(3):
Graph::getEdgeCosts(G1)
```

```
    FAIL
```

`FAIL` was returned, because no edge costs were defined.

```
Graph::getEdges(G1);
G1 := Graph::setEdgeCosts(G1, [[1, 2], [3, 1]], [5, 1/2]):
Graph::getEdgeCosts(G1)
```

```
[[1, 2], [2, 3], [3, 1]]
```

[1, 2]	5
[3, 1]	$\frac{1}{2}$

The first output shows all the edges and the second one the assigned edge costs.

Parameters

G

A graph

Return Values

Table

Graph::getEdgeDescriptions

Returns a table with the edge descriptions.

Syntax

```
Graph::getEdgeDescriptions(G)
```

Description

`Graph::getEdgeDescriptions(G)` returns a table with the edge descriptions of the graph `G`. Thus `Graph::getEdgeDescriptions(G)` returns the weight of all edges in `G`.

Note: Descriptions will most probably only be defined, if transportation problems occur.

Note: If FAIL is returned, no descriptions were defined (this way both, network and graph algorithms handle this situation correct.)

Examples

Example 1

First lets define a graph without edge descriptions:

```
G1 := Graph::createCircleGraph(3):
Graph::getEdgeDescriptions(G1)
```

```
    FAIL
```

`FAIL` was returned, because no edge descriptions were defined.

```
Graph::getEdges(G1);
G1 := Graph::setEdgeDescriptions(G1, [[1, 2], [3, 1]],
                                 ["Shortcut", "Highway 66"]):
```

```
Graph::getEdgeDescriptions(G1)
```

```
[[1, 2], [2, 3], [3, 1]]
```

[1, 2]	"Shortcut"
[3, 1]	"Highway 66"

The first output shows all the edges and the second one the assigned edge descriptions.

Parameters

G

A graph

Return Values

Table

Graph::getEdges

Returns a list with all edges

Syntax

```
Graph::getEdges(G)
```

Description

`Graph::getEdges(G)` returns a list containing all edges of the graph `G`. Each edge is represented by a list containing the two connected vertices.

Examples

Example 1

`Graph::getEdges` only returns the edges, without their capacities:

```
G1 := Graph::createCircleGraph([v1, v2, v3, v4]):
Graph::getEdges(G1)
```

```
[[v1, v2], [v2, v3], [v3, v4], [v4, v1]]
```

```
G2 := Graph::createCompleteGraph(3):
Graph::getEdges(G2)
```

```
[[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]
```

Parameters

`G`

A Graph

Return Values

List of all edges, a list of lists

Graph::getEdgesEntering

Returns the incoming edges

Syntax

```
Graph::getEdgesEntering(G)
```

Description

`Graph::getEdgesEntering(G)` returns a table with the adjacency lists for incident (incoming) edges. Thus `Graph::getEdgesEntering(G)` returns a table containing all those vertices `w` for which there is an edge `[w, v]` in `G`.

Examples

Example 1

A small directed graph is created to show the incoming (incident) edges:

```
V := [1, 2, 3, 4, 5]:
Ed := [[1, 2], [1, 3], [2, 3], [2, 4], [3, 4], [3, 5], [4, 5]]:
G1 := Graph(V, Ed, Directed):
Graph::getEdgesEntering(G1)
```

1	[]
2	[1]
3	[1, 2]
4	[2, 3]
5	[3, 4]

In an undirected graph the output could look like this:

```
G1 := Graph::createCompleteGraph(5):
```

```
Graph::getEdgesEntering(G1)
```

1	[2, 3, 4, 5]
2	[1, 3, 4, 5]
3	[1, 2, 4, 5]
4	[1, 2, 3, 5]
5	[1, 2, 3, 4]

Parameters

G

A graph

Return Values

Table

Graph::getEdgesLeaving

Returns the outgoing edges

Syntax

```
Graph::getEdgesLeaving(G)
```

Description

`Graph::getEdgesLeaving(G)` returns a table with the adjacency lists for adjacent (outgoing) edges. Thus `Graph::getEdgesLeaving(G)` returns a table containing all those vertices `w` for which there is an edge `[v,w]` in `G`.

Examples

Example 1

A small directed graph is created to show the outgoing (adjacent) edges:

```
V := [1, 2, 3, 4, 5]:
Ed := [[1, 2], [1, 3], [2, 3], [2, 4], [3, 4], [3, 5], [4, 5]]:
G1 := Graph(V, Ed, Directed):
Graph::getEdgesLeaving(G1)
```

1	[2, 3]
2	[3, 4]
3	[4, 5]
4	[5]
5	[]

In an undirected graph the output could look like this:

```
G1 := Graph::createCompleteGraph(5):
```

```
Graph::getEdgesLeaving(G1)
```

1	[2, 3, 4, 5]
2	[1, 3, 4, 5]
3	[1, 2, 4, 5]
4	[1, 2, 3, 5]
5	[1, 2, 3, 4]

Parameters

G

A graph

Return Values

Table

Graph::getEdgeNumber

Returns the number of edges.

Syntax

```
Graph::getEdgeNumber(G)
```

Description

`Graph::getEdgeNumber(G)` returns a number representing the number of edges in `G`.

Examples

Example 1

Let us create a graph and find out the number of edges:

```
G1 := Graph([1, a, 3, 4], [[1, a], [1, 3], [1, 4]]):
Graph::getEdgeNumber(G1)
```

```
6
```

We know that a complete graph consists of $|Vertices|^2 - |Vertices|$ edges:

```
G2 := Graph::createCompleteGraph(4):
Graph::getEdgeNumber(G2)
```

```
12
```

Parameters

`G`

A graph

Return Values

Number

Graph::getEdgeWeights

Returns a table with the edge weights.

Syntax

```
Graph::getEdgeWeights(G)
```

Description

`Graph::getEdgeWeights(G)` returns a table with the edge weights of the graph `G`. Thus `Graph::getEdgeWeights(G)` returns the weight of all edges in `G`.

Note: Weights will most probably only be defined, if transportation problems occur.

Note: If FAIL is returned, no weights were defined (this way both, network and graph algorithms handle this situation correct.)

Examples

Example 1

First lets define a graph without edge weights:

```
G1 := Graph::createCircleGraph(3):
Graph::getEdgeWeights(G1)
```

```
    FAIL
```

`FAIL` was returned, because no edge weights were defined.

```
Graph::getEdges(G1);
G1 := Graph::setEdgeWeights(G1, [[1, 2], [3, 1]], [5, 1/2]):
Graph::getEdgeWeights(G1)
```

```
[[1, 2], [2, 3], [3, 1]]
```

[1, 2]	5
[3, 1]	$\frac{1}{2}$

The first output shows all the edges and the second one the assigned edge weights.

Parameters

G

A graph

Return Values

Table

Graph::getSubGraph

Returns a subgraph.

Syntax

```
Graph::getSubGraph(G, Vertex)
```

Description

`Graph::getSubGraph(G, Vertex)` returns a subgraph according to the specified vertices.

`Graph::getSubGraph(G)` returns a graph that only holds the specified vertices and the belonging edges.

Examples

Example 1

First, a complete graph is defined with some additional settings:

```
G1 := Graph::createCompleteGraph(5):
G1 := Graph::setEdgeWeights(G1, [[1,2]], [20]):
G1 := Graph::setEdgeCosts(G1, [[1, 2]], [20]):
G1 := Graph::setEdgeDescriptions(G1, [[1, 2]], ["Shortcut"]):
Graph::printGraphInformation(G1)
```

```
Vertices: [1, 2, 3, 4, 5]
Edges: [[1, 2], [1, 3], [1, 4], [1, 5], [2, 1], [2, 3], [2, 4], [2, 5], [3\
, 1], [3, 2], [3, 4], [3, 5], [4, 1], [4, 2], [4, 3], [4, 5], [5, 1], [5, \
2], [5, 3], [5, 4]]
Vertex weights: no vertex weights.
Edge descriptions: [1, 2] = "Shortcut", [2, 1] = "Shortcut"
Edge weights: [1, 2] = 20, [2, 1] = 20 (other existing edges have no weigh\
t)
Edge costs: [1, 2] = 20, [2, 1] = 20 (other existing edges have costs zero)
```

```
Adjacency list (out): 1 = [2, 3, 4, 5], 2 = [1, 3, 4, 5], 3 = [1, 2, 4, 5]\
, 4 = [1, 2, 3, 5], 5 = [1, 2, 3, 4]
Adjacency list (in): 1 = [2, 3, 4, 5], 2 = [1, 3, 4, 5], 3 = [1, 2, 4, 5],\
 4 = [1, 2, 3, 5], 5 = [1, 2, 3, 4]
Graph is undirected.
```

Now we get the subgraph for the vertices `1,2,4`:

```
G2 := Graph::getSubGraph(G1, [1, 2, 4]):
Graph::printGraphInformation(G2)
```

```
Vertices: [1, 2, 4]
Edges: [[1, 2], [1, 4], [2, 1], [2, 4], [4, 1], [4, 2]]
Vertex weights: no vertex weights.
Edge descriptions: [1, 2] = "Shortcut", [2, 1] = "Shortcut"
Edge weights: [1, 2] = 20, [2, 1] = 20 (other existing edges have no weigh\
t)
Edge costs: [1, 2] = 20, [2, 1] = 20 (other existing edges have costs zero)
Adjacency list (out): 1 = [2, 4], 2 = [1, 4], 4 = [1, 2]
Adjacency list (in): 1 = [2, 4], 2 = [1, 4], 4 = [1, 2]
Graph is undirected.
```

The subgraph for the vertices `1,3,4` looks like:

```
G2 := Graph::getSubGraph(G1, [1, 3, 4]):
Graph::printGraphInformation(G2)
```

```
Vertices: [1, 3, 4]
Edges: [[1, 3], [1, 4], [3, 1], [3, 4], [4, 1], [4, 3]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [3, 4], 3 = [1, 4], 4 = [1, 3]
Adjacency list (in): 1 = [3, 4], 3 = [1, 4], 4 = [1, 3]
Graph is undirected.
```

Parameters

`G`

A graph

`Vertex`

A list containing one or more vertices of `G`.

Return Values

Graph

Graph::getVertexNumber

Returns the number of vertices.

Syntax

```
Graph::getVertexNumber(G)
```

Description

`Graph::getVertexNumber(G)` returns a number representing the number of vertices in `G`.

Examples

Example 1

An example with 100 vertices:

```
G1 := Graph::createCompleteGraph(100):
Graph::getVertexNumber(G1)
```

```
      100
```

Parameters

`G`

A graph

Return Values

Number

Graph::getVertexWeights

Returns a table with the vertex weights.

Syntax

```
Graph::getVertexWeights(G)
```

Description

`Graph::getVertexWeights(G)` returns a table with the vertex weights of the graph `G`. Thus `Graph::getVertexWeights(G)` returns the weight of all vertices in `G`.

Note: If FAIL is returned, no weights were defined (this way both, network and graph algorithms handle this situation correct.)

Examples

Example 1

First lets define a graph without vertex weights:

```
G1 := Graph::createCircleGraph(3):
Graph::getVertexWeights(G1)
```

```
FAIL
```

`FAIL` was returned, because no vertex weights were defined.

```
Graph::getVertices(G1);
G1 := Graph::setVertexWeights(G1, [1, 3], [5, 1/2]):
Graph::getVertexWeights(G1)
```

```
[1, 2, 3]
```

1	5
3	$\frac{1}{2}$

The first output shows all the vertices and the second one the assigned vertex weights.

Parameters

G

A graph

Return Values

Table

Graph::getVertices

Returns a list with all vertices

Syntax

```
Graph::getVertices(G)
```

Description

`Graph::getVertices(G)` returns the list of all vertices of the Graph `G`.

Examples

Example 1

A small creation of two different graphs and the output `getVertices` generates:

```
G1 := Graph::createCompleteGraph(10):
Graph::getVertices(G1)
```

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

```
G2 := Graph::createCircleGraph([x.i $ i = 1..12]):
Graph::getVertices(G2)
```

$$[x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12]$$

Parameters

`G`

A graph

Return Values

List

Graph::inDegree

Returns the indegree of one or more vertices.

Syntax

```
Graph::inDegree(G, <Vertex>)
```

Description

`Graph::inDegree(G, Vertex)` returns the indegree of the vertex `Vertex` in the Graph `G`, i.e., the number of edges `[u,Vertex]`.

`Graph::inDegree(G, [v1, v2, ..., vn])` returns a table in which the keys are `v1, v2, ..., vn` and the corresponding values are the indegrees.

`Graph::inDegree(G)` returns a table in which each node of `G` is mapped to its indegree. `Graph::inDegree(G)` is equivalent to `Graph::inDegree(G, Graph::getVertices(G))`.

Examples

Example 1

In a complete graph of n vertices, each vertex has indegree n - 1:

```
G := Graph::createCompleteGraph(5):
Graph::inDegree(G, [2, 4, 5]), Graph::inDegree(G),
Graph::inDegree(G, Graph::getVertices(G))
```

2 | 4	1 | 4
4 | 4	2 | 4
5 | 4	3 | 4
	4 | 4
	5 | 4

1 | 4
2 | 4
3 | 4
4 | 4
5 | 4

The first table shows what happens, if some vertices are specified. The second and third table return all indegrees, but with two different calls (the second is redundant).

Example 2

Remember that also only one vertex needs to be specified as a list!

```
G := Graph::createCompleteGraph(5):
Graph::inDegree(G, [2])
```

2	4

Parameters

`G`

A Graph

`Vertex`

A list containing one or more vertices.

Return Values

Table containing all the indegrees of the specified vertices.

Graph::isConnected

Finds out if the graph is connected

Syntax

```
Graph::isConnected(G)
```

Description

`Graph::isConnected(G)` returns `TRUE` if `G` is connected, `FALSE` otherwise.

Examples

Example 1

A circle graph is made to create a connected Graph:

```
G1 := Graph::createCircleGraph(3):
Graph::isConnected(G1)
```

```
    TRUE
```

After adding a single vertex to the graph, it is not connected any more:

```
G2 := Graph::addVertices(G1, [4]):
Graph::isConnected(G2)
```

```
    FALSE
```

Parameters

`G`

A graph

Return Values

TRUE or FALSE

Graph::isDirected

Finds out if the graph is directed

Syntax

```
Graph::isDirected(G)
```

Description

`Graph::isDirected(G)` returns `TRUE` if `G` is directed, `FALSE` otherwise.

Examples

Example 1

A circle graph is made to create a directed Graph:

```
G1 := Graph::createCircleGraph(3):
Graph::isDirected(G1)
```

```
TRUE
```

Example 2

Now a complete graph is created in order to get an undirected graph:

```
G1 := Graph::createCompleteGraph(3):
Graph::isDirected(G1)
```

```
FALSE
```

Parameters

G

A graph

Return Values

TRUE or FALSE

Graph::isEdge

Finds out if the edges exists

Syntax

```
Graph::isEdge(G, Edge)
```

Description

`Graph::isEdge(G)` returns `TRUE` if `ALL` specified edges exist in `G`, `FALSE` otherwise.

Examples

Example 1

A circle graph is made to create a directed Graph:

```
G1 := Graph::createCircleGraph(3):
Graph::printGraphInformation(G1)
```

```
Vertices: [1, 2, 3]
Edges: [[1, 2], [2, 3], [3, 1]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [2], 2 = [3], 3 = [1]
Adjacency list (in): 1 = [3], 2 = [1], 3 = [2]
Graph is directed.
```

First let us check for an existing single edge:

```
Graph::isEdge(G1, [[1, 2]] )
```

```
    TRUE
```

Now we check if several edges exist:

```
Graph::isEdge(G1, [[1, 2], [2, 3]] )
```

```
    TRUE
```

What about a non existing edge?

```
Graph::isEdge(G1, [[3, 2]] )
```

```
    FALSE
```

Finally a list of some existing and non existing edges is checked:

```
Graph::isEdge(G1, [[1, 2], [2, 3], [3, 2]])
```

```
    FALSE
```

Parameters

`G`

A graph

`Edge`

A list containing one or more edges

Return Values

TRUE or FALSE

Graph::isVertex

Finds out if special vertices exist in the Graph

Syntax

```
Graph::isVertex(G, Vertex)
```

Description

`Graph::isVertex(G)` returns `TRUE` if `ALL` specified vertices exist in `G`, `FALSE` otherwise.

Examples

Example 1

A circle graph is made to create a directed Graph:

```
G1 := Graph::createCircleGraph(3):
Graph::printGraphInformation(G1)
```

```
Vertices: [1, 2, 3]
Edges: [[1, 2], [2, 3], [3, 1]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [2], 2 = [3], 3 = [1]
Adjacency list (in): 1 = [3], 2 = [1], 3 = [2]
Graph is directed.
```

First, let us check for an existing vertex:

```
Graph::isVertex(G1, [1])
```

```
TRUE
```

Now we check if several vertices exist:

```
Graph::isVertex(G1, [1, 2])
```

```
    TRUE
```

What about a non existing vertex?

```
Graph::isVertex(G1, [4])
```

```
    FALSE
```

Finally a list of some existing and non existing vertices is checked:

```
Graph::isVertex(G1, [1, 2, 4])
```

```
    FALSE
```

Parameters

`G`

A graph

`Vertex`

A list containing one or more vertices

Return Values

TRUE or FALSE

Graph::longestPath

Longest paths from one single node

Syntax

```
Graph::longestPath(G, v, <w>, <Length>, <Path>)
```

Description

`Graph::longestPath(G, v)` returns a table with the length of longest paths from `v` to all other nodes in the Graph with respect to the edge weight.

`Graph::longestPath(G, v, w)` returns the length of a longest path from `v` to `w`.

If the optional argument `Path` is given, a table with longest paths is returned. If both `Length` and `Path` are given, then both the length of the longest paths and the paths are returned. Paths are given as lists of nodes in reverse order.

If `Path` is not given, the option `Length` has no effect.

Note: The Graph `G` must be directed and should not contain cycles.

Examples

Example 1

We construct a Graph and try a few calls to `Graph::longestPath`:

```
V  := [1, 2, 3, 4, 5]:
Ed := [[1, 2], [1, 3], [2, 3], [2, 4], [3, 4], [3, 5], [4, 5]]:
Ew := [7, 6, 5, 4, 2, 2, 1]:
G  := Graph(V, Ed, EdgeWeights = Ew, Directed):
Graph::longestPath(G, 1)
```

1	0
2	7
3	12
4	14
5	15

```
Graph::longestPath(G, 1, Path)
```

2	[2, 1]
3	[3, 2, 1]
4	[4, 3, 2, 1]
5	[5, 4, 3, 2, 1]

Parameters

`G`

A Graph

`v, w`

Vertices in `G`

Options

`Length`

Return a table with the lengths of shortest paths

`Path`

Return a table with the paths themselves

Return Values

Table, an integer or a list of nodes

Algorithms

The implemented algorithm is a variation of the algorithm of Bellman.

Graph::maxFlow

Computes a maximal flow through a graph

Syntax

```
Graph::maxFlow(G, s, t)
```

Description

`Graph::maxFlow(G,s,t)` computes a maximal flow from `s` to `t` in `G` with respect to the edge capacities. `s` and `t` must be nodes in `G`.

`Graph::maxFlow(G,s,t)` returns a sequence containing the flow value, that is the inflow of `s`, which equals the outflow of `s`, and the flow itself in form of table `tbl` with the flow from vertex `v` to vertex `w` is `tbl[[v,w]]`.

Examples

Example 1

In the complete Graph with four vertices and default capacities of 1, the maximum flow from one vertex to another one consists of sending one unit through each of the remaining vertices and one directly, which makes three units altogether:

```
G1 := Graph::createCompleteGraph(4):
Graph::maxFlow(G1, [1], [4])
```

	[1, 2]	1
	[2, 1]	0
	[1, 3]	1
	[3, 1]	0
	[1, 4]	1
3,	[2, 3]	0
	[3, 2]	0
	[4, 1]	0
	[2, 4]	1
	[4, 2]	0
	[3, 4]	1
	[4, 3]	0

Example 2

As a more complex example, the following graph shows that this function also finds flows through multiple edges, unlike `Graph::admissibleFlow`, which only works on completely described flows:

```
V := [1, 2, 3, s, t]:
Edge := [[s, 1], [t, 2], [1, 2], [1, 3], [2, 3], [3, t]]:
up := [5, 5, 2, 6, 6, 1]:
G2 := Graph(V, Edge, EdgeCosts = up, Directed):
Graph::maxFlow(G2, [s], [t])
```

	[1, 2]	0
	[1, 3]	1
1,	[2, 3]	0
	[*s*, 1]	1
	[*t*, 2]	0
	[3, *t*]	1

Parameters

`G`

Graph

`s, t`

Expressions (vertices in `G`)

Return Values

List, containing a number and a table

Algorithms

The implemented algorithm is the preflow-push algorithm of Goldberg &Tarjan with the FIFO selection strategy and an exact distance labeling ("A new approach to the maximum-flow problem", Journal of the ACM 35(4), 1988).

The running time is $O(n^3)$, where n is the number of vertices in the Graph.

Graph::minCost

Computes a minimal cost flow

Syntax

```
Graph::minCost(G)
```

Description

`Graph::minCost(G)` computes a minimal cost flow in `G` with respect to the edge capacities, the edge weights and the vertex weights of `G`.

The vertex weights are interpreted as supply and demand. The edge weights give restrictions for the flow on every edge. The edge costs are the cost for one unit flow over an edge.

The algorithm computes a flow, if there is any, which is possible and satisfactory, i.e., it is within the supply and demand range, which respects the capacities and which has minimal cost.

Examples

Example 1

We construct a Graph with five vertices and seven edges. One of the vertices is a pure source (1), another one is a pure sink (5). No other vertices supply or demand any goods, they only serve as transportation junctions:

```
V  := [1, 2, 3, 4, 5]:
Vw := [25, 0, 0, 0, -25]:
edges := [[1, 2], [1, 3], [2, 3], [2, 4], [3, 4], [3, 5], [4, 5]]:
Ec := [7, 6, 5, 4, 2, 2, 1]:
Ew := [30, 20, 25, 10, 20, 25, 20]:
G1 := Graph(V, edges, EdgeCosts = Ec, EdgeWeights = Ew,
            VertexWeights = Vw, Directed):
Graph::minCost(G1)
```

[1, 2]	5
[1, 3]	20
[2, 3]	0
[2, 4]	5
[3, 4]	0
[3, 5]	20
[4, 5]	5

,

[1, 2]	35
[1, 3]	120
[2, 3]	0
[2, 4]	20
[3, 4]	0
[3, 5]	40
[4, 5]	5

, 220,

1	12
2	5
3	2
4	1
5	0

All 25 units could be transported from vertex 1 to vertex 5, for a total cost of 220. The cost for each edge can be found in the first table, the accumulated costs in the second and the last table holds the dual prices. For example 6 units flow over edge [1, 3] since 6 20 = 120 and 7 units flow over edge [1, 2] since 7 5 = 35.

Parameters

G

Graph

Return Values

Sequence, consisting of three tables and a number. The first table holds the amount flowing over the edge, the second the accumulated costs for each used edge and the number is the sum of all edge-costs for the flow. The last table holds the dual prices for each vertex.

Algorithms

The implemented algorithm is the relaxation algorithm due to Bertsekas (taken from Bertsekas, “Linear Network Optimization”, MIT Press, Cambridge(Mass.)-London, 1991) which is known to be one of the fastest algorithms in practice.

Graph::minCut

Computes a minimal cut

Syntax

```
Graph::minCut(G, q, s)
```

Description

`Graph::minCut(G,q,s)` computes a minimal cut in `G` that separates `q` from `s`, i.e., a subset `T` of the set `S` of edges of `G` such that every path from `q` to`s` contains at least one edge in `T`. The cut is minimal with respect to the capacities of the edges.

`Graph::minCut(G,q,s)` returns a sequence consisting of the cut value (the sum of the edge weights of the cut edges) and a list with the edges of the cut.

Note that `q` is separated from `s`, not vice versa.

Examples

Example 1

In a complete graph, a vertex can be separated from another one only by cutting all edges starting at the first vertex:

```
G1 := Graph::createCompleteGraph(4):
Graph::minCut(G1, [1], [4])
```

```
3, [[1, 2], [1, 3], [1, 4]]
```

Example 2

In the following example, the edge from vertex `q` to vertex `1` could have been used as well, but its edge capacity is higher than that of the edge used, so the minimality condition precludes this choice:

```
V := [1, 2, 3, q, s]:
Edge := [[q, 1], [1, 2], [1, 3], [2, 3], [3, s]]:
up := [5, 2, 6, 6, 1]:
G2 := Graph(V, Edge, EdgeWeights = up, Directed):
Graph::minCut(G2, [q], [s])
```

$$1, [[3, s]]$$

There is no path from vertex `s` to vertex `q` (or any other vertex of the Graph), so no cut is necessary to separate `s` from `q`:

```
Graph::minCut(G2, [s], [q])
```

$$0, []$$

Parameters

`q, s`

Vertices that have to be defined within G

`G`

Graph

Return Values

Sequence, consisting of the “cut value” and a list of edges cut

Graph::minimumSpanningTree

Creates a MST

Syntax

```
Graph::minimumSpanningTree(G, <SearchFor = Weights | Costs>, <ReturnAsTable>)
```

Description

`Graph::minimumSpanningTree(G)` creates a minimum spanning tree of `Graph` `G` according to the weights of the edges and returns a Graph consisting only of them. The same result would be achieved using `Graph::minimumSpanningTree(G, SearchFor = Weights)`

`Graph::minimumSpanningTree(G, SearchFor = Costs)` creates a minimum spanning tree according to the costs of the edges and returns a Graph consisting only of them.

`Graph::minimumSpanningTree(G, ReturnAsTable)` creates a minimum spanning tree according to the weights of the edges and returns a list with two objects. The first is a table consisting of the used edges and their weights. The second object is a number containing the sum of all the edge weights.(The same result can be achieved using: `Graph::minimumSpanningTree(G, SearchFor=Weights, ReturnAsTable)`.)

`Graph::minimumSpanningTree(G, SearchFor=Costs, ReturnAsTable)` creates a minimum spanning tree according to the costs of the edges and returns a list with two objects. The first is a table consisting of the used edges and their costs. The second object is a number containing the sum of all the edge costs.

Examples

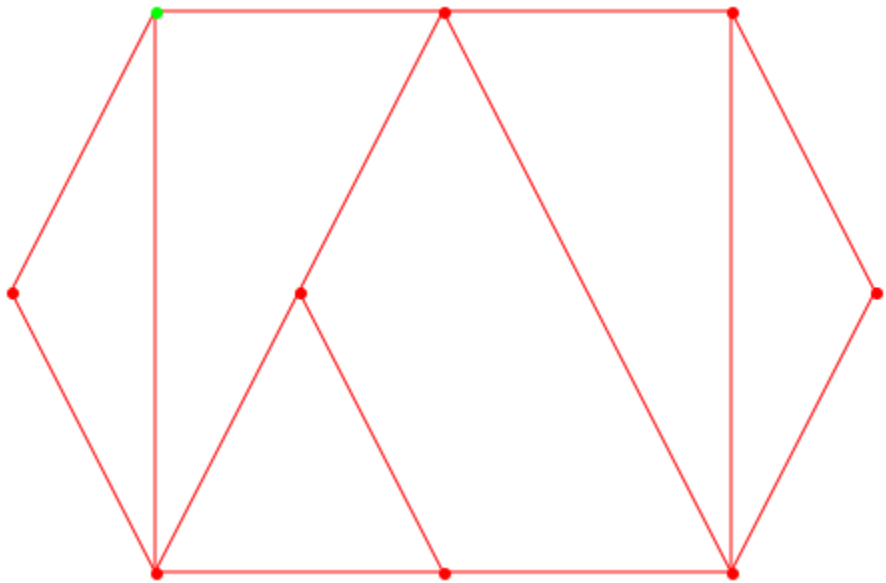
Example 1

The following graph `G` will be used throughout all the examples. For details on the format of `G`, see `Graph`. (Have a look at the edge [*c*, *f*]. This edge is responsible for the different outputs whether `Costs` or `Weights` was chosen.)

```
G := Graph([a, b, c, d, e, f, g, h, i],
           [[a, b], [a, h], [b, h], [b, c], [c, d], [d, f], [d, e],
            [f, e], [h, g], [g, f], [c, i], [h, i], [g, i], [c, f]],
         EdgeWeights = [4, 8, 11, 8, 7, 14, 9, 10, 1, 2, 2, 7, 6,  4],
         EdgeCosts   = [4, 8, 11, 8, 7, 14, 9, 10, 1, 2, 2, 7, 6, 12]):
```

We will plot this graph and all graphs derived from it using `Graph::plotGridGraph` with the following options:

```
plotOptions :=
  VerticesPerLine=7,
  VertexOrder = [
     None, b,    None, c,    None, d,    None,
     a,    None, i,    None, None, None, e,
     None, h,    None, g,    None, f,    None]:

plot(Graph::plotGridGraph(G, plotOptions))
```



Now we use this Graph to create a minimum spanning tree according to the weights of the edges and have a look which edges were used:

```
Graph::minimumSpanningTree(G, SearchFor = Weights, ReturnAsTable),
```

```
Graph::minimumSpanningTree(G, ReturnAsTable)
```

σ_1, σ_1

where

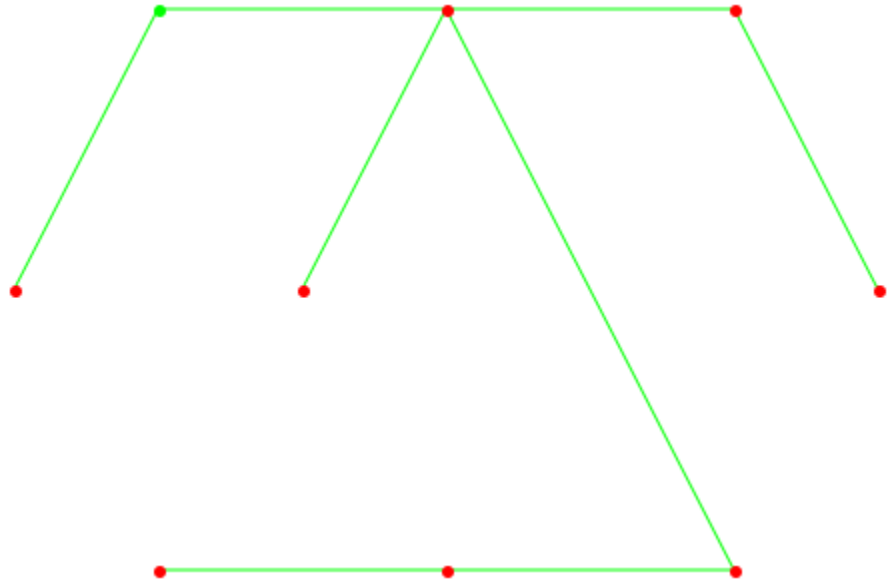
$\sigma_1 =$

$[a, b]$	4
$[b, a]$	4
$[b, c]$	8
$[c, b]$	8
$[c, d]$	7
$[d, c]$	7
$[c, f]$	4
$[d, e]$	9, 37
$[e, d]$	9
$[f, c]$	4
$[c, i]$	2
$[i, c]$	2
$[f, g]$	2
$[g, f]$	2
$[g, h]$	1
$[h, g]$	1

Both calls return exactly the same tables. That was expected and just to show that it is of no importance if the additional `SearchFor=Weights` is omitted.

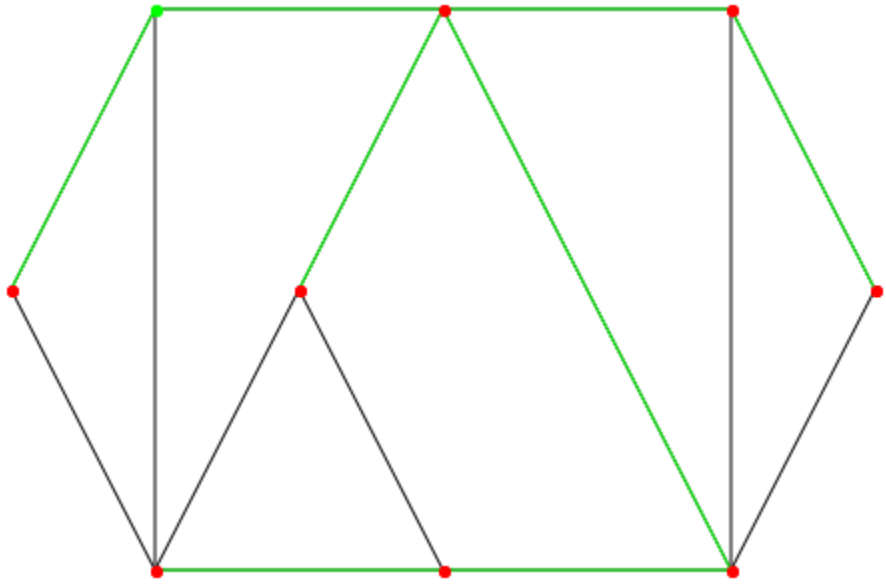
Now we want to get the minimum spanning tree returned as a Graph so we can have a look how it looks like

```
weightMST := Graph::minimumSpanningTree(G):
plot(Graph::plotGridGraph(weightMST, plotOptions,
                          EdgeColor = RGB::Green))
```

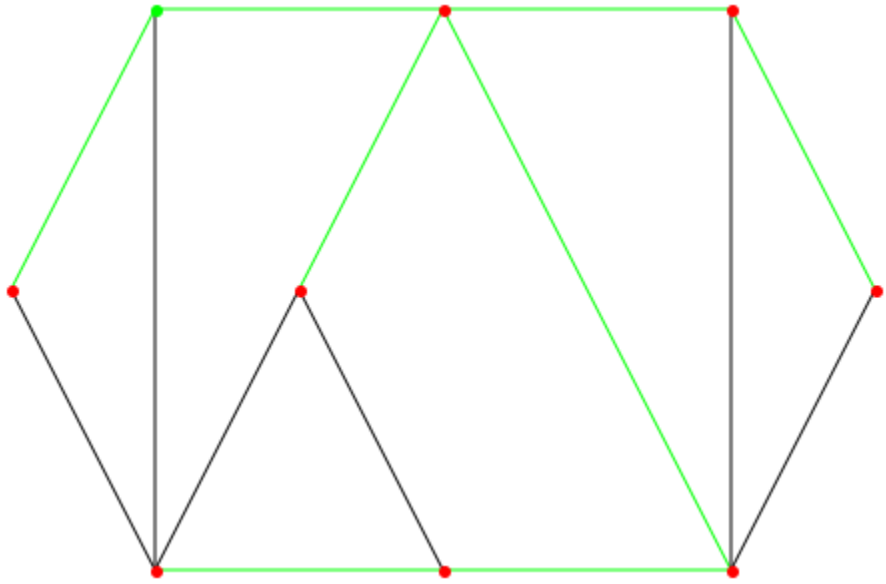


There are two ways of displaying both graphs at the same time:

```
plot(
  Graph::plotGridGraph(G, plotOptions, EdgeColor = RGB::Black),
  Graph::plotGridGraph(weightMST, plotOptions,
                       EdgeColor = RGB::Green)
)
```



```
edgesWeightMST := Graph::getEdges(weightMST):
plot(Graph::plotGridGraph(G, plotOptions,
         EdgeColor = RGB::Black,
         SpecialEdges = edgesWeightMST,
         SpecialEdgeColor = RGB::Green))
```



Example 2

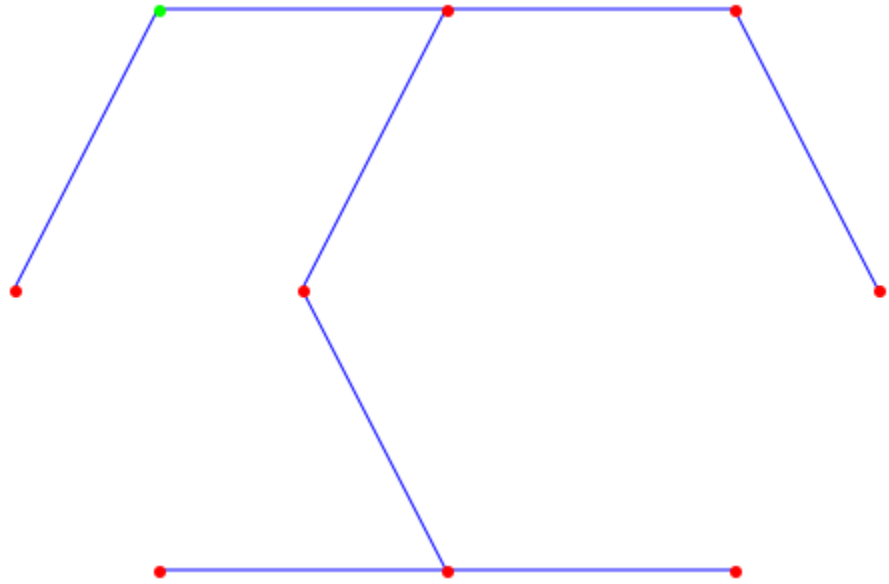
Maybe instead of the weights there is an interest in getting the MST for the costs of the edges.

```
G := Graph([a, b, c, d, e, f, g, h, i],
           [[a, b], [a, h], [b, h], [b, c], [c, d], [d, f], [d, e],
            [f, e], [h, g], [g, f], [c, i], [h, i], [g, i], [c, f]],
         EdgeWeights = [4, 8, 11, 8, 7, 14, 9, 10, 1, 2, 2, 7, 6,  4],
         EdgeCosts   = [4, 8, 11, 8, 7, 14, 9, 10, 1, 2, 2, 7, 6, 12]):

Graph::minimumSpanningTree(G, SearchFor = Costs, ReturnAsTable)
```

[a, b]	4
[b, a]	4
[b, c]	8
[c, b]	8
[c, d]	7
[d, c]	7
[d, e]	9
[e, d]	9, 39
[c, i]	2
[i, c]	2
[f, g]	2
[g, f]	2
[g, h]	1
[h, g]	1
[g, i]	6
[i, g]	6

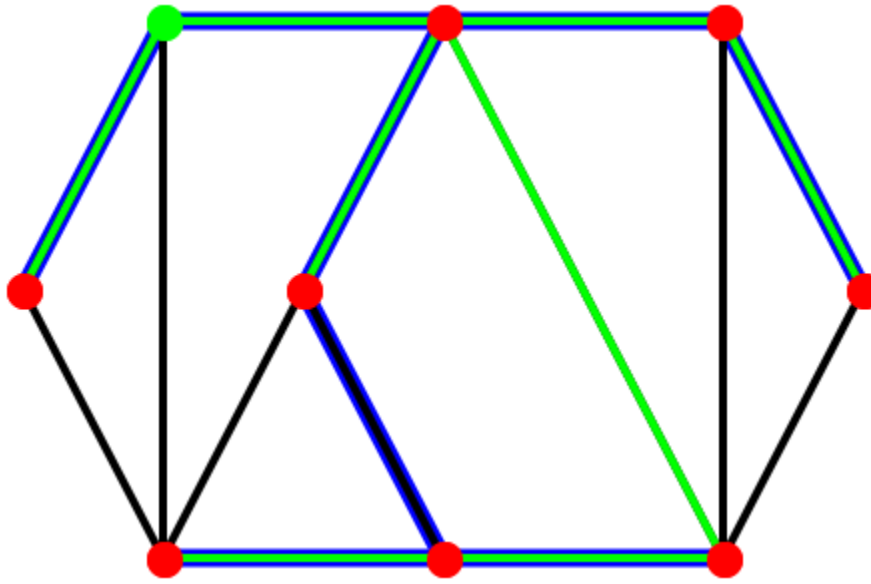
Plotting this spanning tree is just as easy as above:

```
costMST := Graph::minimumSpanningTree(G, SearchFor = Costs):
plot(Graph::plotGridGraph(costMST, plotOptions,
                          EdgeColor = RGB::Blue))
```



To combine both spanning trees, we use different line widths, to avoid one graph being completely covered by the other:

```
plot(
  plot::Group2d(
    Graph::plotGridGraph(costMST, plotOptions,
                         EdgeColor = RGB::Blue),
    LineWidth = 2.5
  ),
  plot::Group2d(
    Graph::plotGridGraph(G, plotOptions,
                EdgeColor = RGB::Black),
    Graph::plotGridGraph(weightMST, plotOptions,
                          EdgeColor = RGB::Green),
    PointSize = 5,
    LineWidth = 1
  )
)
```



Parameters

`G`

`Graph`

Options

`SearchFor`

Can either be `Costs` or `Weights`. Default is `Weights`

`ReturnAsTable`

If omitted, a Graph is returned, otherwise a list containing a table and the sum of the edge weights/costs.

Return Values

Graph consisting of the MST. Only if `ReturnAsTable` was specified, a list containing a table and a number are returned. The table holds the edges with either the weights or costs of each edge and the number is the sum of all edges.

Graph

Creates new graph

Syntax

```
Graph(V, E, <VertexWeights = vw>, <EdgeDescriptions = ed>, <EdgeWeights = ew>, <EdgeCos
```

Description

`Graph(V, E)` creates a graph.

`Graph([v1,...,vn],[e1,...,em])` generates a new undirected graph with n vertices and m edges.

`Graph([1,a,3], [[1,a],[1,3]], Directed)` generates a new directed graph with the vertices `1`, `a`, `3` and the edges `[1,a]`, `[1,3]`.

`Graph([a,b,3], [[a,b],[b,3]], VertexWeights = [1,2,3], EdgeWeights = [4,5])` generates a new directed graph where the vertices have the values `a=1`, `b=2`, `3=3` and the edges `[a,b]=4`, `[b,3]=5`.

`Graph([a,b,3], [[a,b],[b,3]], VertexWeights = [1,None,3], EdgeWeights = [4, None])` generates a new directed graph where the vertices have the values `a=1`, `3=3` and the edges `[a,b]=4`. The difference to the example directly above is that the keyword `None` can be used to not assign a value to a vertex or edge.

Examples

Example 1

An (undirected) graph with four vertices:

```
G1 := Graph([1,a,3], [[1,a],[1,3]]):
Graph::printGraphInformation(G1)
```

```
Vertices: [1, 3, a]
Edges: [[1, 3], [1, a], [3, 1], [a, 1]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [3, a], a = [1], 3 = [1]
Adjacency list (in): 1 = [3, a], a = [1], 3 = [1]
Graph is undirected.
```

Example 2

The same graph but this time with parameter `Directed`:

```
G1 := Graph([1,a,3], [[1,a],[1,3]], Directed):
Graph::printGraphInformation(G1)
```

```
Vertices: [1, 3, a]
Edges: [[1, 3], [1, a]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [3, a], a = [], 3 = []
Adjacency list (in): 1 = [], a = [1], 3 = [1]
Graph is directed.
```

Example 3

The circle graph with predefined vertices:

```
G3 := Graph([a,b,3], [[a,b],[b,3]],
           VertexWeights = [1,2,3],
               EdgeWeights = [4,5]):
Graph::printGraphInformation(G3)
```

```
Vertices: [3, a, b]
Edges: [[3, b], [a, b], [b, 3], [b, a]]
Vertex weights: a = 1, b = 2, 3 = 3 (other existing vertices have no weigh\
t)
Edge descriptions: no edge descriptions.
```

```
Edge weights: [a, b] = 4, [b, 3] = 5, [b, a] = 4, [3, b] = 5 (other existi\
ng edges have no weight)
Edge costs: no edge costs.
Adjacency list (out): a = [b], b = [3, a], 3 = [b]
Adjacency list (in): a = [b], b = [3, a], 3 = [b]
Graph is undirected.
```

Example 4

The circle graph with predefined vertices:

```
G3 := Graph([a,b,3,7], [[a,b],[b,3],[3,7]],
VertexWeights = [1,2,3,4], EdgeWeights = [-1,-2,-5],
EdgeDescriptions = ["Small", None, "Smallest"]):
Graph::printGraphInformation(G3)
```

```
Vertices: [3, 7, a, b]
Edges: [[3, 7], [3, b], [7, 3], [a, b], [b, 3], [b, a]]
Vertex weights: a = 1, b = 2, 3 = 3, 7 = 4 (other existing vertices have n\
o weight)
Edge descriptions: [a, b] = "Small", [3, 7] = "Smallest", [b, a] = "Small"\
, [7, 3] = "Smallest"
Edge weights: [a, b] = -1, [b, 3] = -2, [3, 7] = -5, [b, a] = -1, [3, b] =\
 -2, [7, 3] = -5 (other existing edges have no weight)
Edge costs: no edge costs.
Adjacency list (out): a = [b], b = [3, a], 3 = [7, b], 7 = [3]
Adjacency list (in): a = [b], b = [3, a], 3 = [7, b], 7 = [3]
Graph is undirected.
```

If you look at the edge descriptions, the keyword `None` can be used for every edge which is not supposed to have a description.

Parameters

V

List of vertices

E

List of edges

`vw`, `ew`, `ec`

List of numbers

`ed`

List of strings

Options

`Directed`

The Graph is created as a directed graph.

`Undirected`

The Graph is created as an undirected graph. `Default`.

Graph::outDegree

Returns the outdegree of one or more vertices.

Syntax

```
Graph::outDegree(G, <Vertex>)
```

Description

`Graph::outDegree(G)` returns the number of edges leaving each vertex `Vertex` of the Graph `G`.

`Graph::outDegree(G, Vertex)` returns the outdegree of the vertex `Vertex` in the Graph `G`, i.e., the number of edges `[Vertex, u]`.

`Graph::outDegree(G, [v1, v2, ..., vn])` returns a table in which the keys are `v1, v2, ..., vn` and the corresponding values are the outdegrees.

`Graph::outDegree(G)` returns a table in which each node of `G` is mapped to its outdegree. `Graph::outDegree(G)` is equivalent to `Graph::outDegree(G, Graph::getVertices(G))`.

Examples

Example 1

In a complete graph of n vertices, each vertex has outdegree n - 1:

```
G := Graph::createCompleteGraph(5):
Graph::outDegree(G, [2, 4, 5]), Graph::outDegree(G),
Graph::outDegree(G, Graph::getVertices(G))
```

		1	4	1	4
2	4	2	4	2	4
4	4^2	3	4^2	3	4
5	4	4	4	4	4
		5	4	5	4

The first table shows what happens, if some vertices are specified. The second and third table return all indegrees, but with two different calls (the second is redundant).

Example 2

Remember that also only one vertex needs to be specified as a list !

```
G := Graph::createCompleteGraph(5):
```

```
Graph::outDegree(G, [2])
```

2	4

Parameters

G

A Graph

Options

Vertex

A list containing one or more vertices.

Return Values

Table containing all the outdegrees of the specified vertices.

Graph::plotBipartiteGraph

Plots a Graph in a bipartite layout

Syntax

```
Graph::plotBipartiteGraph(G, <PointSize = n>, <SpecialVertices = [v1, …,vn]>, <SpecialE
```

Description

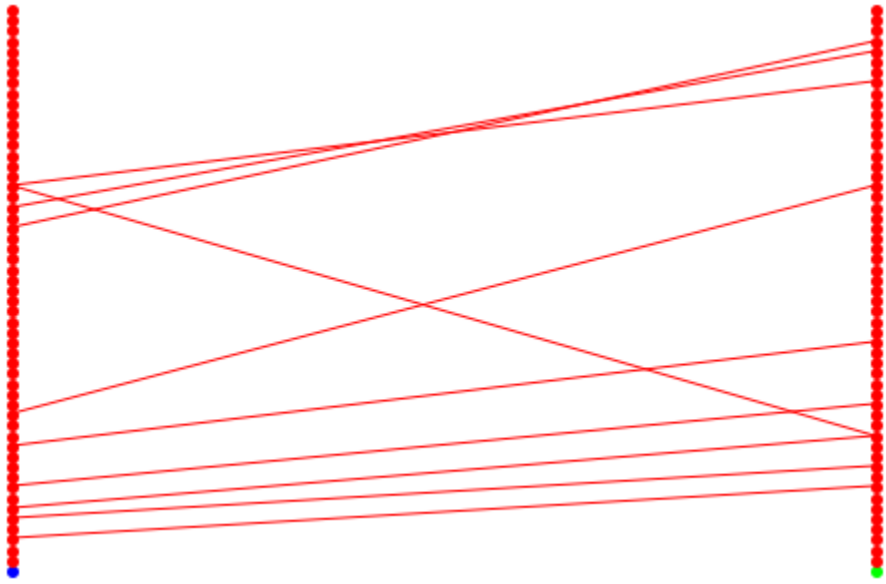
`Graph::plotBipartiteGraph(G)` returns a `plot::Group2d` object in which the vertices are ordered in two rows (from bottom to top). The first vertex in the left row is drawn in blue and the second (the first vertex in the right row) in green. All other vertices are drawn in red. The width of the points is predefined with 40.

Examples

Example 1

A random graph is created and plotted (your output may differ due to random creation):

```
G1 := Graph::createRandomGraph(110, 10, Undirected):
plot(Graph::plotBipartiteGraph(G1)):
```



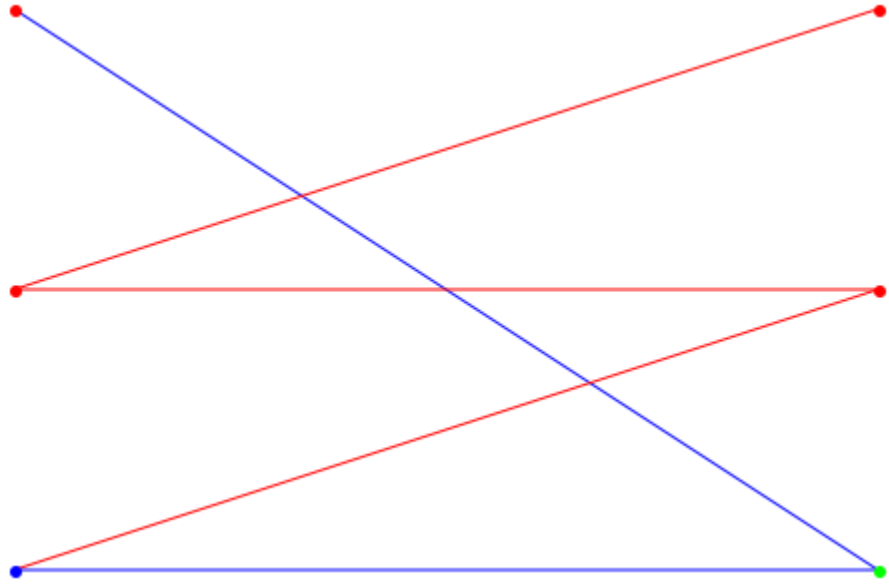
Example 2

If some edges are to be emphasized they can be drawn in a special color:

```
G2 := Graph([1, 2, 3, 4, 5, 6],
            [[1, 2], [2, 3], [3, 4], [4, 5], [1, 6]]):
Graph::bipartite(G2, Lists)
```

$$[[2, 4, 6], [1, 3, 5]]$$

```
edges := [[6, 1], [1, 2]]:
plot(Graph::plotBipartiteGraph(G2, SpecialEdges = edges,
                               SpecialEdgeColor = RGB::Blue))
```



Parameters

G

Graph

n

a positive integer

[v_1, …,v_n]

a list of vertices

[e_1, …,e_n]

a list of edges

Options

`PointSize`

Defines the thickness in which the points are drawn. Default is 40.

`SpecialVertices`

Defines a set of vertices. This option makes only sense if used with the option `SpecialVertexColor`.

`SpecialEdges`

Defines a set of edges. This option makes only sense if used with the option `SpecialEdgeColor`.

`EdgeColor`

Defines a color with which to draw the edges. Default is `RGB ::Red`.

`SpecialEdgeColor`

Defines a color to be used to draw the set of edges specified. This option makes only sense if used with the option `SpecialEdges`.

`VertexColor`

Defines a color with which to draw the vertices. If this option is specified, the first two vertices are set to this color, too. They can be given different colors via `Vertex1Color` and `Vertex2Color`. Default is `RGB::Red`.

`SpecialVertexColor`

Defines a color to be used to draw the set of vertices specified. This option makes only sense if used with the option `SpecialVertices`.

`Vertex1Color`

Defines a color with which to draw the first vertex with (the starting vertex at the bottom of the first set). Default is `RGB::Blue`.

`Vertex2Color`

Defines a color with which to draw the second vertex with (the starting vertex at the bottom of the second set). Default is `RGB::Green`.

Return Values

`plot::Group2d`

Graph::plotCircleGraph

Plots a Graph in a circle layout

Syntax

```
Graph::plotCircleGraph(G, <PointSize = n>, <SpecialVertices = [v1, …,vn]>, <SpecialEdge
```

Description

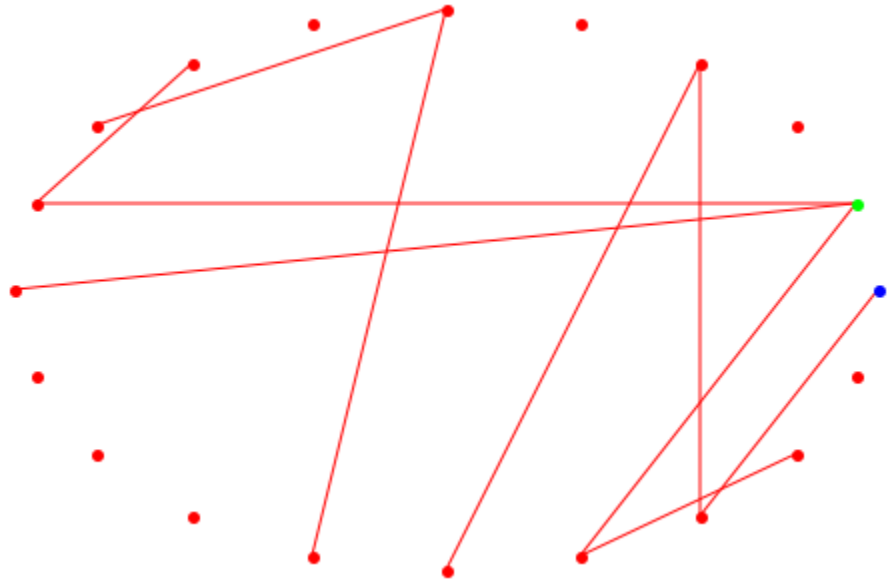
`Graph::plotCircleGraph(G)` returns a `plot::Group2d` object in which the vertices are ordered in a circle (rightmost position upwards). The first vertex is drawn in blue and the second in green. All other vertices are drawn in red. The edges are drawn in red. The width of the points is predefined with 40. If a vertex points to itself it will be drawn outside

Examples

Example 1

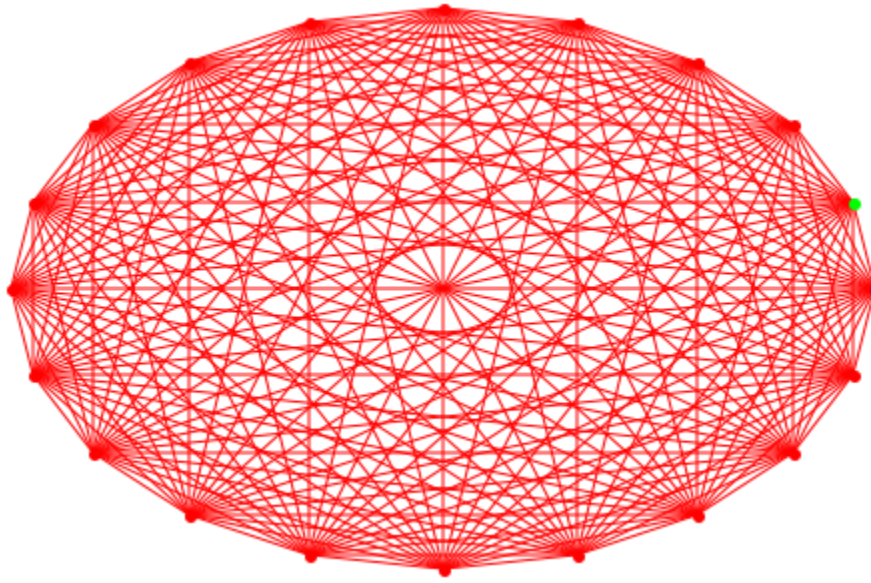
A random graph is created and plotted (your output may differ due random creation):

```
G1 := Graph::createRandomGraph(20, 10, Undirected):
plot(Graph::plotCircleGraph(G1))
```



Next, a complete graph will be plotted.

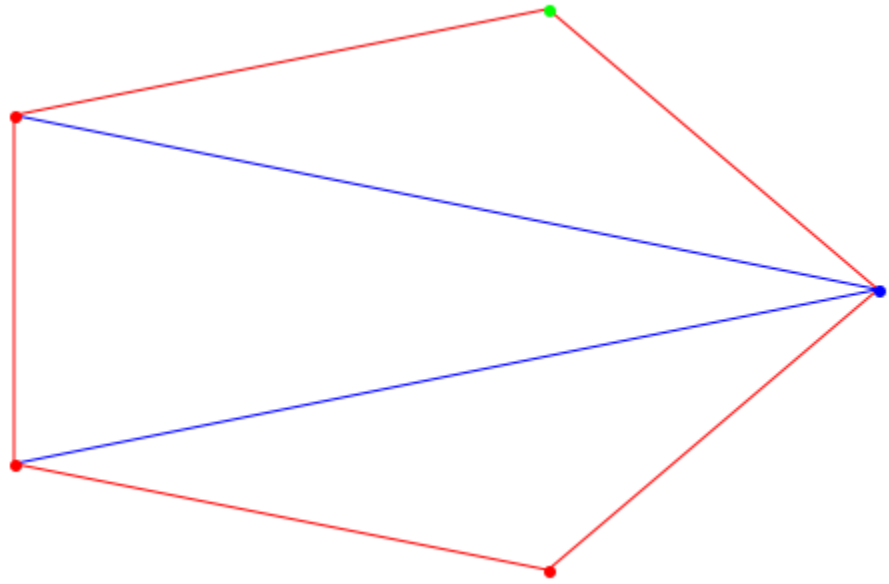
```
G1 := Graph::createCompleteGraph(20):
plot(Graph::plotCircleGraph(G1))
```



Example 2

If some edges are to be emphasized they can be drawn in a special color:

```
G2 := Graph([1, 2, 3, 4, 5],
            [[1, 2], [2, 3], [3, 4], [4, 5],
             [1, 3], [1, 4], [1, 5]]):
edges := [[1, 3], [1, 4]]:
plot(Graph::plotCircleGraph(G2,
                SpecialEdges = edges, SpecialEdgeColor = RGB::Blue))
```



Parameters

G

Graph

n

a positive integer

[v_1, …,v_n]

a list of vertices

[e_1, …,e_n]

a list of edges

Options

`PointSize`

Defines the thickness in which the points are drawn. Default is 40.

`SpecialVertices`

Defines a set of vertices. This option makes only sense if used with the option `SpecialVertexColor`.

`SpecialEdges`

Defines a set of edges. This option makes only sense if used with the option `SpecialEdgeColor`.

`EdgeColor`

Defines a color with which to draw the edges. Default is RGB::Red

`SpecialEdgeColor`

Defines a color to be used to draw the set of edges specified. This option makes only sense if used with the option `SpecialEdges`

`VertexColor`

Defines a color with which to draw the vertices. If this option is specified, the first two vertices are set to this color, too. They must be specified via `Vertex1Color` and `Vertex2Color` to distinct them again. Default is RGB::Red

`SpecialVertexColor`

Defines a color to be used to draw the set of vertices specified. This option makes only sense if used with the option `SpecialVertices`

`Vertex1Color`

Defines a color with which to draw the uppermost left (first) vertex with (the starting vertex). Default is RGB::Blue

`Vertex2Color`

Defines a color with which to draw the second vertex with. Default is RGB::Green

Return Values

`plot::Group2d`

Graph::plotGridGraph

Plots a Graph in a grid layout

Syntax

```
Graph::plotGridGraph(G, <PointSize = n>, <VerticesPerLine = n>, <VertexOrder = [n1, …,n
```

Description

`Graph::plotGridGraph(G)` returns a plot::Scene object in which the vertices are square ordered (topmost left to downmost right). The number of vertices per line is the floor of the squareroot of the number of the vertices. The first vertex is drawn in RGB::Blue and the second in RGB::Green. All other vertices are drawn in RGB::Red. The edges are drawn in RBG::Red. The width of the points is predefined with 40. If the last line contains only one vertex, it will be drawn centered in the middle of the line.

`Graph::plotGridGraph(G, VerticesPerLine=n)` returns a plot::Scene object like described above with one exception. In every line there are exactly `n` vertices. They appear in sorted order depending on their name. If the last row consists of only one vertex, this one will be centered.

`Graph::plotGridGraph(G, VerticesPerLine=[v1..vn])` returns a plot::Scene object like described above with one exception. In line 1 there are exactly `v1` vertices placed. In line 2 there are `v2` vertices and so on. The last line contains `vn` vertices. They appear in sorted order depending on their name. The sum of the numbers specified in VerticesPerLine must equal the number of vertices in the graph.

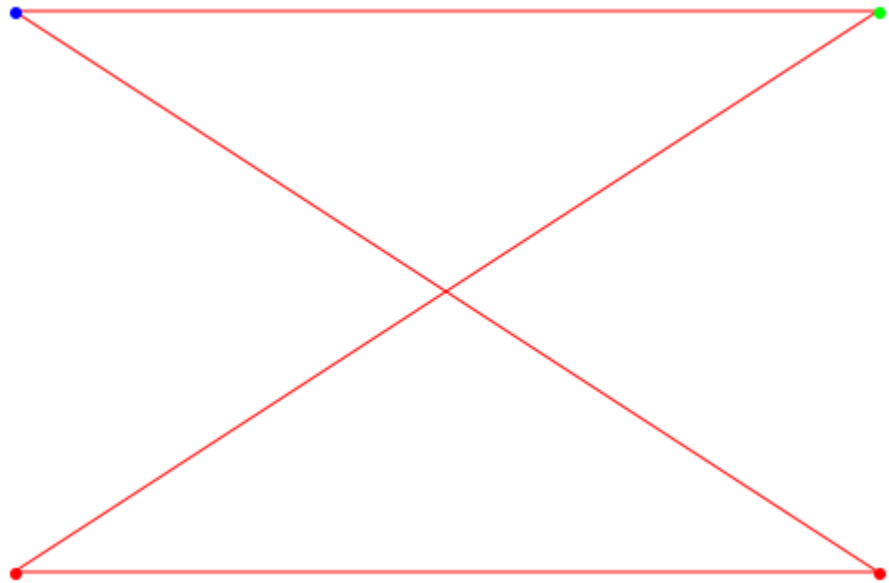
`Graph::plotGridGraph(G, VerticesPerLine=n, VertexOrder=[v1..vn])` returns a plot::Scene object like described above with one exception. In every line there are exactly `n` vertices. They appear in sorted order depending on the order that was specified in VertexOrder. `vi` can consist of any vertex defined as well as the substitute `None`. Nevertheless the number of vertices in `G` must not exceed the number of VerticesPerLine. If the last line holds only one vertex, it will be centered.

Examples

Example 1

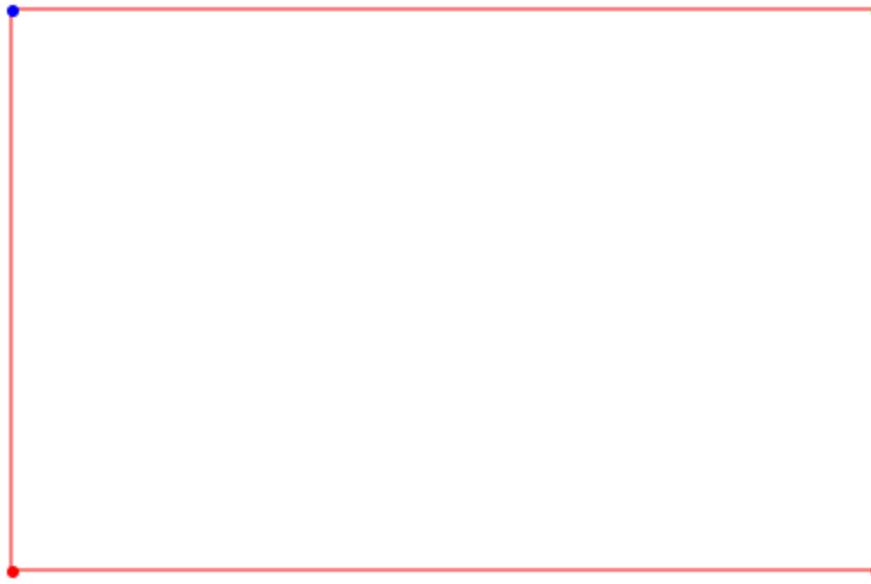
First, a small graph is created and plotted with the default values:

```
G := Graph([a, b, c, d], [[a, b], [b, c], [c, d], [d, a]]):
plot(Graph::plotGridGraph(G))
```



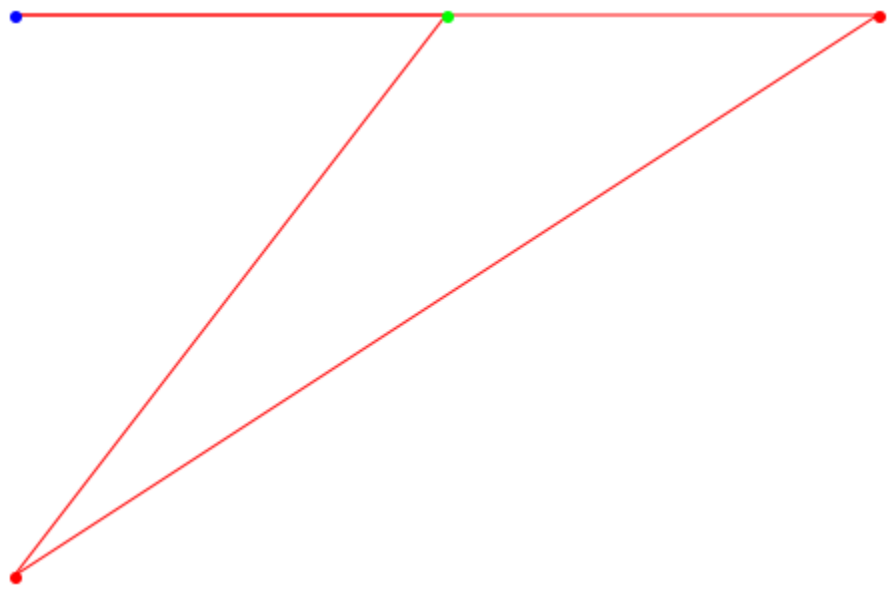
Now, we exchange the two vertices `c` and `d`. The order given above was `[a, b, c, d]`:

```
plot(Graph::plotGridGraph(G, VertexOrder = [a, b, d, c]))
```



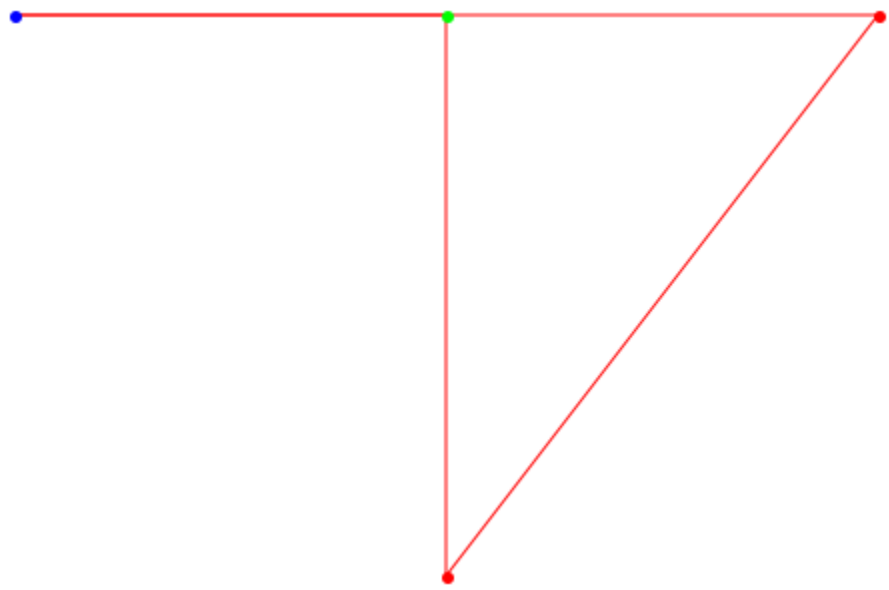
If only one vertex is placed in the last line, it will be centered:

```
plot(Graph::plotGridGraph(G, VertexOrder = [a, b, d, c],
                          VerticesPerLine = 3))
```



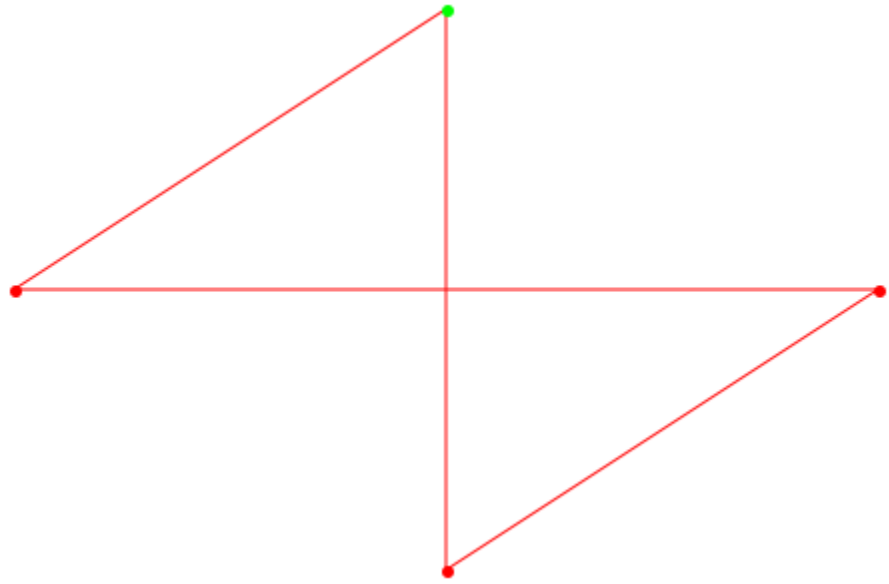
The same result can be gained by defining the Vertices per line specifically (in this case the number of vertices must be no less than the number of vertices in the graph):

```
plot(Graph::plotGridGraph(G, VertexOrder = [a, b, d, c],
                          VerticesPerLine = [3, 1]))
```



Now we get to the point, why the plot routine got the name Grid. The substitute `None` can be used whenever a place should be skipped. Think of some drawing paper with caskets. The layout is exactly the same. In this case it would consist of 3 caskets in each row. None leaves it blank, while a vertex from the Graph is drawn. Because the first casket is empty, the first color is omitted, too. The vertex `a` which is placed in the second casket is drawn as predefined in `Vertex2Color` (`RGB::Green`):

```
plot(Graph::plotGridGraph(G,
       VertexOrder = [None, a,    None,
                      b,    None, c,
                      None, d,    None],
       VerticesPerLine = 3))
```



Example 2

With the knowledge obtained so far, it is possible to get deeper into the art of creating objects. One of the most usefull outputs is that of a tree. Thus a Graph is created to be used for the tree output:

```
TreeGraph := Graph([a, b, c, d, e, f, g, h, i, j, k, l],
             [[a, b], [a, c], [b, d], [b, e], [c, f], [c, g],
              [d, h], [e, i], [e, j], [f, k], [g, l]], Directed):
```

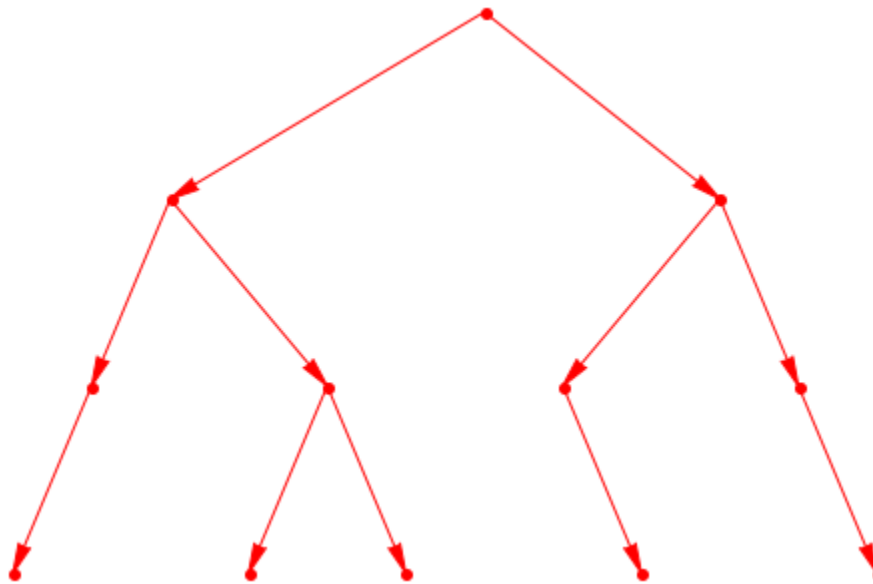
Next we define a special vertex order, because the vertices are not drawn the way they were defined:

```
vOrder :=
[None, None, None, None, None, None,
 a,    None, None, None, None, None,
 None, None, b,    None, None, None,
 None, None, None, c,    None, None,
 None, d,    None, None, e,    None,
 None, f,    None, None, g,    None,
 h,    None, None, i,    None, j,
```

```
 None, None, k,    None, None, l    ]:
```

Now it is time to have a look at how the tree looks:

```
plot(Graph::plotGridGraph(TreeGraph,
              VerticesPerLine = 12, VertexOrder = vOrder))
```

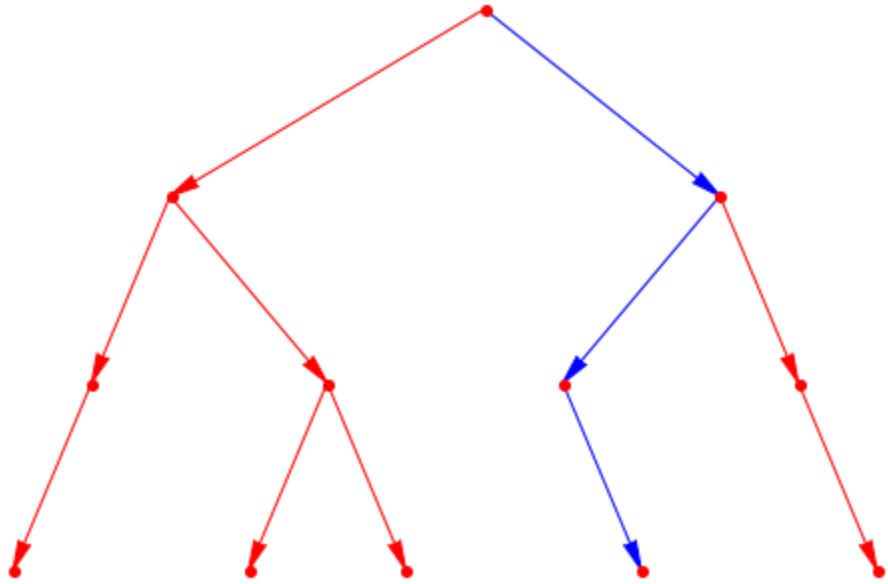


Now we want to see the path from vertex `a` to Vertexb. For this example it will be given explicitly. For bigger graphs one of the `shortestPath` procedures is recommended:

```
specialPath := [[a, c], [c, f], [f, k]]:
```

Finally we draw the path inside the Graph and have a good overview about the path it takes:

```
plot(Graph::plotGridGraph(TreeGraph,
         VerticesPerLine = 12, VertexOrder = vOrder,
         SpecialEdges = specialPath, SpecialEdgeColor = RGB::Blue))
```

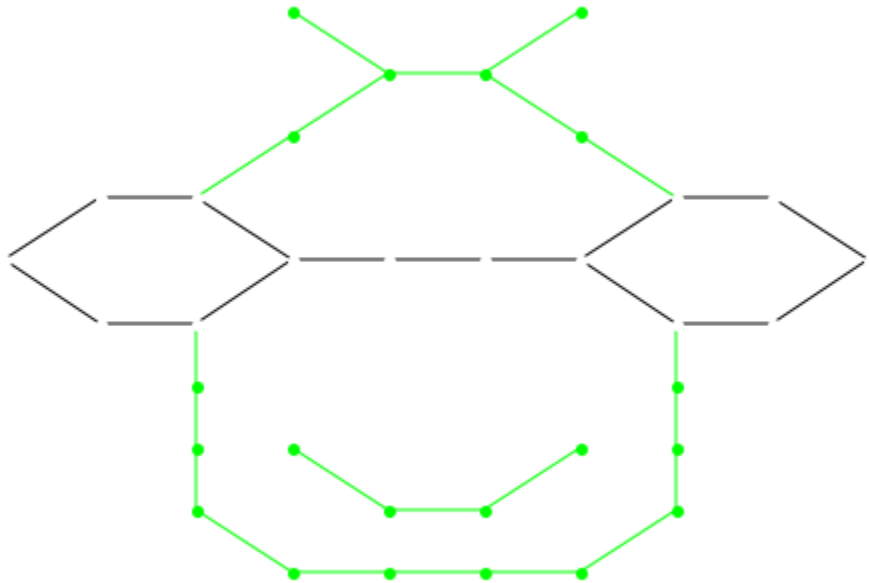


Example 3

To show what can be done with more detailed and complex examples a small outer face is being drawn (have a close look ath the Vertices which are not drawn, because the color is set to RBG::White and thus equals the background-color). Additionaly, the “eyes” have been colored differently, so the usage of `SpecialVertexColor` could be presented:

```
Smile := Graph([1,2,4,5,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
                21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36],
          [[1,4],[2,5],[4,5],[7,4],[5,8],[7,10],[9,10],[10,14],[9,13],
           [13,19],[19,20],[20,14],[14,15],[15,16],[16,17],[17,11],
           [ 8,11],[11,12],[12,18],[18,22],[22,21],[21,17],[21,24],
           [24,28],[28,32],[32,36],[36,35],[35,34],[34,33],[33,29],
           [29,25],[25,23],[23,20],[26,30],[30,31],[31,27]]):
plot(
  Graph::plotGridGraph(Smile, VerticesPerLine = 10,
     VertexOrder = [
       None,None,None,   1,None,None,   2,None,None,None,
       None,None,None,None,   4,   5,None,None,None,None,
       None,None,None,   7,None,None,   8,None,None,None,
       None,   9,  10,None,None,None,None,  11,  12,None,
```

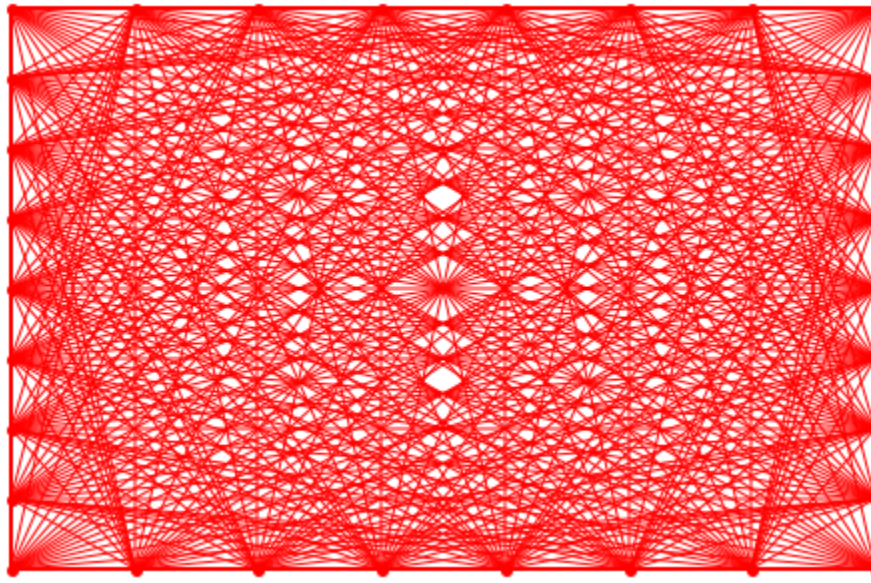
```
        13,None,None,  14,  15,  16,  17,None,None,  18,
      None,  19,  20,None,None,None,None,  21,  22,None,
      None,None,  23,None,None,None,None,  24,None,None,
      None,None,  25,  26,None,None,  27,  28,None,None,
      None,None,  29,None,  30,  31,None,  32,None,None,
      None,None,None,  33,  34,  35,  36,None,None,None],
    SpecialEdges = [[13,9],[9,10],[10,14],[14,20],[20,19],[19,13],
      [14,15],[15,16],[16,17],[17,21],[21,22],[22,18],[18,12],[12,11],
      [11,17]],
    SpecialVertices = [1,2,4,5,7,8,23,24,25,26,27,28,29,30,31,
      32,33,34,35,36],
    VertexColor = RGB::White,
    SpecialVertexColor = RGB::Green, EdgeColor = RGB::Green,
    SpecialEdgeColor = RGB::Black)
)
```



Example 4

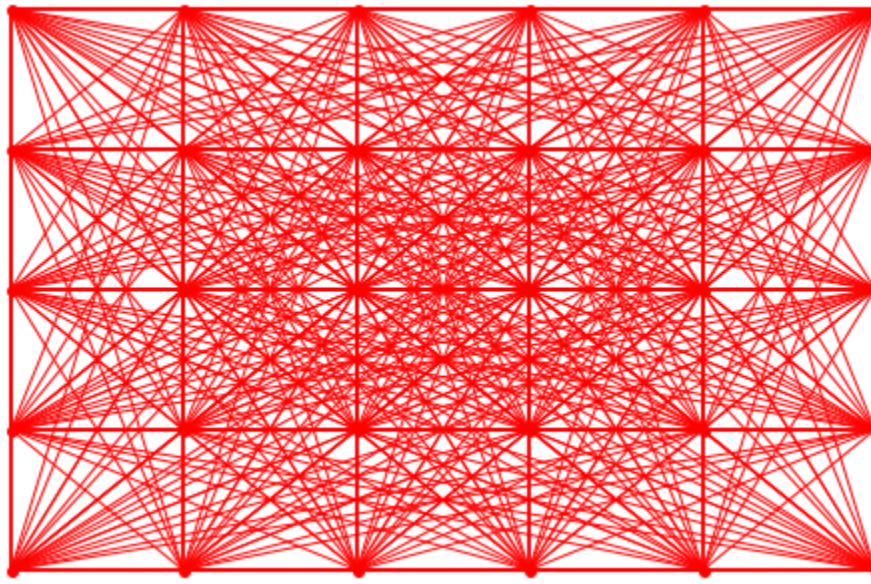
The next example is a complete graph drawn with the vertices ordered in a square so that all vertices can be connected inside the square:

```
CG := Graph::createCompleteGraph(30):
plot(Graph::plotGridGraph(CG,
       VerticesPerLine = [8, 2, 2, 2, 2, 2, 2, 8],
       Vertex1Color = RGB::Red, Vertex2Color = RGB::Red))
```



Using the default values in every line vertices are drawn and the graph looks not as "dense" as the above one:

```
plot(Graph::plotGridGraph(CG, Vertex1Color = RGB::Red,
                          Vertex2Color = RGB::Red))
```

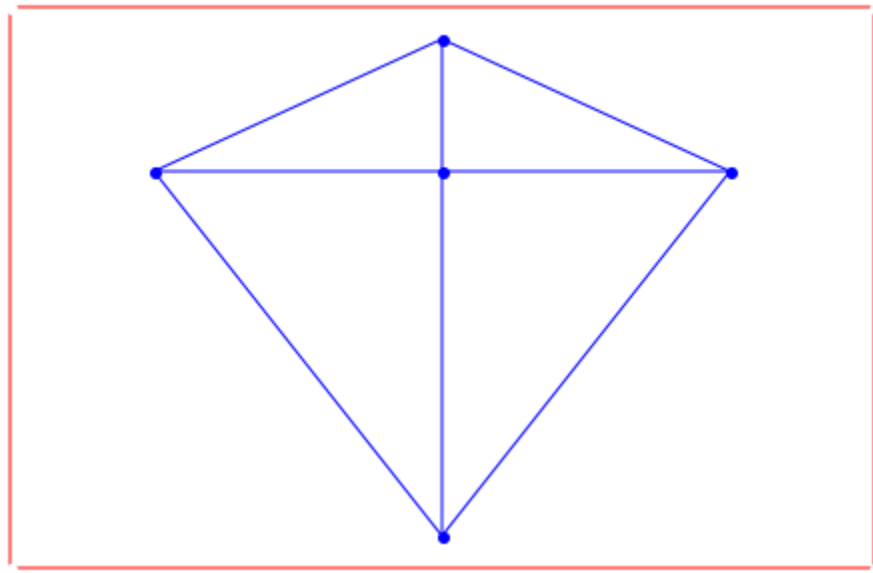


Example 5

The last example shows a small kite inside a frame:

```
Shield := Graph([1, 2, 3, 4, 5, 6, 7, 8, 9],
          [[1, 2], [1, 8], [2, 9], [8, 9], [3, 5], [4, 5],
           [5, 6], [5, 7], [3, 6], [6, 7], [3, 4], [4, 7]]):
plot(Graph::plotGridGraph(Shield, Vertex1Color = RGB::White,
       Vertex2Color = RGB::White, VertexColor = RGB::White,
       VerticesPerLine = 7,
       VertexOrder = [
           1,None,None,None,None,None,   2,
        None,None,None,   3,None,None,None,
        None,None,None,None,None,None,None,
        None,None,None,None,None,None,None,
        None,None,None,None,None,None,None,
        None,   4,None,   5,None,   6,None,
        None,None,None,None,None,None,None,
        None,None,None,None,None,None,None,
        None,None,None,None,None,None,None,
        None,None,None,None,None,None,None,
```

```
 None,None,None,None,None,None,None,
 None,None,None,None,None,None,None,
 None,None,None,None,None,None,None,
 None,None,None,None,None,None,None,
 None,None,None,None,None,None,None,
 None,None,None,None,None,None,None,
 None,None,None,   7,None,None,None,
    8,None,None,None,None,None,   9],
SpecialVertices = [3, 4, 5, 6, 7],
SpecialVertexColor = RGB::Blue,
SpecialEdges = [[3, 5], [4, 5], [5, 6], [5, 7],
                [3, 6], [6, 7], [3, 4], [4, 7]],
SpecialEdgeColor = RGB::Blue))
```



Parameters

G

Graph

n

a positive integer

[n_1, …,n_m]

a list of positive integers

[v_1, …,v_n]

a list of vertices

[e_1, …,e_n]

a list of edges

Options

PointSize

Defines the thickness in which the points are drawn. Default is 40.

VerticesPerLine

If specified as single number, this many vertices are placed in every row. If specified as list, the number of vertices per line are read out of the list. If the vertices are specified as list, the number of vertices must match either the number of vertices defined in the graph, or, if specified, the number defined in `VertexOrder`.

VertexOrder

Defines an order in which the vertices are to be placed. It starts in the upper left and ends in the lower right. The number of specified vertices must match the number of vertices defined in the graph or the sum of the vertices specified in VerticesPerLine.

SpecialVertices

Defines a set of vertices. This option makes only sense if used with the option `SpecialVertexColor`.

`SpecialEdges`

Defines a set of edges. This option makes only sense if used with the option `SpecialEdgeColor`.

`EdgeColor`

Defines a color with which to draw the edges. Default is RGB::Red

`SpecialEdgeColor`

Defines a color to be used to draw the set of edges specified. This option makes only sense if used with the option `SpecialEdges`

`VertexColor`

Defines a color with which to draw the vertices. If this option is specified, the first two vertices are set to this color, too. They must be specified via `Vertex1Color` and `Vertex2Color` to distinct them again. Default is RGB::Red

`SpecialVertexColor`

Defines a color to be used to draw the set of vertices specified. This option makes only sense if used with the option `SpecialVertices`

`Vertex1Color`

Defines a color with which to draw the uppermost left (first) vertex with (the starting vertex). If `VertexOrder` holds a `None` for this vertex, it will be skipped. Default is RGB::Blue

`Vertex2Color`

Defines a color with which to draw the second vertex with. If `VertexOrder` holds a `None` for this vertex, it will be skipped. Default is RGB::Green

Return Values

Plot::Scene.

Graph::printEdgeCostInformation

Prints the edge costs of a graph

Syntax

```
Graph::printEdgeCostInformation(G)
```

Description

`Graph::printEdgeCostInformation` prints the edge costs of a graph.

`Graph::printEdgeCostInformation(G)` prints the edge costs of the graph G

Examples

Example 1

A circle graph is created and the edge costs of it printed to screen:

```
G := Graph::createCircleGraph(3):
Graph::printEdgeCostInformation(G)
```

```
No edge costs defined.
```

```
G := Graph::setEdgeCosts(G, [[1, 2], [2, 3], [3, 1]], [10, 20, 30]):
Graph::printEdgeCostInformation(G)
```

```
Edge costs existing in the graph:
----------------------------------------
Edge [1, 2] has cost 10
Edge [2, 3] has cost 20
Edge [3, 1] has cost 30
```

Parameters

G

`Graph`

Return Values

Text containing information about the edge costs of a graph.

Graph::printEdgeDescInformation

Prints the edge descriptions of a graph

Syntax

```
Graph::printEdgeDescInformation(G)
```

Description

`Graph::printEdgeDescInformation` prints the edge descriptions of a graph.

`Graph::printEdgeDescInformation(G)` prints the edge descriptions of the graph *G*

Examples

Example 1

A circle graph is created and the edge descriptions of it printed to screen:

```
G := Graph::createCircleGraph(3):
Graph::printEdgeDescInformation(G)
```

```
No edge descriptions defined.
```

```
G := Graph::setEdgeDescriptions(G, [[1, 2], [2, 3], [3, 1]],
                                   ["Shortcut","Highway","Speedup"]):
Graph::printEdgeDescInformation(G)
```

```
Edge descriptions existing in the graph:
----------------------------------------
Edge [1, 2] = "Shortcut"
Edge [2, 3] = "Highway"
Edge [3, 1] = "Speedup"
```

Parameters

G

Graph

Return Values

Text containing information about the edge descriptions of a graph.

Graph::printEdgeInformation

Prints the edges of a graph

Syntax

```
Graph::printEdgeInformation(G)
```

Description

`Graph::printEdgeInformation` prints the edges of a graph.

`Graph::printEdgeInformation(G)` prints the edges used in the graph *G*.

Examples

Example 1

A circle graph is created and the edges of it printed to screen:

```
G := Graph::createCircleGraph(3):
Graph::printEdgeInformation(G)
```

```
Edges existing in the graph:
----------------------------
[1, 2], [2, 3], [3, 1]
```

Example 2

A complete graph is created and the edges of it printed to screen:

```
G := Graph::createCompleteGraph(3):
Graph::printEdgeInformation(G)
```

```
Edges existing in the graph:
```

```
----------------------------
[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]
```

Parameters

G

Graph

Return Values

Text containing information about the edges of a graph.

Graph::printEdgeWeightInformation

Prints the edge weights of a graph

Syntax

```
Graph::printEdgeWeightInformation(G)
```

Description

`Graph::printEdgeWeightInformation` prints the edge weights of a graph.

`Graph::printEdgeWeightInformation(G)` prints the edge weights of the graph *G*

Examples

Example 1

A circle graph is created and the edge weights of it printed to screen:

```
G := Graph::createCircleGraph(3):
Graph::printEdgeWeightInformation(G)
```

```
No edge weights defined.
```

```
G := Graph::setEdgeWeights(G, [[1, 2], [2, 3], [3, 1]], [10, 20, 30]):
Graph::printEdgeWeightInformation(G)
```

```
Edge weights existing in the graph:
----------------------------------------
Edge [1, 2] has weight 10
Edge [2, 3] has weight 20
Edge [3, 1] has weight 30
```

Parameters

G

Graph

Return Values

Text containing information about the edge weights of a graph.

Graph::printGraphInformation

Prints the edges of a graph

Syntax

```
Graph::printGraphInformation(G)
```

Description

`Graph::printGraphInformation` prints a summary of various information about a graph.

`Graph::printGraphInformation(G)` prints a summary of the graph *G*

Examples

Example 1

A circle graph is created and a summary of it printed to screen:

```
G := Graph::createCircleGraph(3):
Graph::printGraphInformation(G)
```

```
Vertices: [1, 2, 3]
Edges: [[1, 2], [2, 3], [3, 1]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [2], 2 = [3], 3 = [1]
Adjacency list (in): 1 = [3], 2 = [1], 3 = [2]
Graph is directed.
```

Example 2

A complete graph is created and a summary of it printed to screen:

```
G := Graph::createCompleteGraph(3):
Graph::printGraphInformation(G)
```

```
Vertices: [1, 2, 3]
Edges: [[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [2, 3], 2 = [1, 3], 3 = [1, 2]
Adjacency list (in): 1 = [2, 3], 2 = [1, 3], 3 = [1, 2]
Graph is undirected.
```

Parameters

G

Graph

Return Values

Text containing information about the graph.

Graph::printVertexInformation

Prints vertex information of a graph

Syntax

```
Graph::printVertexInformation(G)
```

Description

`Graph::printVertexInformation` prints the edges of a graph.

`Graph::printVertexInformation(G)` prints the edges of the graph G

Examples

Example 1

A circle graph is created and information of the vertices printed to screen:

```
G := Graph::createCircleGraph(3):
Graph::printVertexInformation(G)
```

```
Vertices existing in the graph:
------------------------------
Vertex 1 has weight None
Vertex 2 has weight None
Vertex 3 has weight None
```

Example 2

A complete graph is created and information of the vertices printed to screen:

```
G := Graph::createCompleteGraph(3):
Graph::printVertexInformation(G)
```

```
Vertices existing in the graph:
-------------------------------
Vertex 1 has weight None
Vertex 2 has weight None
Vertex 3 has weight None
```

Parameters

G

Graph

Return Values

Text containing information about the vertices of a graph.

Graph::removeEdge

Removes one or several edges from a graph

Syntax

```
Graph::removeEdge(G, e)
```

```
Graph::removeEdge(G, l)
```

Description

`Graph::removeEdge(G, [e1, ..., en])` removes edges `e1...en` from graph `G`.

`Graph::removeEdge` deletes one or several edges from a graph. An edge is represented by a list containing two vertices of the graph. A warning is printed if the specified edge is not contained in the graph.

`Graph::removeEdge(G, e)` removes the edge `e` from the graph `G`.

`Graph::removeEdge(G, l)` removes all edges in list `l` from graph `G`.

Examples

Example 1

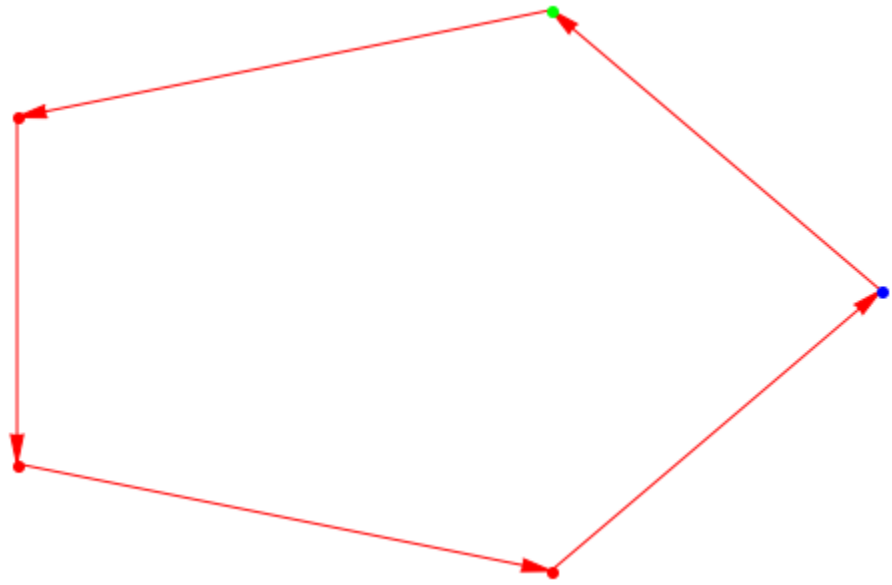
Removing an edge from a cyclic graph results in a (degenerated) tree:

```
G1 := Graph::createCircleGraph(5):
Graph::printGraphInformation(G1)
```

```
Vertices: [1, 2, 3, 4, 5]
Edges: [[1, 2], [2, 3], [3, 4], [4, 5], [5, 1]]
Vertex weights: no vertex weights.
```

```
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [2], 2 = [3], 3 = [4], 4 = [5], 5 = [1]
Adjacency list (in): 1 = [5], 2 = [1], 3 = [2], 4 = [3], 5 = [4]
Graph is directed.
```

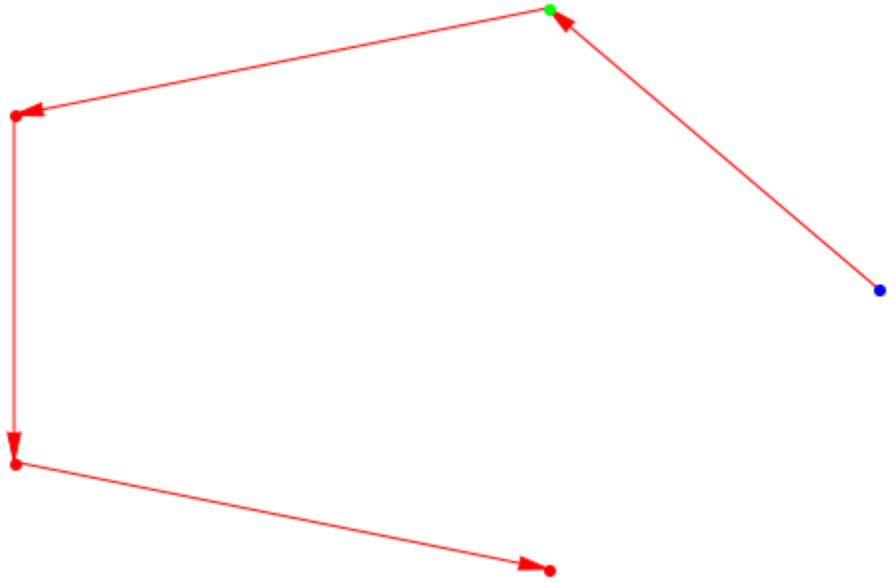
```
plot(Graph::plotCircleGraph(G1))
```



```
G2 := Graph::removeEdge(G1, [[5, 1]]):
Graph::printGraphInformation(G2)
```

```
Vertices: [1, 2, 3, 4, 5]
Edges: [[1, 2], [2, 3], [3, 4], [4, 5]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [2], 2 = [3], 3 = [4], 4 = [5], 5 = []
Adjacency list (in): 1 = [], 2 = [1], 3 = [2], 4 = [3], 5 = [4]
```

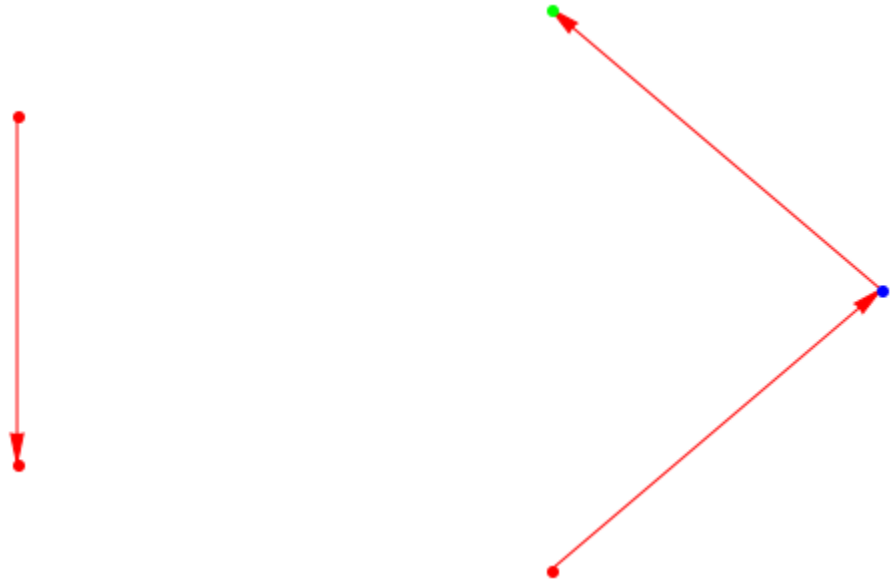
```
Graph is directed.

plot(Graph::plotCircleGraph(G2))
```



If more than one edge is to be removed they must also be specified in a list:

```
edges := [[2, 3], [4, 5]]:
G3 := Graph::removeEdge(G1, edges):
Graph::printGraphInformation(G3)
```

```
Vertices: [1, 2, 3, 4, 5]
Edges: [[1, 2], [3, 4], [5, 1]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [2], 2 = [], 3 = [4], 4 = [], 5 = [1]
Adjacency list (in): 1 = [5], 2 = [1], 3 = [], 4 = [3], 5 = []
Graph is directed.

plot(Graph::plotCircleGraph(G3))
```



Parameters

`l`

A list of edges

`e`

An edge

`G`

A graph

Return Values

Graph

Graph::removeVertex

Removes one or several vertices from a graph

Syntax

```
Graph::removeVertex(G, v)
```

```
Graph::removeVertex(G, l)
```

Description

`Graph::removeVertex(G, v)` removes vertex `v` from graph `G`.

`Graph::removeVertex(G, [v1, ..., vn])` removes vertices `v1...vn` from graph `G`.

`Graph::removeVertex` deletes one or several vertices from a graph. A warning is printed if the specified vertex is not contained in the graph.

Note: If a vertex is connected to other vertices with edges, they will be removed from the graph, too!

`Graph::removeVertex(G, v)` removes the vertex `v` from the graph `G`.

`Graph::removeVertex(G, l)` removes all vertices in list `l` from graph `G`.

Examples

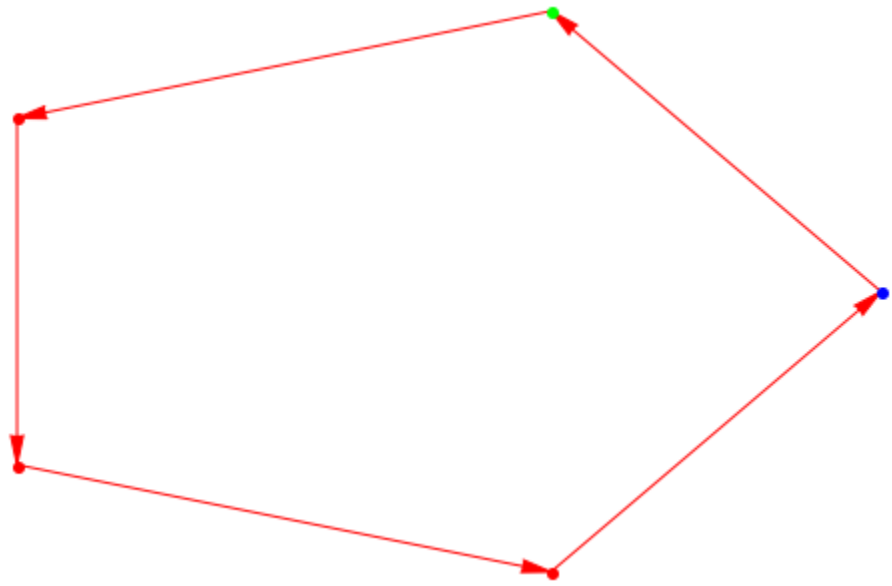
Example 1

Removing a vertex from a cyclic graph removes also two edges:

```
G1 := Graph::createCircleGraph(5):
Graph::printGraphInformation(G1)
```

```
Vertices: [1, 2, 3, 4, 5]
Edges: [[1, 2], [2, 3], [3, 4], [4, 5], [5, 1]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [2], 2 = [3], 3 = [4], 4 = [5], 5 = [1]
Adjacency list (in): 1 = [5], 2 = [1], 3 = [2], 4 = [3], 5 = [4]
Graph is directed.
```

```
plot(Graph::plotCircleGraph(G1))
```

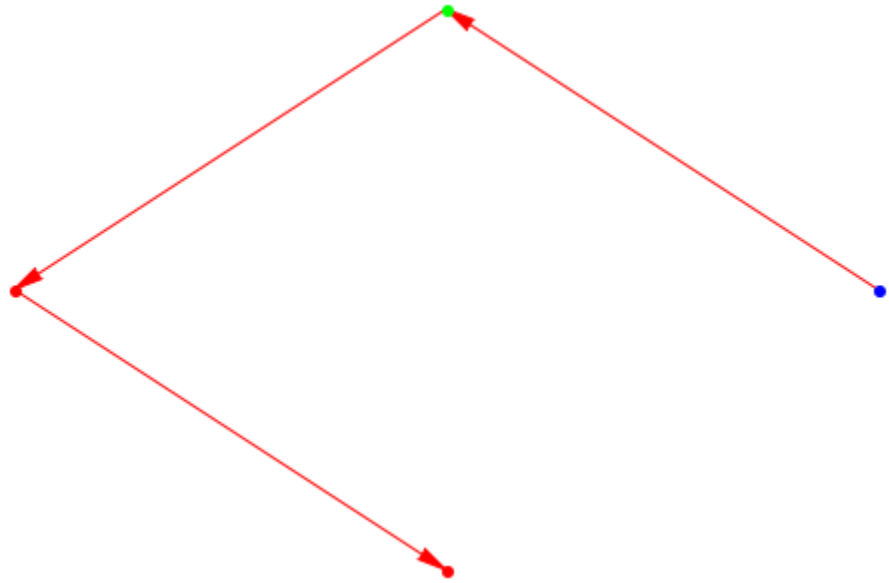


```
G2 := Graph::removeVertex(G1, [1]):
Graph::printGraphInformation(G2)
```

```
Vertices: [2, 3, 4, 5]
Edges: [[2, 3], [3, 4], [4, 5]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 2 = [3], 3 = [4], 4 = [5], 5 = []
```

```
Adjacency list (in): 2 = [], 3 = [2], 4 = [3], 5 = [4]
Graph is directed.
```

```
plot(Graph::plotCircleGraph(G2))
```

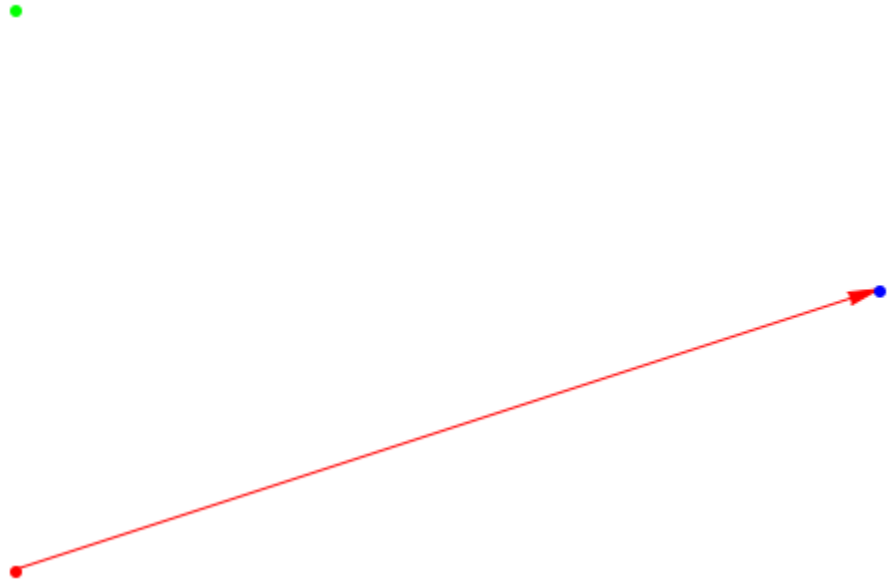


If more than one vertex is to be removed they must also be specified in a list:

```
vertices := [2, 4]:
G3 := Graph::removeVertex(G1, vertices):
Graph::printGraphInformation(G3)
```

```
Vertices: [1, 3, 5]
Edges: [[5, 1]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: no edge weights.
Edge costs: no edge costs.
Adjacency list (out): 1 = [], 2 = [], 3 = [], 5 = [1]
Adjacency list (in): 1 = [5], 2 = [], 3 = [], 5 = []
Graph is directed.
```

```
plot(Graph::plotCircleGraph(G3))
```



Parameters

`l`

A list of vertices

`v`

A vertex

`G`

A graph

Return Values

Graph

Graph::residualGraph

Computes the residual graph

Syntax

```
Graph::residualGraph(G, f, <Extended>)
```

Description

`Graph::residualGraph(G, flow)` computes the residual of the graph `G` with respect to the flow `flow`, meaning the graph that remains when the flow `flow` is "subtracted" from `G`.

`Graph::residualGraph` computes the residual graph with respect to a given flow. A flow in a Graph is a table `tbl`, where `tbl[[i,j]]` gives the number of units flowing from vertex `i` to vertex `j`.

If the optional argument `Extended` is given, then also those edges with a zero residual capacity are considered, otherwise these edges are omitted.

Examples

Example 1

In the following call, `G2` is the graph consisting of the remaining transport capacities after a given flow:

```
G1 := Graph::createCompleteGraph(3):
G2 := Graph::residualGraph(G1,
     table( [1, 2] = 1, [2, 1] = 1/2,
            [1, 3] = 0, [3, 1] = 0.5,
            [2, 3] = 1, [3, 2] = 0 ) ):
Graph::getEdgeWeights(G2)
```

[2, 1]	$\frac{1}{2}$
[1, 3]	1
[3, 1]	0.5
[3, 2]	1

The algorithm detects the lack of edge weights and edge costs and sets all edge weights and costs to default values of 1.

Example 2

The resulting graph depends on whether the option `Extended` is used:

```
V := [1, 2, 3, q, s]:
Edge := [[q, 1], [1, 2], [1, 3], [2, 3], [3, s]]:
up := [5, 4, 4, 2, 5]:
G := Graph(V,Edge,EdgeWeights = up, Directed):
flow := table([q, 1] = 5, [3, s] = 5, [1, 2] = 1,
              [1, 3] = 4, [2, 3] = 1):
G1 := Graph::residualGraph(G, flow):
Graph::printGraphInformation(G1);
```

```
Vertices: [1, 2, 3, q, s]
Edges: [[2, 1], [3, 1], [3, 2], [s, 3], [1, q], [1, 2], [2, 3]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: [1, 2] = 3, [2, 3] = 1, [2, 1] = 1, [3, 1] = 4, [3, 2] = 1, \
[s, 3] = 5, [1, q] = 5 (other existing edges have no weight)
Edge costs: [1, 2] = 1, [2, 3] = 1, [2, 1] = -3, [3, 1] = 0, [3, 2] = -1, \
[s, 3] = 0, [1, q] = 0 (other existing edges have costs zero)
Adjacency list (out): 1 = [2, q], 2 = [1, 3], 3 = [1, 2], q = [], s = [3]
Adjacency list (in): 1 = [2, 3], 2 = [1, 3], 3 = [2, s], q = [1], s = []
Graph is directed.
```

Edge Weights contain the residual graph with all the flows. Edge Costs show the flow that was substracted or added. For example edge [1, 2] had weight 4. After a flow of 3 was sent over it, the residual edge [2, 1] contains the flow of -3 and the residual edge [1, 2] contains the flow of 1. Since the negative flow of the reverted edge plus the flow of the

edge in the residual graph have to sum up to the flow it shows that the flow is calculated correctly. (- (- 3) + 1 = 4)

```
G1 := Graph::residualGraph(G, flow, Extended):
Graph::printGraphInformation(G1);
```

```
Vertices: [1, 2, 3, q, s]
Edges: [[2, 1], [3, 1], [3, 2], [s, 3], [1, q], [1, 2], [1, 3], [2, 3], [3\
, s], [q, 1]]
Vertex weights: no vertex weights.
Edge descriptions: no edge descriptions.
Edge weights: [q, 1] = 5, [1, 2] = 4, [1, 3] = 4, [2, 3] = 2, [3, s] = 5, \
[2, 1] = -4, [3, 1] = -4, [3, 2] = -2, [s, 3] = -5, [1, q] = -5 (other exi\
sting edges have no weight)
Edge costs: [1, 2] = 3, [1, 3] = 0, [2, 3] = 1, [3, s] = 0, [q, 1] = 0, [2\
, 1] = 1, [3, 1] = 4, [3, 2] = 1, [s, 3] = 5, [1, q] = 5 (other existing e\
dges have costs zero)
Adjacency list (out): 1 = [2, 3, q], 2 = [1, 3], 3 = [1, 2, s], q = [1], s\
 = [3]
Adjacency list (in): 1 = [2, 3, q], 2 = [1, 3], 3 = [1, 2, s], q = [1], s \
= [3]
Graph is directed.
```

Parameters

`G`

Graph

`flow`

The predefined flow

Options

`Extended`

Include edges with zero capacities

Return Values

`Graph`

Graph::revert

Reverts the edges of a graph.

Syntax

```
Graph::revert(G)
```

Description

`Graph::revert(G)` returns a graph in which all edges [u, v] and their properties belong to edges [v, u].

`Graph::revert` overloads the system function `revert`.

Examples

Example 1

First, a circle graph is defined with some additional settings:

```
G1 := Graph::createCircleGraph(3):
G1 := Graph::setEdgeWeights(G1, [[1, 2]], [20]):
G1 := Graph::setEdgeCosts(G1, [[1, 2]], [20]):
G1 := Graph::setEdgeDescriptions(G1, [[1, 2]], ["Shortcut"]):
Graph::printGraphInformation(G1)
```

```
Vertices: [1, 2, 3]
Edges: [[1, 2], [2, 3], [3, 1]]
Vertex weights: no vertex weights.
Edge descriptions: [1, 2] = "Shortcut"
Edge weights: [1, 2] = 20 (other existing edges have no weight)
Edge costs: [1, 2] = 20 (other existing edges have costs zero)
Adjacency list (out): 1 = [2], 2 = [3], 3 = [1]
Adjacency list (in): 1 = [3], 2 = [1], 3 = [2]
Graph is directed.
```

Now we revert the graph:

```
G2 := revert(G1):
Graph::printGraphInformation(G2)
```

```
Vertices: [1, 2, 3]
Edges: [[1, 3], [2, 1], [3, 2]]
Vertex weights: no vertex weights.
Edge descriptions: [2, 1] = "Shortcut"
Edge weights: [2, 1] = 20 (other existing edges have no weight)
Edge costs: [2, 1] = 20 (other existing edges have costs zero)
Adjacency list (out): 1 = [3], 2 = [1], 3 = [2]
Adjacency list (in): 1 = [2], 2 = [3], 3 = [1]
Graph is directed.
```

Parameters

G

A graph

Return Values

Graph

Graph::setEdgeCosts

Assigns edge costs to edges.

Syntax

```
Graph::setEdgeCosts(G, Edge, EdgeCosts, <OnlySpecifiedEdges>)
```

Description

`Graph::setEdgeCosts(G, Edges, EdgeCosts)` returns a graph where `Edges` have the edge costs `EdgeCosts`.

Note: If *OnlySpecifiedEdges* is stated and an undirected graph is to be changed, only the edges specified are used and not the inverted ones. For example if a call `Graph::setEdgeCosts(G, [[u,v]], [1])` is invoked, only the edge [*u*, *v*] gets 1. The edge [*v*, *u*] will not be changed.

Note: The substitute `None` can be used when a specified edge should not get the assigned costs.

Examples

Example 1

First lets define a graph without edge costs:

```
G1 := Graph::createCircleGraph(3):
Graph::getEdgeCosts(G1)
```

```
FAIL
```

`FAIL` was returned, because no edge costs werde defined.

```
Graph::getEdges(G1);
G1 := Graph::setEdgeCosts(G1, [[1, 2], [3, 1]], [5, 1/2]):
Graph::getEdgeCosts(G1)
```

$$[[1, 2], [2, 3], [3, 1]]$$

[1, 2]	5
[3, 1]	$\frac{1}{2}$

The first output shows all the edges and the second one the assigned edge costs.

```
G1 := Graph::setEdgeCosts(G1, [[2, 3]], [infinity]):
Graph::getEdgeCosts(G1)
```

[1, 2]	5
[3, 1]	$\frac{1}{2}$
[2, 3]	∞

It is easy to see that only the edge cost of `[2,3]` was changed.

Example 2

First lets define a graph without edge costs:

```
G1 := Graph::createCompleteGraph(3):
Graph::getEdgeCosts(G1)
```

FAIL

`FAIL` was returned, because no edge costs werde defined.

```
Graph::getEdges(G1);
G2 := Graph::setEdgeCosts(G1, [[1, 2], [3, 1]], [5, 1/2]):
Graph::getEdgeCosts(G2)
```

[[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]

[1, 2]	5
[2, 1]	5
[1, 3]	$\frac{1}{2}$
[3, 1]	$\frac{1}{2}$

The first output shows all the edges (the graph is undirected !) and the second one the assigned edge costs. Not only the specified edges were set, but also the reverted edges.

```
Graph::getEdges(G1);
G2 := Graph::setEdgeCosts(G1, [[1, 2], [3, 1]], [5, 1/2],
                          OnlySpecifiedEdges):
Graph::getEdgeCosts(G2)
```

[[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]

[1, 2]	5
[3, 1]	$\frac{1}{2}$

It is easy to see that only the specified edge costs were changed and not the reverted edges, too.

Example 3

There exist also the possibility to set the costs via a table instead of a list.

```
tbl := table([1, 2] = 15, [1, 3] = 20):
G2 := Graph::createCompleteGraph(3):

G2 := Graph::setEdgeCosts(G2, [[1, 2], [3, 1]], tbl):
Graph::getEdgeCosts(G2)
```

[1, 2]	15
[2, 1]	15
[1, 3]	20
[3, 1]	20

And again, but this time only the specified edges:

```
tbl := table([1, 2] = 15, [1, 3] = 20):
G2 := Graph::createCompleteGraph(3):

G2 := Graph::setEdgeCosts(G2, [[1, 2], [3, 1]], tbl,
                          OnlySpecifiedEdges):
Graph::getEdgeCosts(G2)
```

[1, 2]	15
[3, 1]	20

Parameters

`G`

A graph

`Edge`

A list of one or more edges

`EdgeCosts`

A list of one or more numbers, or a table consisting of the edges with their costs.

Options

`OnlySpecifiedEdges`

Only the edges specified in `Edge` will be set.

Return Values

New graph with the corrected edge costs.

Graph::setEdgeDescriptions

Assigns edge Descriptions to edges.

Syntax

```
Graph::setEdgeDescriptions(G, Edge, EdgeDescriptions, <OnlySpecifiedEdges>)
```

Description

`Graph::setEdgeDescriptions(G, Edges, EdgeDescriptions)` returns a graph where `Edges` have the edge descriptions `EdgeDescriptions`.

Note: If *OnlySpecifiedEdges* is stated and an undirected graph is to be changed, only the edges specified are used and not the inverted ones. For example if a call `Graph::setEdgeDescriptions(G, [[u,v]], [1])` is invoked, only the edge [u, v] gets 1. The edge [v, u] will not be changed.

Note: The substitute `None` can be used when a specified edge should not get the assigned description.

Examples

Example 1

First lets define a graph without edge descriptions:

```
G1 := Graph::createCircleGraph(3):
Graph::getEdgeDescriptions(G1)
```

```
FAIL
```

`FAIL` was returned, because no edge descriptions werde defined.

```
Graph::getEdges(G1);
G1 := Graph::setEdgeDescriptions(G1, [[1, 2],[3, 1]],
                                 ["Route 66", "Speedway"]):
Graph::getEdgeDescriptions(G1)
```

```
[[1, 2], [2, 3], [3, 1]]
```

[1, 2]	"Route 66"
[3, 1]	"Speedway"

The first output shows all the edges and the second one the assigned edge descriptions.

```
G1 := Graph::setEdgeDescriptions(G1, [[2, 3]], ["Shortcut"]):
Graph::getEdgeDescriptions(G1)
```

[1, 2]	"Route 66"
[3, 1]	"Speedway"
[2, 3]	"Shortcut"

It is easy to see that only the edge description of `[2,3]` was changed.

Example 2

First lets define a graph without edge Descriptions:

```
G1 := Graph::createCompleteGraph(3):
Graph::getEdgeDescriptions(G1)
```

```
FAIL
```

`FAIL` was returned, because no edge descriptions werde defined.

```
Graph::getEdges(G1);
G2 := Graph::setEdgeDescriptions(G1, [[1, 2], [3, 1]],
                                 ["Route 66", "Speedway"]):
```

```
Graph::getEdgeDescriptions(G2)
```

```
[[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]
```

[1, 2]	"Route 66"
[2, 1]	"Route 66"
[1, 3]	"Speedway"
[3, 1]	"Speedway"

The first output shows all the edges (the graph is undirected !) and the second one the assigned edge Descriptions. Not only the specified edges were set, but also the reverted edges.

```
Graph::getEdges(G1);
G2 := Graph::setEdgeDescriptions(G1, [[1, 2], [3, 1]],
                                 ["Route 66", "Speedway"],
                                 OnlySpecifiedEdges):
Graph::getEdgeDescriptions(G2)
```

```
[[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]
```

[1, 2]	"Route 66"
[3, 1]	"Speedway"

It is easy to see that only the specified edge Descriptions were changed and not the reverted edges, too.

Example 3

There exist also the possibility to set the Descriptions via a table instead of a list.

```
tbl := table([1, 2] = "Highway", [1, 3] = "Road to nowhere"):
G2 := Graph::createCompleteGraph(3):
Graph::getEdgeDescriptions(G2):

G2 := Graph::setEdgeDescriptions(G2, [[1, 2], [3, 1]], tbl):
Graph::getEdgeDescriptions(G2)
```

[1, 2]	"Highway"
[2, 1]	"Highway"
[1, 3]	"Road to nowhere"
[3, 1]	"Road to nowhere"

And again, but this time only the specified edges:

```
tbl := table([1, 2] = "Highway", [1, 3] = "Road to nowhere"):
G2 := Graph::createCompleteGraph(3):
Graph::getEdgeDescriptions(G2):
G2 := Graph::setEdgeDescriptions(G2, [[1, 2], [3, 1]], tbl,
                                 OnlySpecifiedEdges):
Graph::getEdgeDescriptions(G2)
```

[1, 2]	"Highway"
[3, 1]	"Road to nowhere"

Parameters

`G`

A graph

`Edge`

A list of one or more edges

`EdgeDescriptions`

A list of one or more numbers, or a table consisting of the edges with their descriptions.

Options

`OnlySpecifiedEdges`

Only the edges specified in `Edge` will be set.

Return Values

New graph with the corrected edge Descriptions.

Graph::setEdgeWeights

Assigns edge weights to edges.

Syntax

```
Graph::setEdgeWeights(G, Edge, EdgeWeights, <OnlySpecifiedEdges>)
```

Description

`Graph::setEdgeWeights(G, Edges, EdgeWeights)` returns a graph where `Edges` have the edge weights `EdgeWeights`.

Note: If *OnlySpecifiedEdges* is stated and an undirected graph is to be changed, only the edges specified are used and not the inverted ones. For example if a call `Graph::setEdgeWeights(G, [[u,v]], [1])` is invoked, only the edge [*u*, *v*] gets 1. The edge [*v*, *u*] will not be changed.

Note: The substitute `None` can be used when a specified edge should not get the assigned weights.

Examples

Example 1

How to set edge weights with a list:

```
G1 := Graph::createCircleGraph(3):
Graph::getEdgeWeights(G1)
```

```
FAIL
```

`FAIL` was returned, because no edge weights were defined.

```
Graph::getEdges(G1);
G1 := Graph::setEdgeWeights(G1, [[1, 2], [3, 1]], [5, 1/2]):
Graph::getEdgeWeights(G1)
```

$$[[1, 2], [2, 3], [3, 1]]$$

[1, 2]	5
[3, 1]	$\frac{1}{2}$

The first output shows all the edges and the second one the assigned edge weights.

```
G1 := Graph::setEdgeWeights(G1, [[2, 3]], [infinity]):
Graph::getEdgeWeights(G1)
```

[1, 2]	5
[3, 1]	$\frac{1}{2}$
[2, 3]	∞

It is easy to see that only the edge weight of `[2, 3]` was changed.

Example 2

How to set edge weights with a table:

```
G1 := Graph::createCompleteGraph(3):
Graph::getEdgeWeights(G1)
```

FAIL

`FAIL` was returned, because no edge weights were defined.

```
Graph::getEdges(G1);
G2 := Graph::setEdgeWeights(G1, [[1, 2], [3, 1]], [5, 1/2]):
Graph::getEdgeWeights(G2)
```

[[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]

[1, 2]	5
[2, 1]	5
[1, 3]	$\frac{1}{2}$
[3, 1]	$\frac{1}{2}$

The first output shows all the edges (the graph is undirected !) and the second one the assigned edge weights. Not only the specified edges were set, but also the reverted edges.

```
Graph::getEdges(G1);
G2 := Graph::setEdgeWeights(G1, [[1, 2], [3, 1]], [5, 1/2],
                            OnlySpecifiedEdges):
Graph::getEdgeWeights(G2)
```

[[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]

[1, 2]	5
[3, 1]	$\frac{1}{2}$

It is easy to see that only the specified edge Weights were changed and not the reverted edges, too.

Example 3

There exist also the possibility to set the weights via a table instead of a list.

```
tbl := table([1, 2] = 15, [1, 3] = 20):
G2 := Graph::createCompleteGraph(3):

G2 := Graph::setEdgeWeights(G2, [[1, 2], [3, 1]], tbl):
Graph::getEdgeWeights(G2)
```

[1, 2]	15
[2, 1]	15
[1, 3]	20
[3, 1]	20

And again, but this time only the specified edges:

```
tbl := table([1, 2] = 15, [1, 3] = 20):
G2 := Graph::createCompleteGraph(3):

G2 := Graph::setEdgeWeights(G2, [[1, 2], [3, 1]], tbl,
                            OnlySpecifiedEdges):
Graph::getEdgeWeights(G2)
```

[1, 2]	15
[3, 1]	20

Parameters

`G`

A graph

`Edge`

A list of one or more edges

`EdgeWeights`

A list of one or more numbers, or a table consisting of the edges with their weights.

Options

`OnlySpecifiedEdges`

Only the edges specified in `Edge` will be set.

Return Values

New graph with the corrected edge weights.

Graph::setVertexWeights

Assigns vertex weights to vertices.

Syntax

```
Graph::setVertexWeights(G, Vertex, VertexWeights)
```

Description

`Graph::setVertexWeights(G, Vertex, VertexWeights)` returns a graph where the vertices in `Vertex` have the vertex weights `VertexWeights`.

Note: The substitute `None` can be used when a specified edge should not get the assigned weights.

Examples

Example 1

How to set vertex weights with a list:

```
G1 := Graph::createCircleGraph(3):
Graph::getVertexWeights(G1)
```

```
FAIL
```

`FAIL` was returned, because no vertex weights were defined.

```
Graph::getVertices(G1);
G1 := Graph::setVertexWeights(G1, [1, 3], [5, 1/2]):
Graph::getVertexWeights(G1)
```

```
[1, 2, 3]
```

1	5
3	$\frac{1}{2}$

The first output shows all vertices and the second the assigned vertex weights.

Example 2

How to set vertex weights with a table:

```
G1 := Graph::createCompleteGraph(3):
Graph::getVertexWeights(G1)
```

FAIL

FAIL was returned, because no vertex weights were defined.

```
G2 := Graph::createCompleteGraph(3):
tbl := table(1 = 15, 3 = 20):
G2 := Graph::setVertexWeights(G2, [1, 3], tbl):
Graph::getVertexWeights(G2)
```

1	15
3	20

Parameters

G

A graph

Vertex

A list of one or more vertices

VertexWeights

A list of one or more numbers, or a table consisting of the vertices with their weights.

Return Values

New graph with the corrected vertex weights.

Graph::shortestPathAllPairs

Shortest paths from and to all vertices

Syntax

```
Graph::shortestPathAllPairs(G, <SearchFor = Weights | Costs>)
```

Description

`Graph::shortestPathAllPairs(G)` returns a table with all paths between all vertices.

`Graph::shortestPathAllPairs(G, SearchFor=Costs)` returns a `table` with all paths according to the edge costs.

`Graph::shortestPathAllPairs(G, SearchFor=Weights)` returns a `table` with all paths according to the edge weights. (Default)

Examples

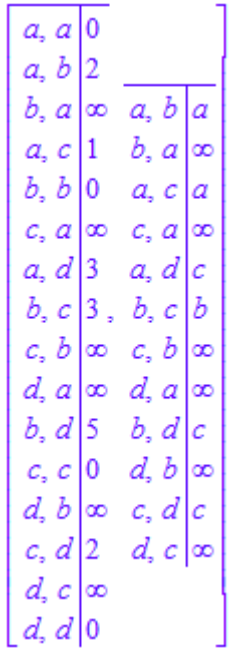
Example 1

A small graph to be used for the algorithms:

```
G := Graph([a, b, c, d], [[a, b], [a, c], [b, c], [c, d]],
           EdgeWeights = [2, 1, 3, 2],
           EdgeCosts = [1, 3,1, 2],
           Directed):
```

Now the shortest path between all vertices is found according to the edge weights, because no specification was given and defaults are used.

```
Graph::shortestPathAllPairs(G)
```



a, a	0
a, b	2
b, a	∞
a, c	1
b, b	0
c, a	∞
a, d	3
b, c	3 ,
c, b	∞
d, a	∞
b, d	5
c, c	0
d, b	∞
c, d	2
d, c	∞
d, d	0

a, b	a
b, a	∞
a, c	a
c, a	∞
a, d	c
b, c	b
c, b	∞
d, a	∞
b, d	c
d, b	∞
c, d	c
d, c	∞

The interpretation of the table is as follows:

The first table holds each path: (FromVertex, ToVertex) = weight/cost. The second table is a bit more tricky. The left hand side again is the path itself. On the right hand side though, the vertex that was found before the final vertex was reached is stated. If for example the path from `a` to `d` is to be found with all vertices that are used within this path it is done in the following way: First take the path itself `(a, d)`. The predecessor is `c`. Now have a look for the path `(a, c)`. It's predecessor is `a`. Since the predecessor equals the first vertex in the path to be found, the search is over and the path `a -> c -> d` is found. To search the graph for costs the option `SearchFor=Costs` has to be added.

```
Graph::shortestPathAllPairs(G, SearchFor = Costs)
```

a, a	0
a, b	1
b, a	∞
a, c	2
b, b	0
c, a	∞
a, d	4
b, c	1 ,
c, b	∞
d, a	∞
b, d	3
c, c	0
d, b	∞
c, d	2
d, c	∞
d, d	0

a, b	a
b, a	∞
a, c	b
c, a	∞
a, d	c
b, c	b
c, b	∞
d, a	∞
b, d	c
d, b	∞
c, d	c
d, c	∞

Example 2

Now the weights of the graph are changed, so that negative edge weights are assigned. You will see that this does not influence the correctness of the results the algorithm returns (like for example Dijkstra).

```
G := Graph([a, b, c, d], [[a, b], [a, c], [b, c], [c, d]],
           EdgeWeights = [2, 1, 3, 2],
           EdgeCosts = [1, 3,1, 2],
           Directed):

G := Graph::setEdgeWeights(G, Graph::getEdges(G), [2, 1, -3, 2]):

Graph::shortestPathAllPairs(G)
```

a, a	0			
a, b	2			
b, a	∞	a, b	a	
a, c	−1	b, a	∞	
b, b	0	a, c	b	
c, a	∞	c, a	∞	
a, d	1	a, d	c	
b, c	−3,	b, c	b	
c, b	∞	c, b	∞	
d, a	∞	d, a	∞	
b, d	−1	b, d	c	
c, c	0	d, b	∞	
d, b	∞	c, d	c	
c, d	2	d, c	∞	
d, c	∞			
d, d	0			

Parameters

G

Graph

Options

SearchFor

Defines whether the weights of the graph are considered or the costs. Default is `Weights`.

Return Values

List consisting of two tables. The first table holds the sum of the path weights or costs and the second the predecessors for every path (to find the complete path).

Algorithms

The algorithm is also known as Floyd-Warshall or Roy-Warshall algorithm. The idea behind it is to solve the problem by continuous matrix multiplication. he only difference is that Floyd uses the assignment $a_{i,j} := min(a_{i,j}, a_{i,k} + a_{k,j})$.

References

[1] Ahuja, Magnanti, Orlin: Network Flows, Prentice-Hall, 1993 Section 5.6

Graph::shortestPathSingleSource

Shortest paths from one single vertex

Syntax

```
Graph::shortestPathSingleSource(G, StartVertex, <EndVertex = v>, <SearchWith = Dijkstra
```

Description

`Graph::shortestPathSingleSource(G, StartVertex=vertex)` gives the length of a shortest path from `StartVertex` to every other vertex in `G`.

`Graph::shortestPathSingleSource(G, StartVertex=sv)` returns a table with all paths from *sv* to any other.

`Graph::shortestPathSingleSource(G, StartVertex=sv, ReturnAsGraph)` returns a `table` with all paths from *sv* to any other because EndVertex has to be set in order to get a Graph as return value.

With `Graph::shortestPathSingleSource(G, StartVertex=sv, EndVertex=ev, SearchWith=Dijkstra, SearchFor=Costs)` returns a `table` from vertex *sv* to vertex *ev* according to Dijkstra which used the edge-costs for its algorithm.

Note: Using Dijkstra for shortest path can be erroneus if the graph contains negative edges.

Note: If `ReturnAsGraph` is stated and `EndVertex` omitted, a table is returned nevertheless.

Examples

Example 1

A small graph to be used for the algorithms:

```
G := Graph([a, b, c, d], [[a, b], [a, c], [b, c], [c, d]],
           EdgeWeights = [2, 1, 3, 2],
           EdgeCosts = [1, 3, 1, 2], Directed):
```

Now the shortest path is found according to Bellman using edge weights, because no specification was given and defaults are used:

```
Graph::shortestPathSingleSource(G, StartVertex = [a])
```

$$\left[\begin{array}{c|c} a & 0 \\ b & 2 \\ c & 1 \\ d & 3 \end{array}, \begin{array}{c|c} b & a \\ c & a \\ d & c \end{array}\right]$$

To search the graph with Bellman for costs the option `SearchFor=Costs` has to be added:

```
Graph::shortestPathSingleSource(G, StartVertex = [a],
                                SearchFor=Costs)
```

$$\left[\begin{array}{c|c} a & 0 \\ b & 1 \\ c & 2 \\ d & 4 \end{array}, \begin{array}{c|c} b & a \\ c & b \\ d & c \end{array}\right]$$

Example 2

Now the weights of the graph are changed, so that negative edge weights are assigned. After this the procedure is called again with Bellman and afterwards with Dijkstra to compare the results:

```
G := Graph([a, b, c, d], [[a, b], [a, c], [b, c], [c, d]],
           EdgeWeights = [2, 1, 3, 2],
           EdgeCosts = [1, 3, 1, 2], Directed):

G := Graph::setEdgeWeights(G, Graph::getEdges(G),
                           [2, 1, -3, 2]):

Graph::shortestPathSingleSource(G, StartVertex = [a],
```

```
                                    SearchWith = Bellman),
Graph::shortestPathSingleSource(G, StartVertex = [a],
                                    SearchWith = Dijkstra)
```

$$\left[\begin{array}{c|c} a & 0 \\ b & 2 \\ c & -1 \\ d & 1 \end{array}, \begin{array}{c|c} b & a \\ c & b \\ d & c \end{array}\right], \left[\begin{array}{c|c} a & 0 \\ b & 2 \\ c & -1 \\ d & 3 \end{array}, \begin{array}{c|c} b & a \\ c & a \\ d & c \end{array}\right]$$

This is a typical example where Dijkstra can make a mistake because he does not correct earlier solutions (a so called greedy strategy). Although vertex `c` gets the correct value `-1`, at the time `d` got the value `3`, vertex `c` still held the value `1`. This happens because Dijkstra first searches the best solutions (a->c = 1) then traverses further (c->d = 1 + 2 = 3). In spite of changing the value of vertex `c` the value for d is never to be changed again (because no other path ever reaches it again):

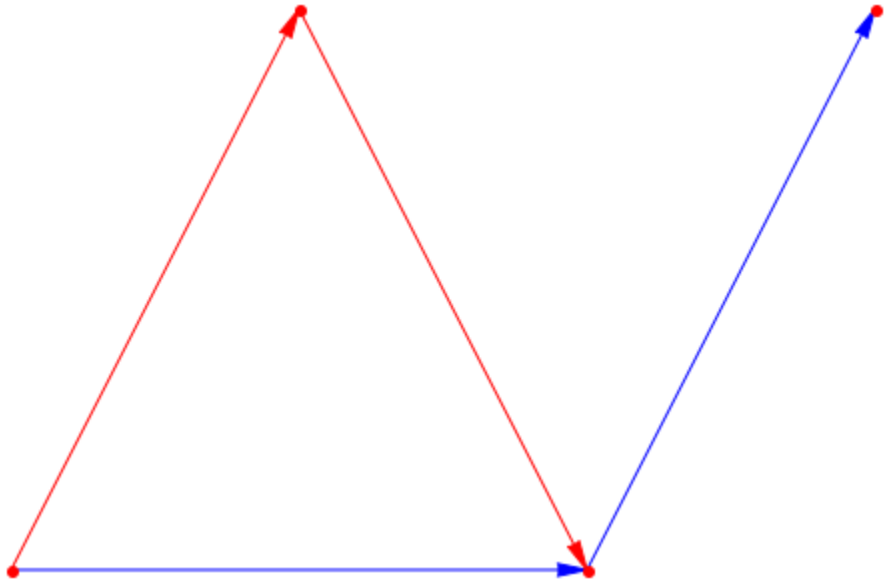
It might be interesting to see a shortest path inside the graph. Here are two steps that accomplish this task:

Fist step (creation of a shortest path graph [in this case with Dijkstra]):

```
dijk := Graph::shortestPathSingleSource(G, StartVertex = [a],
                                        EndVertex = [d],
                                        SearchWith = Dijkstra,
                                        ReturnAsGraph):
```

Second step (combination of the graphs using `plotGridGraph`):

```
plot(Graph::plotGridGraph(G, VerticesPerLine = 4,
       VertexOrder = [None, b, None, d, a, None, c, None],
       VertexColor = RGB::Red,
       SpecialEdges = Graph::getEdges(dijk),
       SpecialEdgeColor = RGB::Blue))
```



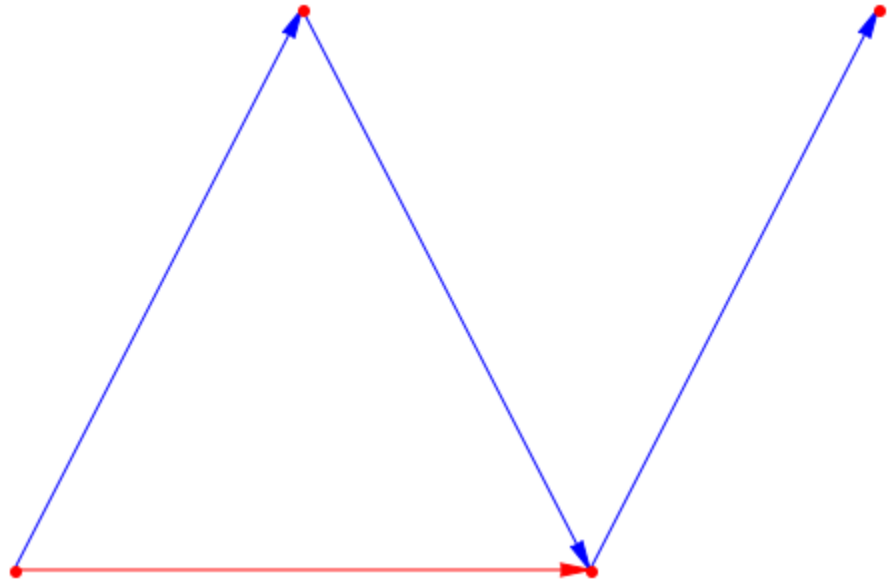
The same with Bellman to show the differences:

Fist step (creation of a shortest path graph [in this case with Dijkstra]):

```
bellm := Graph::shortestPathSingleSource(G, StartVertex = [a],
                                         EndVertex = [d],
                                         SearchWith = Bellman,
                                         ReturnAsGraph):
```

Second step (combination of the graphs using `plotGridGraph`):

```
plot(Graph::plotGridGraph(G, VerticesPerLine = 4,
       VertexOrder = [None, b, None, d, a, None, c, None],
       VertexColor = RGB::Red,
       SpecialEdges = Graph::getEdges(bellm),
       SpecialEdgeColor = RGB::Blue))
```



Parameters

`G`

Graph

`vertex`

A vertex in G

Options

`EndVertex`

Specifies a single vertex to which the shortest path is to be found.

`SearchWith`

Defines the algorithm to use. Dijkstra can be erroneus if the graph consists of negative edges. Default is `Bellman`

`SearchFor`

Defines whether the weights of the graph are considered or the costs. Default is `Weights`.

`ReturnAsGraph`

If stated and EndVertex is set, the path is returned as a Graph. If stated and EndVertex is not set, this option is omitted.

Return Values

Either a list consisting of two tables or a Graph. The first table holds the weights or cost for each vertex and the second the predecessors for every vertex (to find the path)

Algorithms

Both, Bellman and Dijkstra expect a Graph without negative circles. Only Dijkstra may return erroneous results when negative edges (either weights or costs) are specified.

The Bellman algorithm originated from: Ahuja, Magnanti, Orlin: Graph Flows, Prentice-Hall, 1993 Section 5.4

Graph::stronglyConnectedComponents

Finds the strongly connected components

Syntax

```
Graph::stronglyConnectedComponents(G)
```

Description

`Graph::stronglyConnectedComponents(G)` finds the strongly connected components of `G`

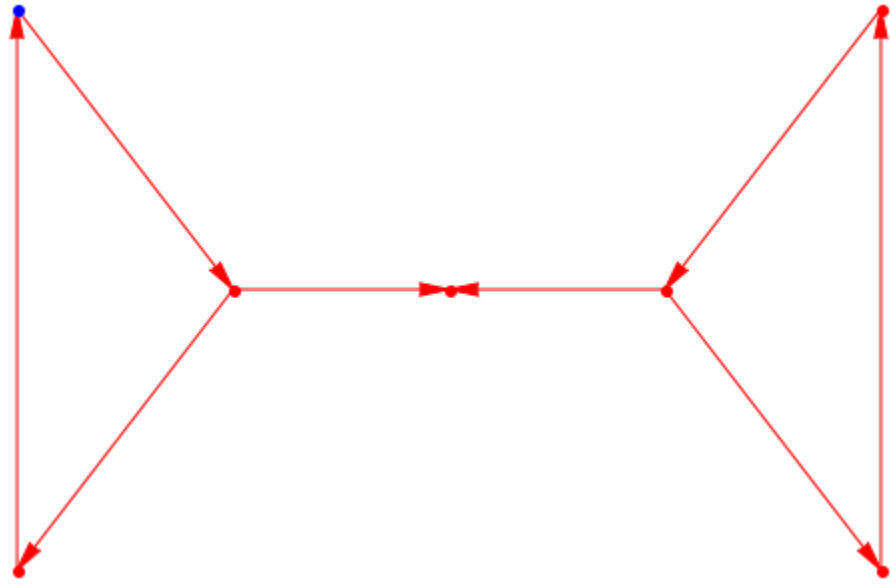
`Graph::stronglyConnectedComponents` returns all the strongly connected components of a graph. Single vertices form a component of themselves.

Examples

Example 1

Two obvious components pointing to a single vertex:

```
G1 := Graph([a, b, c, d, e, f, g],
            [[a, b], [b, g], [g, a], [b, c],
             [d, c], [e, d], [d, f], [f, e]], Directed):

plot(Graph::plotGridGraph(G1,
       VertexOrder = [a,    None, None, None, e,
                      None, b,    c,    d,    None,
                      g,    None, None, None, f],
       VerticesPerLine=5))
```



The graphical output reveals the two "big" components `[a, b, g]` and `[d, e, f]`. The single vertex `[c]` forms a component of itself:

```
G2 := Graph::stronglyConnectedComponents(G1)
```

$$[\text{Graph}(\ldots), \text{Graph}(\ldots), \text{Graph}(\ldots)]$$

A list containing three Graphs is returned. Now we find out which vertices belong to each component:

```
Graph::getVertices(op(G2, 1))
```

$$[d, e, f]$$

```
Graph::getVertices(op(G2, 2))
```

$$[a, b, g]$$

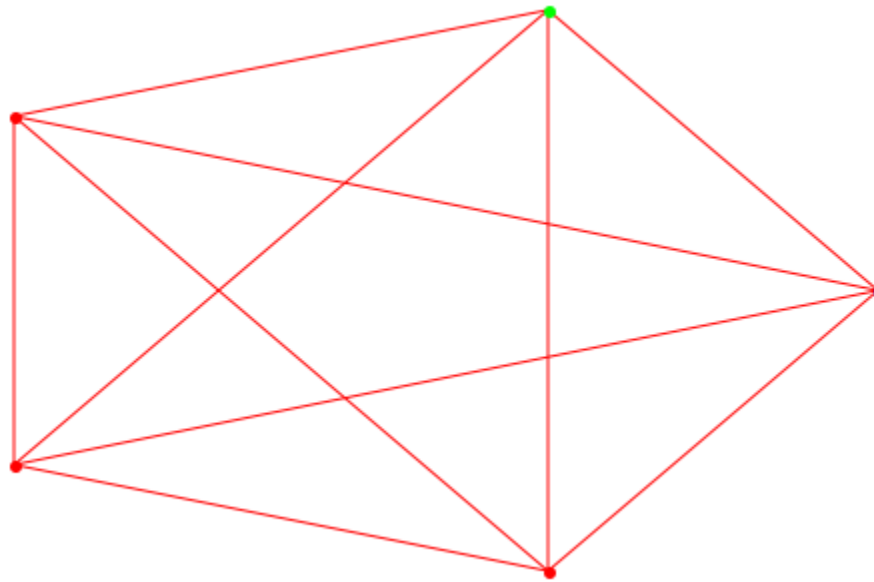
```
Graph::getVertices(op(G2, 3))
```

$$[c]$$

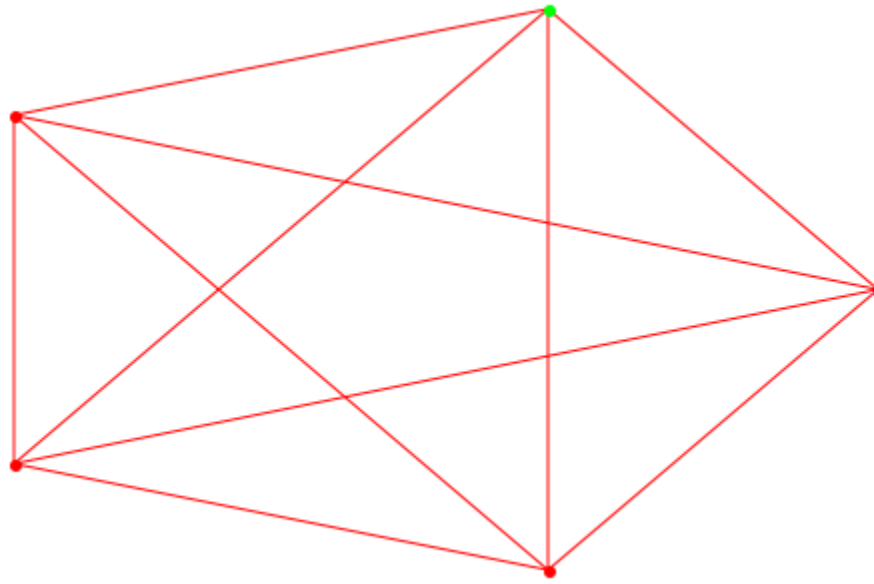
Example 2

A complete graph is returned as a single component:

```
G3 := Graph::createCompleteGraph(5):
plot(Graph::plotCircleGraph(G3))
```



```
G4 := Graph::stronglyConnectedComponents(G3):
plot(Graph::plotCircleGraph(op(G4)))
```



It was necessary to use op(G4), because G4 is a list containing a graph!

Parameters

G

A graph

Return Values

List of graphs containing the strongly connected components.

Graph::topSort

Topological sorting of the vertices

Syntax

```
Graph::topSort(G)
```

Description

`Graph::topSort(G)` computes a topological sorting of the graph `G`, i.e., a numbering T of the vertices, such that $T_i < T_j$ whenever there is an edge $[i, j]$ in the graph. Single vertices are positioned at the beginning.

`Graph::topSort` returns a list containing two tables. The first table holds the ordering of the vertices. The second table shows the predecessors of each vertex. If several vertex u_i precede a vertex v, the first vertex in the ordering of u_i is the predecessor of v. If no predecessor exist, the value will be *infinity*.

Note: If `G` contains any cycle then a topological sorting does not exist and the call of `Graph::topSort` results in an error.

Examples

Example 1

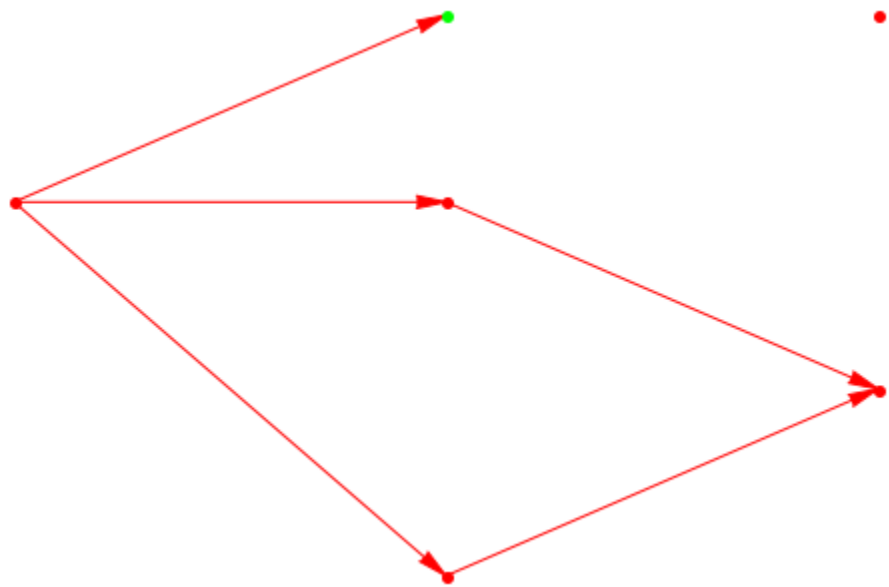
A "butterfly" graph that is decomposed in three strongly connected components:

```
G1 := Graph([a, b, c, d, e, f],
            [[a, b], [a, c], [a, d], [c, e], [d, e]],
            Directed):
Graph::topSort(G1)
```

$$\begin{bmatrix} 1 & f \\ 2 & a \\ 3 & b \\ 4 & c \\ 5 & d \\ 6 & e \end{bmatrix}, \quad \begin{bmatrix} a & a \\ b & a \\ c & a \\ d & a \\ e & c \\ f & \infty \end{bmatrix}$$

The first table shows the ordering of the vertices. The left side holds the order for each vertex, whereas the right side holds the name of the vertex. The second table shows the predecessors of each vertex. If no predecessor exist, the right side holds *infinity*. Otherwise the right side holds the vertex that is the direct predecessor of the vertex on the left side. To see how the graph looks a graphical plotting helps:

```
plot(Graph::plotGridGraph(G1,
         VertexOrder = [None, b,    f,
                        a,    c,    None,
                        None, None, e,
                        None, d,    None],
         VerticesPerLine=3))
```



Parameters

G

A graph

Return Values

List containing two tables.

groebner – Gröbner bases

groebner::dimension
groebner::eliminate
groebner::gbasis
groebner::normalf
groebner::spoly
groebner::stronglyIndependentSets

groebner::dimension

Dimension of the affine variety generated by polynomials

Syntax

```
groebner::dimension(polys, <order>)
```

Description

`groebner::dimension(polys)` computes the dimension of the affine variety generated by the polynomials in the set or list `polys`.

The rules laid down in the introduction to the groebner package concerning the polynomial types and the ordering apply.

The polynomials in the list `polys` must all be of the same type. In particular, do not mix polynomials created via `poly` and polynomial expressions!

Examples

Example 1

An example from the book of Cox, Little and O'Shea (see below):

```
groebner::dimension([y^2*z^3, x^5*z^4, x^2*y*z^2])
```

```
2
```

Parameters

`polys`

A list or set of polynomials or polynomial expressions of the same type. The coefficients in these polynomials and polynomial expressions can be arbitrary arithmetical expressions.

`order`

One of the identifiers `DegInvLexOrder`, `DegreeOrder`, and `LexOrder`, or a user-defined term ordering of type `Dom::MonomOrdering`. The default ordering is `DegInvLexOrder`.

Return Values

Nonnegative integer

Algorithms

First, the Gröbner basis of the given polynomials with respect to the given monomial ordering is computed using `groebner::gbasis`. This Gröbner basis is then used to compute the dimension of the affine variety generated by the polynomials.

References

The implemented algorithm is described in Cox, Little, O'Shea: “Ideals, Varieties and Algorithms”, Springer, 1992, Chapter 9.

See Also

MuPAD Functions

`groebner::gbasis` | `poly`

groebner::eliminate

Eliminate variables

Syntax

```
groebner::eliminate(sys, vars)
```

Description

`groebner::eliminate(sys, vars)` returns a list of polynomial expressions obtained by eliminating the elements of `vars` from `sys`. In other words, the return value does not contain the variables in `vars`, every zero of the original system `sys` must be a zero of the return value, and every tuple of numbers that makes the return value zero can be extended to a solution of `sys`.

Examples

Example 1

Suppose that $x^2 + y = 0$ and $x + y = 0$, what does this imply for y?

```
groebner::eliminate({x^2 +y, y+x}, {x})
```

$$[y^2 + y]$$

We infer that for every pair (x, y) solving the system, y must satisfy $y^2 + y = 0$, that is, $y = 0$ or $y = -1$. Indeed:

```
solve({x^2 +y, y+x}, {x, y})
```

$$\{[x = 0, y = 0], [x = 1, y = -1]\}$$

Parameters

`sys`

A set or list of polynomial expressions. The coefficients in these polynomial expressions can be arbitrary arithmetical expressions.

`vars`

A set or list of identifiers

Return Values

List of polynomial expressions

Algorithms

`groebner::eliminate` proceeds by computing a lexical-order Gröbner basis. Hence the computation complexity grows fast when increasing the number of variables.

See Also

MuPAD Functions
`groebner::gbasis`

groebner::gbasis

Computation of a reduced Gröbner basis

Syntax

```
groebner::gbasis(polys, <order>, options)
```

Description

`groebner::gbasis(polys)` computes a reduced Gröbner basis of the ideal generated by the polynomials in the list `polys`.

The rules laid down in the introduction to the groebner package concerning the polynomial types and the ordering apply.

The polynomials in the list `polys` must all be of the same type. In particular, do not mix polynomials created via `poly` and polynomial expressions!

The ordering strategy indicated by `Reorder` is used automatically when polynomial expressions are used.

Examples

Example 1

We demonstrate the effect of various input formats. First, we use polynomial expressions to define the polynomial ideal. The Gröbner basis is returned as a list of polynomial expressions:

```
groebner::gbasis([x^2 - y^2, x^2 + y], LexOrder)
```

$$[x^2 + y,\ x^4 - x^2]$$

Next, the same polynomials are defined via `poly`. Note that `poly` fixes the ordering of the variables.

```
groebner::gbasis([poly(x^2 - y^2, [x, y]),
                  poly(x^2 + y, [x, y])], LexOrder)
```

$$[\mathrm{poly}(x^2+y,[x,y]),\ \mathrm{poly}(y^2+y,[x,y])]$$

Changing the ordering of the variables in `poly` changes the lexicographical ordering. This results in a different basis:

```
groebner::gbasis([poly(x^2 - y^2, [y, x]),
                  poly(x^2 + y, [y, x])], LexOrder)
```

$$[\mathrm{poly}(y+x^2,[y,x]),\ \mathrm{poly}(x^4-x^2,[y,x])]$$

With `Reorder` the ordering of the variables may be changed internally:

```
groebner::gbasis([poly(x^2 - y^2, [x, y]),
                  poly(x^2 + y, [x, y])], LexOrder, Reorder)
```

$$[\mathrm{poly}(y+x^2,[y,x]),\ \mathrm{poly}(x^4-x^2,[y,x])]$$

Example 2

Polynomials over arbitrary fields are allowed. In particular, you can use the field of rational functions in some given variable(s):

```
F := Dom::Fraction(Dom::DistributedPolynomial([y])):
F::Name := "Q(y)":
groebner::gbasis(
[poly(y*z^2 + 1, [x, z], F),
poly((y^2 + 1)*x^2 - y - z^3, [x, z], F)])
```

$$\left[\mathrm{poly}\left(x^2+\frac{z}{y+y^3}-\frac{y}{y^2+1},[x,z],\mathrm{Q(y)}\right),\ \mathrm{poly}\left(z^2+\frac{1}{y},[x,z],\mathrm{Q(y)}\right)\right]$$

```
delete F:
```

Parameters

`polys`

A list or set of polynomials or polynomial expressions of the same type. The coefficients in these polynomials and polynomial expressions can be arbitrary arithmetical expressions. If `polys` are polynomials over an arbitrary domain, then their coefficients must be domain elements and the domain must be a field.

`order`

One of the identifiers `DegInvLexOrder`, `DegreeOrder`, and `LexOrder`, or a user-defined term ordering of type `Dom::MonomOrdering`. The default ordering is `DegInvLexOrder`.

Options

`Factor`

With this option, `groebner::gbasis` returns a set of lists, such that each list is the Gröbner basis of an ideal. The union of these ideals is a superset of the ideal given as input, and a subset of the radical of that ideal. In other words, it has the same variety (only the multiplicity of points can change).

`IgnoreSpecialCases`

With this option, `groebner::gbasis` handles all coefficients in all intermediate results as nonzero unless these coefficients are equal to zero for all parameter values. In other words, if the coefficients are rational functions of the free parameters, then results are correct on all of the parameter space except on an algebraic variety of lower dimension.

`Reorder`

With this option `groebner::gbasis` internally may change the lexicographical ordering of variables to decrease running time.

With this option the variables are sorted internally such that they have a “heuristic optimal” ordering. Consequently, the ordering of the variables in the output polynomials may differ from their ordering in the input polynomials. For details on the ordering strategy, see W. Boege, R. Gebauer und H. Kredel: “Some Examples for Solving Systems

of Algebraic Equations by Calculating Groebner Bases" im J. Symbolic Comp. (1986) Vol. 1, 83-98.

Re-ordering is always applied when polynomial expressions are used for input.

`Monic`

Option, specified as `Monic = N`

This option sets the normalizing routine to `N`. For every polynomial `f` in the polynomial ring, `N(f, o)` must return some associate of `f`, where `o` is the chosen order.

The method `N` should be chosen such that it produces simple output.

By default, `polylib::primpart` is used for polynomials with integer coefficients; other polynomials are divided by their leading coefficient.

`Order`

Option, specified as `Order = order`

This option is equivalent to passing `order` as an argument.

Return Values

List of polynomials. The output polynomials have the same type as the polynomials of the input list.

Algorithms

In most cases, `groebner::gbasis` computes the basis via the Buchberger algorithm with the "sugar" selection strategy being used.

References

For general information, see T. Becker and V. Weispfenning: "Gröbner Bases", Springer (1993). For details on the sugar selection strategy, see A. Giovini, T. Mora, G. Niesi,

L. Robbiano, C. Traverso: “One sugar cube, please — or Selection strategies in the Buchberger algorithm”, Proc. ISSAC '91, Bonn, 49-54 (1991).

See Also

MuPAD Functions

`poly`

groebner::normalf

Complete reduction modulo a polynomial ideal

Syntax

```
groebner::normalf(p, polys, <order>)
```

Description

`groebner::normalf(p, polys)` computes a normal form of the polynomial `p` by complete reduction modulo all polynomials in the list `polys`.

The rules laid down in the introduction to the groebner package concerning the polynomial types and the ordering apply.

The polynomials in the list `polys` must all be of the same type as `p`. In particular, do not mix polynomials created via `poly` and polynomial expressions.

Examples

Example 1

We consider the ideal generated by the following polynomials:

```
p1 := poly(x^2 - x + 2*y^2, [x,y]):
p2 := poly(x + 2*y - 1, [x,y]):
```

We compute the normal form of the following polynomial `p` modulo the ideal generated by `p1`, `p2` with respect to lexicographical ordering:

```
p :=  poly(x^2*y - 2*x*y + 1, [x,y]):
groebner::normalf(p, [p1, p2], LexOrder);
```

$$\mathrm{poly}(-2y^3+2y^2-y+1, [x, y])$$

Note that p1, p2 do not form a Gröbner basis. The corresponding Gröbner basis leads to a different normal form of p:

```
groebner::normalf(p, groebner::gbasis([p1, p2]), LexOrder)
```

$$\mathrm{poly}\left(-\frac{5\,y}{9}+1,\,[x,\,y]\right)$$

```
delete p1, p2, p:
```

Parameters

p

A polynomial or a polynomial expression. The coefficients in this polynomial and polynomial expression can be arbitrary arithmetical expressions.

polys

A list of polynomials of the same type as p. In particular, if p is a polynomial expression, polys must be a list of polynomial expressions.

order

One of the identifiers DegInvLexOrder, DegreeOrder, and LexOrder, or a user-defined term ordering of type Dom::MonomOrdering. The default ordering is DegInvLexOrder.

Return Values

Polynomial of the same type as the input polynomials. If polynomial expressions are used as input, then a polynomial expression is returned.

Algorithms

A polynomial g is a reduced form of a polynomial p modulo a list of polynomials p_1, …, p_n, if $g \equiv p$ and none of the leading terms of the p_i divides the leading term of p, or

if — for some i — g is a reduced form of p - $q\,p_i$, where q is the quotient of the leading monomial of p and the leading monomial of p_i. A reduced form always exists, but need not be unique. It is unique, if the p_i form a Gröbner basis.

In the implementation of `groebner::normalf`, reduction modulo some p_i of largest possible total degree is preferred, if reduction modulo several p_i is possible.

See Also

MuPAD Functions

`groebner::gbasis` | `poly`

groebner::spoly

The S-polynomial of two polynomials

Syntax

`groebner::spoly(`p_1`, `p_2`, <order>)`

Description

`groebner::spoly(p1, p2)` computes the S-polynomial of the polynomials `p1` and `p2`.

The rules laid down in the introduction to groebner concerning the polynomial types and the ordering apply.

The polynomials must be of the same type. In particular, do not mix polynomials created via `poly` and polynomial expressions!

Examples

Example 1

The polynomials

```
p1 := poly(x^2 - x + 2*y^2, [x, y]):
p2 := poly(x + 2*y - 1, [x, y]):
```

generate the following S-polynomial with respect to lexicographical ordering:

```
groebner::spoly(p1, p2, LexOrder)
```

$$\mathrm{poly}(-2\,x\,y+2\,y^2,\,[x,\,y])$$

```
delete p1, p2:
```

Parameters

p_1, p_2

A list or set of polynomials or polynomial expressions of the same type. The coefficients in these polynomials and polynomial expressions can be arbitrary arithmetical expressions.

`order`

One of the identifiers `DegInvLexOrder`, `DegreeOrder`, and `LexOrder`, or a user-defined term ordering of type `Dom::MonomOrdering`. The default ordering is `DegInvLexOrder`.

Return Values

Polynomial of the same type as the input polynomials. If polynomial expressions are used as input, then a polynomial expression is returned.

Algorithms

The S-polynomial of two polynomials p_1, p_2 is defined to be

$$\frac{\mathrm{lcm}(\mathrm{lterm}(p_1), \mathrm{lterm}(p_2))}{\mathrm{lmonomial}(p_1)}\, p_1 - \frac{\mathrm{lcm}(\mathrm{lterm}(p_1), \mathrm{lterm}(p_2))}{\mathrm{lmonomial}(p_2)}\, p_2,$$

where `lterm` and `lmonomial` are used in the same sense as the MuPAD functions of the same name. This formula is constructed such that the leading terms of the two summands cancel.

See Also

MuPAD Functions
`poly`

groebner::stronglyIndependentSets

Strongly independent set of variables

Syntax

```
groebner::stronglyIndependentSets(G)
```

Description

`groebner::stronglyIndependentSets(G)` computes a strongly independent set of variables modulo the ideal generated by `G`.

A set of variables S is strongly independent modulo an ideal I if no leading term of an element of the Gröbner basis of I consists entirely of elements of S. A set is maximally strongly independent if no proper superset of it is strongly independent. Two maximally strongly independent set may be of different size.

`groebner::stronglyIndependentSets` accepts Gröbner bases in the format returned by `groebner::gbasis`.

Examples

Example 1

The following example has been given by Moeller and Mora in 1983.

```
G:=map([X0^8*X2, X0*X3, X1^8*X3, X1^7*X3^2, X1^6*X3^3,
  X1^5*X3^4, X1^4*X3^5, X1^3*X3^6, X1^2*X3^7, X1*X3^8],
  poly, [X3, X2, X1, X0]):
groebner::stronglyIndependentSets(G)
```

```
[2, {X0, X1}, {{X2, X3}, {X0, X1}, {X1, X2}}]
```

```
delete G:
```

Parameters

G

The Gröbner basis of an ideal: a list.

Return Values

List of the form `[d, S, M]`, where `d` is an integer equal to the dimension of the ideal generated by `G`, `S` is the greatest strongly independent set of variables, and `M` is a set consisting of all maximal strongly independent sets of variables or a piecewise consisting of such lists.

References

[1] Kredel H. and V. Weispfenning, “Computing dimension and independent sets for polynomial ideals”, JSC volume 6 (1988), 231-247.

See Also

MuPAD Functions

`groebner::gbasis`

import – Import Data

import::csv
import::readbitmap
import::readdata
import::readlisp

import::csv

Read CSV data from an ASCII file

Syntax

```
import::csv(filename, <separator>, <NonNested>, <Trim>, <DecimalComma>)
import::csv(n, <separator>, <NonNested>, <Trim>, <DecimalComma>)
```

Description

`import::csv` is used to read CSV ("Comma Separated Values" or "Character Separated Values") data files produced by external programs, like Microsoft Excel®. CSV is an ASCII based tabular data file format, formally defined in RfC 4180, that has fields separated by the comma character.

Note: Some localized versions of Microsoft Excel use semicolons instead of commas! Set the parameter `separator` to change the default separator.

`import::csv(filename)` reads the data in the file `filename`. File data separated by a comma are regarded as different data elements. The result is a list of lists, each sublist representing one line of the file.

`import::csv(filename, separator)` reads the data in the file `filename`. File data separated by the character `separator` are regarded as different data elements. The result is a list of lists, each sublist representing one line of the file.

`import::csv(filename, separator, NonNested)` reads the data in the file `filename` as a single data record. File data separated by the character `separator` are regarded as different data elements. The result is a plain non-nested list containing the data of all lines of the file.

In contrast to `finput`, the data must not be ended by a colon or semicolon. Data separated by `separator` are interpreted as single data items. The default separator is a comma.

Empty lines are ignored.

All data elements in the file that cannot be converted to valid MuPAD numbers are imported as MuPAD strings.

`import::csv` tries to convert a number contained in the CSV file to a valid MuPAD number. For example: 1,234.56 or 1 234.56 are converted to the MuPAD number 1234.56. Many countries use a comma to separate the integral and fractional part instead of the dot used in England and the US. For example: 1234,56 or 1.234,56 are converted to 1234.56. `import::csv` expects this number format if the option `DecimalComma` is given.

Note: A comma as CSV separator doesn't make sense if the comma is used to separate thousands in a number or the decimal comma is used. In most cases, the CSV file uses a semicolon to separate data. So, a semicolon should be used as separator.

Note: All numbers contained in the CSV file must use the same radix separator, mixed formats cannot be converted.

With `NonNested`, the result will be a list containing all data. Otherwise, the result is a list of list, each "inner" list representing a line of the CSV file.

With `Trim`, leading and trailing blanks in strings are removed.

If the file is specified by a string, the corresponding file is opened and closed, automatically. If the user has opened a text file in `Read` mode and passes the file descriptor to import::readdata, the file remains open and needs to be closed by the user.

Files compressed with `gzip` or in a compatible format, whose names end in "`.gz`", are automatically decompressed while being read by `import::csv`.

`import::csv(filename)` searches for the file in various directories:

- First, the name is interpreted as a relative file name: `filename` is concatenated to each directory given by the environment variable `READPATH`.
- Then the file name is interpreted as an absolute path name.
- Then the file name is interpreted relative to the "working directory".
- Last, the file name is concatenated to the directory path.

If a file can be opened with one of this names, then the file is read.

Note that the meaning of "working directory" depends on the operating system. On Microsoft Windows systems and on Apple Mac OS X systems, the "working directory" is the folder where MuPAD is installed. On UNIX systems, it is the current working directory in which MuPAD was started; when started from a menu or desktop item, this is typically the user's home directory.

A path separator ("/") is inserted as necessary when concatenating a given path and `filename`.

If a file is specified by a file name, there is no need to open or close the file via `fopen` and `fclose`, respectively. This is done automatically by `import::readdata`.

Instead of a file name, also a file descriptor of a file opened via `fopen` can be used. Note that the file must have been opened in `Read` mode by `fopen`. If a file descriptor is used, the corresponding file is not closed automatically but must be closed by the user via `fclose`.

Examples

Example 1

We wish to read CSV data into a MuPAD session. Assume that the file "`datafile.csv`" contains the following two columns of ASCII data:

```
a ,12.5
  a-b   ,1234.56
```

`import::csv` returns the following list representing the data in the file:

```
data := import::csv("datafile.csv")
```

```
     [["a ", 12.5], [" a-b  ", 1234.56]]
```

```
data := import::csv("datafile.csv", Trim)
```

```
     [["a", 12.5], ["a-b", 1234.56]]
```

```
data := import::csv("datafile.csv", NonNested)
```

```
["a ", 12.5, " a-b  ", 1234.56]
```

Example 2

Let us assume that the file "`datafile.csv`" contains the following ASCII data:

```
a ;12.5
a-b;1,234.56
a b; -12345.6789E-02
```

We specify the data separator `";"` for reading the data:

```
import::csv("datafile.csv", ";")
```

```
[["a ", 12.5], ["a-b", 1234.56], ["a b", -123.456789]]
```

Example 3

Let us assume that the file "`datafile.csv`" contains the following ASCII data:

```
abc;12,5
a-b;1.234,56
a b; -12345.6789E-02
```

We specify the data separator `";"` and the option `DecimalComma` for reading the data:

```
import::csv("datafile.csv", ";", DecimalComma)
```

```
[["a ", 12.5], ["a-b", 1234.56], ["a b", -123.456789]]
```

Parameters

`filename`

The file name: a non-empty character string

n

A file descriptor provided by `fopen`: a positive integer

separator

The separator between data elements: a character string of length 1 (a single character). The default separator is a comma (the single character string `","`).

Options

NonNested

Return all file data as a single data record in a non-nested list. The data of all lines are ordered sequentially in this list.

Trim

Leading and trailing blanks in strings are removed.

DecimalComma

A decimal comma instead of a decimal point is used as the radix separator in the CSV file.

Return Values

Nested list of lists. The sublists contain the data of the individual lines. With the option `NonNested`, a plain list containing all data elements from every line in the file.

See Also

MuPAD Functions

`FILEPATH` | `fopen` | `import::readdata` | `readbytes` | `READPATH`

import::readbitmap

Read bitmap data

Syntax

```
import::readbitmap(filename, <ReturnType = DOM_HFARRAY | DOM_ARRAY | DOM_LIST>)
```

Description

`import::readbitmap` is used for reading ASCII or binary data files storing bitmap images of pictures. The following standard graphical formats can be read: BMP, DCX, DDS, WAD, GIF, ICO, JPG, LIF, MDL, PCD, PCX, PIC, PIX, PNG, PNM, PSD, PSP, PXR, RAW, SGI, TGA, TIF, WAL, XPM. The format of the pixel data is determined automatically from the contents of the file. The return value `[w, h, colordata]` provides the pixel height `h`, the pixel width `w`, and the color data of the bitmap image.

Either the complete return value or just the third element, `colordata`, can be passed to the function `plot::Raster` to generate a plot object that can be used in a MuPAD graphics. E.g., the command

```
plot(plot::Raster(import::readbitmap("mypicture.jpeg")))
```

creates a MuPAD graphics of the bitmap stored in the JPG file "`mypicture.jpeg`".

Note: Most of the standard graphical formats store the pixel data row by row in the usual reading order starting with the upper left corner of the image. The pixel data in the returned array `colordata` (if requesting `ReturnType = DOM_ARRAY`), however, are to be interpreted as follows:

`colordata[1, 1]` is the RGB color of the lower left corner.

`colordata[h, 1]` is the RGB color of the upper left corner.

`colordata[1, w]` is the RGB color of the lower right corner.

`colordata[h, w]` is the RGB color of the upper right corner.

The interpretation of the other return types is analogous, see below for details on the return types.

This is consistent with the interpretation of a color array by `plot::Raster`.

`import::readbitmap(filename)` searches for the file in various directories:

- First, the name is interpreted as a relative file name: `filename` is concatenated to each directory given by the environment variable `READPATH`.
- Then the file name is interpreted as an absolute path name.
- Then the file name is interpreted relative to the "working directory."
- Last, the file name is concatenated to the directory path.

If a file can be opened with one of this names, then the file is read.

Note that the meaning of "working directory" depends on the operating system. On Microsoft Windows systems and on Apple Mac OS X systems, the "working directory" is the folder where MuPAD is installed. On UNIX systems, it is the current working directory in which MuPAD was started; when started from a menu or desktop item, this is typically the user's home directory.

A path separator ("/") is inserted as necessary when concatenating a given path and `filename`.

`import::readbitmap` does not accept file handles returned by `fopen`. Nor can it handle files which have been compressed by `gzip`, but since most bitmap formats employ high quality compression in any case, there is little reason to try compressing them again in any case.

Examples

Example 1

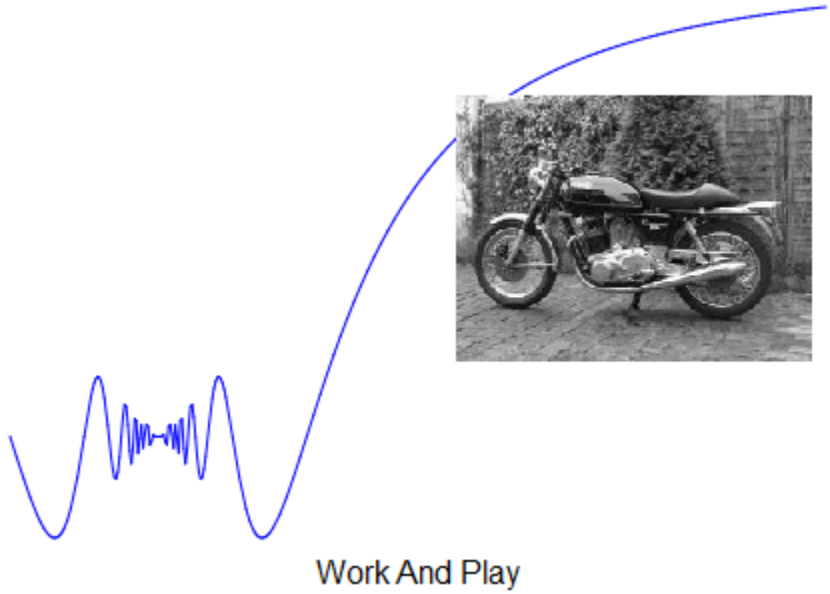
We import a PGM (portable graymap) picture:

```
[w, h, Norton] := import::readbitmap("Norton.pgm"):
```

The bitmap image is to be embedded in a MuPAD graphics. We use the width `w` and the height `h` to place the bitmap in a rectangle whose sides have the same ratio as the

original bitmap. With `Scaling = Constrained` we make sure that this aspect ratio is also used in the final graphics:

```
xmin := 2: xmax := xmin + w/100:
ymin := 0.5: ymax := ymin + h/100:
plot(plot::Function2d(x*sin(PI/x), x = -1..4.5, AdaptiveMesh = 2),
     plot::Raster(Norton, x = xmin ..xmax, y = ymin .. ymax),
     Scaling = Constrained, Footer = "Work And Play"):
```



Parameters

`filename`

The file name: a non-empty character string

Options

`ReturnType`

Option, specified as `ReturnType = DOM_HFARRAY | DOM_ARRAY | DOM_LIST`

Set the type of the actual color data returned as `colordata`.

If set to `DOM_LIST`, `colordata` is a nested list, the outermost list containing `h` lists, each of which represents one row of image data and contains `w` lists of three floating-point numbers, each of which represents an "RGB Colors" color.

If set to `DOM_ARRAY`, `colordata` is an array containing lists with color information, as in `array(2, 1..h, 1..w, [color`$_1$`, color`$_2$`, …] )`. The interpretation is analogous to the nested lists described above.

If set to `DOM_HFARRAY`, which is the default setting, `colordata` is a `DOM_HFARRAY` of dimensions `hfarray(3, 1..h, 1..w, 1..3, [actual data])`. The interpretation of these floating-point values is as described above for the `DOM_LIST` case.

Return Values

list`[w, h, colordata]`. The integer `w` is the pixel width of the bitmap. The integer `h` is the pixel height of the bitmap. `colordata` provides the `RGB` colors of the bitmap. Its type depends on the setting of the option `ReturnType`.

See Also

MuPAD Functions

`import::readdata` | `readbytes` | `READPATH`

MuPAD Graphical Primitives

`plot::Raster`

import::readdata

Read data from an ASCII file

Syntax

```
import::readdata(filename | n, <separator>, <NonNested>)
```

Description

`import::readdata(filename)` reads the data in the file `filename`. File data separated by whitespace are regarded as different data elements. The result is a list of lists, each sublist representing one line of the file.

`import::readdata(filename, separator)` reads the data in the file `filename`. File data separated by the character `separator` are regarded as different data elements. The result is a list of lists, each sublist representing one line of the file.

`import::readdata(filename, separator, NonNested)` reads the data in the file `filename` as a single data record. File data separated by the character `separator` are regarded as different data elements. The result is a plain non-nested list containing the data of all lines of the file.

`import::readdata(filename)` searches for the file in various directories:

- First, the name is interpreted as a relative file name: `filename` is concatenated to each directory given by the environment variable `READPATH`.
- Then the file name is interpreted as an absolute path name.
- Then the file name is interpreted relative to the “working directory.”
- Last, the file name is concatenated to the library directory.

If a file can be opened with one of this names, then the file is read.

Note that the meaning of “working directory” depends on the operating system. On Microsoft Windows systems and on Apple Mac OS X systems, the “working directory”

is the folder where MuPAD is installed. On UNIX systems, it is the current working directory in which MuPAD was started; when started from a menu or desktop item, this is typically the user's home directory.

A path separator ("/") is inserted as necessary when concatenating a given path and `filename`.

If a file is specified by a file name, there is no need to open or close the file via `fopen` and `fclose`, respectively. This is done automatically by `import::readdata`.

Instead of a file name, also a file descriptor of a file opened via `fopen` can be used. Note that the file must have been opened in `Read` mode by `fopen`. If a file descriptor is used, the corresponding file is not closed automatically but must be closed by the user via `fclose`.

Files compressed by `gzip` or a compatible program (having a name ending in "`.gz`") are decompressed automatically upon reading.

All data elements in the file are interpreted as MuPAD objects. If a data element cannot be interpreted as a MuPAD object, it is imported as a MuPAD string. Otherwise, the corresponding MuPAD object is inserted into the list returned by `import::readdata`.

Note: Note that the MuPAD objects corresponding to the data elements are evaluated after reading. E.g., the data element "sin(0)" in the file is evaluated and imported as the MuPAD integer 0. Beware: the characters "`;`" and "`:`" have a specific meaning if not specified as separators in `import::readdata`: they separate MuPAD commands. Hence, if a read data element contains one of this characters, MuPAD interprets this data element as a sequence of statements and, upon evaluation, returns the value of the *last* statement as the MuPAD object corresponding to the data element. Cf. "Example 3" on page 13-14.

In contrast to `finput`, the data elements in the file do not have to be ended by a colon or a semicolon.

Empty lines in the file are ignored.

Examples

Example 1

We wish to read statistical data into a MuPAD session to test the correlation of two data samples. Assume that the file "`datafile`" contains the following two columns of ASCII data (each column representing a data sample):

```
 0.12    0.2534
 2.324   5.72
 1.02    2.2232
 4.02    7.321
 7.4    14.9
-7.4   -15.1
```

`import::readdata` returns the following list representing the data in the file:

```
data := import::readdata("datafile")
```

$$[[0.12, 0.2534], [2.324, 5.72], [1.02, 2.2232], [4.02, 7.321], [7.4, 14.9], [-7.4, -15.1]]$$

The data structure `stats::sample` converts this nested list into two data columns:

```
s := stats::sample(data)
```

```
 0.12  0.2534
2.324    5.72
 1.02  2.2232
 4.02   7.321
  7.4    14.9
 -7.4   -15.1
```

The following computation shows that there is a very strong correlation between the data in the first column and the data in the second column:

```
stats::correlation(s, 1, 2)
```

$$0.9982703003$$

If the data in the file are supposed to represent a single sample (data record), we may ignore the fact that the numbers are arranged on several lines. With `NonNested`, the data are read as a single sample:

```
data := import::readdata("datafile", NonNested)
```

$$[0.12, 0.2534, 2.324, 5.72, 1.02, 2.2232, 4.02, 7.321, 7.4, 14.9, -7.4, -15.1]$$

Mean and standard deviation of the data are:

```
stats::mean(data), stats::stdev(data)
```

$$1.900133333, 7.568789783$$

```
delete data, s:
```

Example 2

Let us assume that the file "`datafile`" contains the following ASCII data:

```
1 | 2   | 3

4| 5 | 6.65786
7| 8 |9| 5 | "ahfjd" | ab100|-23
```

We specify the data separator `"|"` for reading the data:

```
import::readdata("datafile", "|")
```

$$[[1, 2, 3], [4, 5, 6.65786], [7, 8, 9, 5, "ahfjd", ab100, -23]]$$

Note that whitespace inside the data elements as well as the empty line in the file are ignored.

Example 3

We first create the ASCII data files that will be used in this example. We recall that x degrees Celsius are $\frac{9x}{5} + 32$ degrees Fahrenheit. First, two data files are created

containing the matching temperatures from - 5 degrees Celsius to 30 degrees Celsius in steps of 5 degrees Celsius:

```
n1 := fopen(Text, "data1", Write):
n2 := fopen(Text, "data2", Write):
for celsius from -5 to 20 step 5 do
  fahrenheit := 9/5*celsius + 32:
  fprint(Unquoted, n1, celsius, " ", fahrenheit):
  fprint(n2, celsius, fahrenheit):
end_for:
fclose(n1):
fclose(n2):
```

The file "`data1`" now contains the following data:

```
-5 23
0 32
5 41
10 50
15 59
20 68
```

The file "`data2`" contains the following data:

```
-5:23:
0:32:
5:41:
10:50:
15:59:
20:68:
```

Now, we import the data:

```
import::readdata("data1")
```

$$[[-5, 23], [0, 32], [5, 41], [10, 50], [15, 59], [20, 68]]$$

Reading data from the file "`data2`" yields an unexpected result:

```
import::readdata("data2")
```

$$[[23], [32], [41], [50], [59], [68]]$$

What went wrong? Remember that the default data separator is whitespace. Consequently, MuPAD reads the expression `-5:23:` as the only data element in the first line. When MuPAD evaluates this data element, it interprets it as a sequence of two MuPAD statements. The result of the statement sequence is the result of the last of the two statements, i.e., the number 23. This is the first datum in the resulting list. For getting the data as desired, an appropriate separator must be specified. The file "`data2`" should be read as follows:

```
import::readdata("data2", ":")
```

$$[[-5, 23], [0, 32], [5, 41], [10, 50], [15, 59], [20, 68]]$$

We use the option `NonNested` to get a plain list containing all data elements without putting each record (line) in a sublist of its own:

```
import::readdata("data2", ":", NonNested)
```

$$[-5, 23, 0, 32, 5, 41, 10, 50, 15, 59, 20, 68]$$

```
delete n1, n2:
```

Example 4

Here we can see that the data are evaluated after reading. First, we create the data file:

```
n1 := fopen(Text, "data3", Write) :
fprint(Unquoted, n1, a, " 12 ", b):
fclose(n1):
```

Now, the data are read:

```
import::readdata("data3")
```

$$[[a, 12, b]]$$

If `a` and `b` have values, we get:

```
a := 3:  b := 34:  import::readdata("data3")
```

$$[[3, 12, 34]]$$

```
delete n1, a, b:
```

Example 5

First, we create a data file with random floating-point data that a separated by blank characters:

```
n := fopen(Text, "data4", Write):
for i from 1 to 3 do
    fprint(Unquoted, n, (frandom(), " ") $ j = 1..4);
end_for;
fclose(n):
```

This file is reopened for reading with `fopen`:

```
n := fopen(Text, "data4", Read)
```

```
     64
```

The file descriptor `n` returned by `fopen` can be passed to `import::readdata`:

```
import::readdata(n)
```

```
[[0.2703567032, 0.8142678572, 0.1145977439, 0.247668289],
  [0.436855213, 0.7507294917, 0.5143284818, 0.47002619],
  [0.06956333824, 0.5063265159, 0.4145331467, 0.365909575]]
```

Note, however, that the file was opened explicitly by the user with `fopen` and is not closed automatically by `import::readdata`. Consequently, the user is supposed to close the file explicitly via `fclose`:

```
fclose(n):
delete i, n:
```

Parameters

`filename`

The file name: a non-empty character string

n

A file descriptor provided by `fopen`: a positive integer

separator

The separator between data elements: a character string of length 1 (a single character). The default separator is whitespace.

Options

NonNested

Return all file data as a single data record in a non-nested list. The data of all lines are ordered sequentially in this list.

Return Values

Nested list of lists. The sublists contain the data of the indiviual lines. With the option `NonNested`, a plain list containing all data elements from every line in the file.

See Also

MuPAD Functions

`finput` | `fopen` | `fread` | `ftextinput` | `import::csv` | `import::readbitmap` | `pathname` | `read`

import::readlisp

Parse Lisp-formatted string

Syntax

```
import::readlisp(s)
```

Description

`import::readlisp(s)` parses the Lisp-formatted string `s` and returns the corresponding MuPAD expression.

`import::readlisp` returns the constructed MuPAD expression as an unevaluated call. So the result of `import::readlisp` is in every case of type `DOM_EXPR`.

If the parsed string `s` contains only white spaces, then the unevaluated `null()` expression is returned.

Examples

Example 1

A first example:

```
import::readlisp("(INTEGRATE (EXPT X -1) X)")
```

$$\int \frac{1}{X}\,\mathrm{d}X$$

```
import::readlisp("(EXP 2.0)")
```

$$e^{2.0}$$

Example 2

In “Example 1” on page 13-19 above we can see that the corresponding MuPAD expression is not evaluated. Let us have a closer look at this behavior:

```
domtype(import::readlisp("(INTEGRATE (EXPT X -1) X)")),
eval(import::readlisp("(INTEGRATE (EXPT X -1) X)")),
domtype(import::readlisp("(EXP 2.0)")),
eval(import::readlisp("(EXP 2.0)"))
```

DOM_EXPR, $\ln(X)$, DOM_EXPR, 7.389056099

Example 3

Another example demonstrating that `import::readlisp` returns an unevaluated call:

```
x := 2:  import::readlisp("(* x (/ 2 y))")
```

$x\,\frac{2}{y}$

```
eval(import::readlisp("(* x (/ 2 y))"))
```

$\frac{4}{y}$

Example 4

An empty string is converted into an unevaluated call of `null()`:

```
type(import::readlisp(""))
```

"null"

We try to convert an illegal Lisp string:

```
import::readlisp("(* 2(EXP 3)")
```

```
Error: The closing parenthesis is missing. [import::parseLambda]
```

Parameters

s

A string

Return Values

MuPAD expression of type `DOM_EXPR`

intlib – Integration Utilities

intlib::byparts
intlib::changevar
intlib::intOverSet
intlib::printWarnings

intlib::byparts

Integration by parts

Syntax

```
intlib::byparts(integral, du)
```

Description

`intlib::byparts(integral, du)` performs on `integral` the integration by parts, where `du` is the part to be integrated and returns an expression containing the unevaluated partial integral.

Mathematically, the rule of integration by parts is formally defined for indefinite integrals as

$$\int u'(x)\, v(x)\, \mathrm{d}x = u(x)\, v(x) - \int u(x)\, v'(x)\, \mathrm{d}x$$

and for definite integrals as

$$\int_a^b u'(x)\, v(x)\, \mathrm{d}x = u(b)\, v(b) - u(a)\, v(a) - \int_a^b u(x)\, v'(x)\, \mathrm{d}x$$

.

`intlib::byparts` works for indefinite as well as for definite integrals.

If MuPAD cannot solve the integral for `du` in case of definite integration, the function call is returned unevaluated.

The first argument should contain a symbolic integral of type `"int"`. Such an expression can be obtained with `hold` or `freeze` (cf. “Example 1” on page 14-3).

The second argument `du` should typically be a partial expression of the integrand in `integral`.

Examples

Example 1

As a first example we apply the rule of integration by parts to the integral $\int_a^b x\,e^x\,dx$. By using the function `hold` we ensure that the first argument is of type `"int"`:

```
intlib::byparts(hold(int)(x*exp(x), x = a..b), exp(x))
```

$$b\,e^b - a\,e^a - \int_a^b e^x\,dx$$

In this case the ansatz is chosen as $u'(x) = e^x$ and thus $v(x) = x$.

Example 2

In the following we give a more advanced example using the method of integration by parts for solving the integral $\int e^{a\,x}\sin(b\,x)\,dx$. For this we have to prevent that the integrator already evaluates the integrals. Thus we first inactivate the requested integral with the function `freeze`

```
F := freeze(int)(exp(a*x)*sin(b*x), x)
```

$$\int e^{a\,x}\sin(b\,x)\,dx$$

and apply afterwards partial integration with $u'(x) = e^{a\,x}$:

```
F1 := intlib::byparts(F, exp(a*x))
```

$$\frac{e^{a\,x}\sin(b\,x)}{a} - \int \frac{b\,e^{a\,x}\cos(b\,x)}{a}\,dx$$

This result contains another symbolic integral, which MuPAD can solve directly:

```
eval(F1)
```

$$\frac{e^{a\,x}\sin(b\,x)}{a} - \frac{b\,e^{a\,x}\,(a\cos(b\,x)+b\sin(b\,x))}{a\,(a^2+b^2)}$$

Example 3

Here we demonstrate the difference between indefinite and definite integration by parts. If in the indefinite case the partial part cannot be solved, simply the unevaluated integral is plugged into the integration rule:

```
intlib::byparts(hold(int)(x*f(x), x),f(x))
```

$$x\int f(x)\,dx - \int\int f(x)\,dx\,dx$$

This is no longer true for the definite case:

```
intlib::printWarnings(TRUE):
intlib::byparts(hold(int)(x*f(x), x=a..b),f(x))
```

```
Warning: No closed form for 'int(f(x), x)' is found. [intlib::byparts]
```

$$\text{intlib::byparts}\left(\int_a^b x\,f(x)\,dx,\ f(x)\right)$$

Parameters

`integral`

Integral: an arithmetical expression containing a symbolic `"int"` call of the form `int(du*v, x)` or `int(du*v, x = a..b)`

`du`

The part to be integrated: an arithmetical expression

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`intlib::changevar` | `subs`

intlib::changevar

Change of variable

Syntax

```
intlib::changevar(integral, eq, <var>)
```

Description

`intlib::changevar(integral, eq)` performs a change of variable for indefinite and definite integrals.

Mathematically, the substitution rule is formally defined for indefinite integrals as

$$\int f(g(x))\, g'(x)\, \mathrm{d}x = \left(\left(\int f(t)\, \mathrm{d}t \right) \Big|_{t = g(x)} \right)$$

and for definite integrals as

$$\int_a^b f(g(x))\, g'(x)\, \mathrm{d}x = \int_{g(a)}^{g(b)} f(t)\, \mathrm{d}t$$

`intlib::changevar(integral, eq)` performs in `integral` the change of variable defined by `eq` and returns an unevaluated new integral. You can use the `eval` command to find the closed form of this new integral providing that the closed form exists.

`intlib::changevar` works for indefinite as well as for definite integrals.

The first argument should contain a symbolic integral of type `"int"`. Such an expression can be obtained with `hold` or `freeze`. See “Example 1” on page 14-7.

If more than two variables occur in `eq`, the new variable must be given as third argument.

If MuPAD cannot solve the given equation `eq` an error will occur.

Examples

Example 1

As a first example we perform a change of variable for the integral $\int_a^b f(x+c)\,\mathrm{d}x$. By using the `hold` function we ensure that the first argument is of type `"int"`:

```
intlib::changevar(hold(int)(f(x + c), x = a..b),
                  t = x + c, t)
```

$$\int_{a+c}^{b+c} f(t)\,\mathrm{d}t$$

Note that in this case the substitution equation has two further variables besides x. Thus it is necessary to specify the new integration variable as third argument.

Example 2

In the following example we use the change of variable method for solving the integral $\int\cos(\ln(x))\,\mathrm{d}x$. First we perform the transformation $t = ln(x)$:

```
f1 := intlib::changevar(hold(int)(cos(ln(x)), x),
                        t = ln(x), t)
```

$$\int e^t \cos(t)\,\mathrm{d}t$$

Now we can evaluate the integral with the MuPAD integrator:

```
f2:=eval(f1)
```

$$\frac{e^t\,(\cos(t)+\sin(t))}{2}$$

Finally we change the variable t back to x and get the result:

```
F := simplify(f2 | t = ln(x))
```

$$\frac{\sqrt{2}\, x \sin\left(\frac{\pi}{4}+\ln(x)\right)}{2}$$

We can also verify the solution of the integral:

```
simplify(diff(F,x) - cos(ln(x)))
```

$$0$$

Parameters

`integral`

The integral: an arithmetical expression containing a symbolic `"int"` call

`eq`

Equation defining the new integration variable in terms of the old one: an equation

`var`

The new integration variable: an identifier

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`intlib::byparts` | `subs`

intlib::intOverSet

Integration over a set

Syntax

```
intlib::intOverSet(f, x, S)
```

Description

`intlib::intOverSet(f, x, S)` computes the integral $\int_{-\infty}^{\infty} f(x)\, i_S(x)\, \mathrm{d}x$ where $i_S(x)$ is the indicator function of the set S.

If S is an interval (a, b) with $a \le b$, the call is equivalent to `int(f, x=a..b)`. However, by definition, interchanging the borders to `int(f, x=b..a)` just reverses the sign of the latter while (b, a) is empty and any integral over the empty set is zero.

The function may return unevaluated if the integral could not be computed.

Examples

Example 1

For intervals, calling `intlib::intOverSet` is just equivalent to calling definite integration:

```
int(1/x, x=1..2), intlib::intOverSet(1/x, x, Dom::Interval(1, 2))
```

$$\ln(2), \ln(2)$$

If the lower border is greater than the upper, this does not hold anymore:

```
int(1/x, x=2..1), intlib::intOverSet(1/x, x, Dom::Interval(2, 1))
```

$$-\ln(2), 0$$

Example 2

In more complex cases, the function returns unevaluated:

```
intlib::intOverSet(1/x^2, x, solve(t > sin(t), t))
```

$$\int_{x \in \mathrm{solve}(\sin(t) < t,\, t)} \frac{1}{x^2}\, dx$$

Parameters

f

Arithmetical expression

x

Identifier

S

Set-theoretic expression

Return Values

Arithmetical expression.

See Also

MuPAD Functions
`int`

intlib::printWarnings

Enable or disable warnings

Syntax

```
intlib::printWarnings(TRUE)
```

```
intlib::printWarnings(FALSE)
```

```
intlib::printWarnings()
```

Description

`intlib::printWarnings` lets you enable or disable warnings.

By default, MuPAD does not display warnings during integration. To enable warnings, use the `intlib::printWarnings(TRUE)` function call. If later you want to disable warnings, use the `intlib::printWarnings(FALSE)` function call. See “Example 1” on page 14-11.

The `intlib::printWarnings()` function call shows whether warnings are enabled or disabled. See “Example 1” on page 14-11.

The output of `intlib::printWarnings` displays the previous setting. You can save this previous setting and switch to a new setting in a single function call. See “Example 2” on page 14-12.

Examples

Example 1

Enable the warnings by setting the value of `intlib::printWarnings` to `TRUE`:

```
intlib::printWarnings(TRUE):
```

Compute the integral of $|x|$ under the assumption that `x` is an integer. MuPAD cannot integrate the expression over a discrete subset of the real numbers. The system issues a warning and integrates over the set $\mathbb{R}$ of real numbers:

```
int(abs(x), x) assuming x in Z_
```

```
Warning: Cannot integrate when 'x' has property 'Z_'. The assumption that 'x' has the p
```

$$\frac{x^2\,\mathrm{sign}(x)}{2}$$

If you evaluate the same integral again, MuPAD does not recalculate the integral. The system remembers the previous result and returns it, skipping the warning:

```
int(abs(x), x) assuming x in Z_
```

$$\frac{x^2\,\mathrm{sign}(x)}{2}$$

To check whether the warnings are enabled or disabled, use the `intlib::printWarnings()` function call:

```
intlib::printWarnings()
```

TRUE

Disable the warnings for further computations:

```
intlib::printWarnings(FALSE):
```

Example 2

Enable the warnings and save the previous setting in a single function call:

```
old := intlib::printWarnings(TRUE):
```

Assume that `x` is positive. Then, integrate `x` over the interval [- 2, 1]. In this case, the system issues a warning, temporarily disregards the assumption `x > 0`, and integrates over the interval [- 2, 1]:

```
assume(x > 0):
int(x, x = -2..1)
```

```
Warning: The assumption that 'x' has property 'Dom::Interval([-2], [1])' instead of giv
```

$$-\frac{3}{2}$$

Restore the setting of `intlib::printWarnings`:

```
intlib::printWarnings(old):
```

The warnings are disabled now:

```
intlib::printWarnings()
```

FALSE

For further computations, clear the assumption on the variable `x`:

```
unassume(x):
```

Return Values

Previously set value TRUE or FALSE

See Also

MuPAD Functions

`int`

More About

- “Integration”

linalg – Linear Algebra

linalg::addCol
linalg::addRow
linalg::adjoint
linalg::angle
linalg::basis
linalg::charmat
linalg::charpoly
linalg::col
linalg::companion
linalg::concatMatrix
linalg::cond
linalg::crossProduct
linalg::delCol
linalg::delRow
linalg::eigenvalues
linalg::eigenvectors
linalg::expr2Matrix
linalg::factorCholesky
linalg::factorLU
linalg::factorQR
linalg::frobeniusForm
linalg::gaussElim
linalg::gaussJordan
linalg::hermiteForm
linalg::hessenberg
linalg::hilbert
linalg::htranspose
linalg::intBasis
linalg::inverseLU
linalg::invhilbert
linalg::invpascal
linalg::isHermitian

linalg::isPosDef
linalg::isUnitary
linalg::jordanForm
linalg::kroneckerProduct
linalg::matdim
linalg::matlinsolve
linalg::matlinsolveLU
linalg::minpoly
linalg::multCol
linalg::multRow
linalg::ncols
linalg::nonZeros
linalg::normalize
linalg::nrows
linalg::nullspace
linalg::ogCoordTab
linalg::orthog
linalg::pascal
linalg::permanent
linalg::pseudoInverse
linalg::randomMatrix
linalg::rank
linalg::row
linalg::scalarProduct
linalg::setCol
linalg::setRow
linalg::smithForm
linalg::sqrtMatrix
linalg::stackMatrix
linalg::submatrix
linalg::substitute
linalg::sumBasis
linalg::swapCol
linalg::swapRow
linalg::sylvester
linalg::tr
linalg::toeplitz
linalg::toeplitzSolve
linalg::transpose
linalg::vandermonde

linalg::invvandermonde
linalg::vandermondeSolve
linalg::vecdim
linalg::vectorOf
linalg::wiedemann

linalg::addCol

Linear combination of matrix columns

Syntax

`linalg::addCol(A, c1, c2, s1)`

`linalg::addCol(A, c1, c2, s1, s2)`

Description

`linalg::addCol(A, c1, c2, s1)` adds s_1 times column c_1 to column c_2, in the matrix A.

`linalg::addCol(A, c1, c2, s)` returns a copy of the matrix A in which column c_2 of A is replaced by $s\,col(A, c_1) + col(A, c_2)$.

`linalg::addCol(A, c1, c2, s1, s2)` returns a copy of the matrix A in which column c_2 of A is replaced by $s_1\,col(A, c_1) + s_2\,col(A, c_2)$.

Examples

Example 1

The following defines a 3×3 matrix over the integers:

```
A := Dom::Matrix(Dom::Integer)(
  [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
)
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

We replace the 2nd column by - *col*(*A*, 1) + *col*(*A*, 2), i.e., we subtract the first column from the second:

```
linalg::addCol(A, 1, 2, -1)
```

$$\begin{pmatrix} 1 & 1 & 3 \\ 4 & 1 & 6 \\ 7 & 1 & 9 \end{pmatrix}$$

Example 2

The following defines a 2×3 matrix over the reals:

```
B := Dom::Matrix(Dom::Real)(
  [[sin(2), 0, 1], [1, PI, 0]]
)
```

$$\begin{pmatrix} \sin(2) & 0 & 1 \\ 1 & \pi & 0 \end{pmatrix}$$

If `s` is an expression that does not represent a real number then an error message is reported. The following tries to replace the 1st column by *x* *col*(*B*, 3) + *col*(*B*, 1), where *x* is an identifier which cannot be converted to the component ring `Dom::Real` of *B*:

```
delete x: linalg::addCol(B, 3, 1, x)
```

```
Error: Cannot convert 'x'. [linalg::addCol]
```

Example 3

If symbolic expressions are involved, then one may define matrices over a component ring created by `Dom::ExpressionField`. The following example defines a matrix over this default component ring:

```
delete a11, a12, a21, a22, x:
C := matrix([[a11, a12], [a21, a22]])
```

$$\begin{pmatrix} a11 & a12 \\ a21 & a22 \end{pmatrix}$$

We retry the input from the previous example:

```
linalg::addCol(C, 2, 1, x)
```

$$\begin{pmatrix} a11 + a12\,x & a12 \\ a21 + a22\,x & a22 \end{pmatrix}$$

Parameters

A

An $m \times n$ matrix of a domain of category `Cat::Matrix`

c_1, c_2

The column indices: positive integers less or equal to n

s_1, s_2

Expressions that can be converted to the component ring of A

Return Values

Matrix of the same domain type as A.

See Also

MuPAD Domains
`Dom::Matrix`

MuPAD Functions
`linalg::addRow` | `linalg::col` | `linalg::multCol` | `linalg::multRow`

linalg::addRow

Linear combination of matrix rows

Syntax

```
linalg::addRow(A, r₁, r₂, s)
```

```
linalg::addRow(A, r₁, r₂, s₁, s₂)
```

Description

`linalg::addRow(A, r₁, r₂, s₁)` adds s_1 times row r_1 to row r_2, in the matrix `A`.

`linalg::addRow(A, r₁, r₂, s)` returns a copy of the matrix A in which row r_2 of A is replaced by $s\,row(A, r_1) + row(A, r_2)$.

`linalg::addRow(A, r₁, r₂, s₁, s₂)` returns a copy of the matrix A in which row r_2 of A is replaced by $s_1\,row(A, r_1) + s_2\,row(A, r_2)$.

Examples

Example 1

The following defines a 3×3 matrix over the integers:

```
A := Dom::Matrix(Dom::Integer)(
  [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
)
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

We replace the 2nd row by - $row(A, 1) + row(A, 2)$, i.e., we subtract the first row from the second:

```
linalg::addRow(A, 1, 2, -1)
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 3 & 3 \\ 7 & 8 & 9 \end{pmatrix}$$

Example 2

The following defines a 2×3 matrix over the reals:

```
B := Dom::Matrix(Dom::Real)(
  [[sin(2), 0, 1], [1, PI, 0]]
)
```

$$\begin{pmatrix} \sin(2) & 0 & 1 \\ 1 & \pi & 0 \end{pmatrix}$$

If `s` is an expression that does not represent a real number then an error message is reported. The following tries to replace the 1st row by $x\,row(B, 2) + row(B, 1)$, where x is an identifier which cannot be converted to the component ring `Dom::Real` of B:

```
delete x: linalg::addRow(B, 2, 1, x)
```

```
Error: Cannot convert 'x'. [linalg::addRow]
```

Example 3

If symbolic expressions are involved, then one may define matrices over the component ring created by `Dom::ExpressionField`. The following example defines a matrix over this default component ring:

```
delete a11, a12, a21, a22, x:
C := matrix([[a11, a12], [a21, a22]])
```

$$\begin{pmatrix} a11 & a12 \\ a21 & a22 \end{pmatrix}$$

We retry the input from the previous example:

```
linalg::addRow(C, 2, 1, x)
```

$$\begin{pmatrix} a11 + a21\,x & a12 + a22\,x \\ a21 & a22 \end{pmatrix}$$

Parameters

A

An $m \times n$ matrix of a domain of category `Cat::Matrix`

r_1, r_2

The row indices: positive integers less or equal to m

s, s_1, s_2

Expressions that can be converted to the component ring of A

Return Values

Matrix of the same domain type as A.

See Also

MuPAD Functions

`linalg::addCol` | `linalg::multCol` | `linalg::multRow` | `linalg::row`

linalg::adjoint

Adjoint of a matrix

Syntax

```
linalg::adjoint(A)
```

Description

`linalg::adjoint(A)` computes the adjoint $Adj(A)$ of the $n \times n$ matrix A. The adjoint matrix satisfies the equation $A\ \mathrm{Adj}(A) = \det A\ I_n$, where I_n is the $n \times n$ identity matrix.

The component ring of `A` must be of category `Cat::CommutativeRing`.

Examples

Example 1

We define a matrix over the rationals:

```
MatQ := Dom::Matrix(Dom::Rational):
A := MatQ([[0, 2, 1], [2, 1, 0], [1, 0, 2]])
```

$$\begin{pmatrix} 0 & 2 & 1 \\ 2 & 1 & 0 \\ 1 & 0 & 2 \end{pmatrix}$$

Then the adjoint matrix of `A` is given by:

```
Ad := linalg::adjoint(A)
```

$$\begin{pmatrix} 2 & -4 & -1 \\ -4 & -1 & 2 \\ -1 & 2 & -4 \end{pmatrix}$$

We check the property of the adjoint matrix Ad mentioned above:

```
A * Ad = det(A)*MatQ::identity(3)
```

$$\begin{pmatrix} -9 & 0 & 0 \\ 0 & -9 & 0 \\ 0 & 0 & -9 \end{pmatrix} = \begin{pmatrix} -9 & 0 & 0 \\ 0 & -9 & 0 \\ 0 & 0 & -9 \end{pmatrix}$$

Parameters

A

A square matrix of a domain of category `Cat::Matrix`

Return Values

Matrix of the same domain type as A.

Algorithms

The adjoint of a square matrix A is the matrix whose (i, j)-th entry is the (j, i)-th cofactor of A.

The (j, i)-th *cofactor* of A is defined by $\alpha'_{ji} = (-1)^{i+j} \det\left(A_{ij}\right)$, where A_{ij} is the submatrix of A obtained from A by deleting the i-th row and j-th column.

See Also

MuPAD Functions

`det`

linalg::angle

Angle between two vectors

Syntax

```
linalg::angle(u, v)
```

Description

`linalg::angle(u,v)` computes the angle φ between the two vectors `u` and `v`, defined by

$$\varphi = \mathrm{acos}\left(\frac{\langle \vec{u}, \vec{v}\rangle}{\|\vec{u}\|_2 \, \|\vec{v}\|_2}\right),$$

where $\langle \vec{u}, \vec{v}\rangle$ denotes the scalar product of two vectors given by `linalg::scalarProduct`, and $\|\cdot\|_2$ the 2-norm of a vector, i.e., $\|\vec{u}\|_2 = \sqrt{\langle \vec{u}, \vec{u}\rangle}$.

`linalg::angle` does not check if the computation is defined in the corresponding component ring. This can lead to an error message, as shown in “Example 2” on page 15-13.

The following relationship between the angle between $\vec{u}$ and $\vec{v}$ and the angle between $\vec{u}$ and $-\vec{v}$ holds: $\varphi(\vec{u}, \vec{v}) = \pi - \varphi(\vec{u}, \vec{v})$.

An error message is returned if the vectors are not defined over the same component ring.

Examples

Example 1

We compute the angle between the two vectors $\begin{pmatrix} 2 \\ 5 \end{pmatrix}$ and $\begin{pmatrix} -3 \\ 3 \end{pmatrix}$:

```
phi := linalg::angle(
  matrix([2, 5]), matrix([-3, 3])
)
```

$$\arccos\left(\frac{\sqrt{18}\,\sqrt{29}}{58}\right)$$

We use the function `float` to get a floating-point approximation of this number:

```
float(phi)
```

$$1.165904541$$

We give two further examples:

```
linalg::angle(
  matrix([1, -1]), matrix([1, 1])
)
```

$$\frac{\pi}{2}$$

```
linalg::angle(
  matrix([1, 1]), matrix([-1, -1])
)
```

$$\pi$$

Example 2

`linalg::angle` does not check whether the term $\frac{\langle \vec{u}, \vec{v} \rangle}{\|\vec{u}\|_2 \, \|\vec{v}\|_2}$ is defined in the corresponding component ring.

As an example, we try to compute the angle between two vectors with components in $\mathbb{Z}_7$:

```
MatZ7 := Dom::Matrix(Dom::IntegerMod(7))
```

$$\text{Dom::Matrix}(\text{Dom::IntegerMod}(7))$$

The following call leads to an error because the 2-norm cannot be computed:

```
linalg::angle(MatZ7([1, 1]), MatZ7([-1, -1]))
```

```
Error: An integer exponent is expected. [(Dom::IntegerMod(7))::_power]
```

Note that the domain `Dom::IntegerMod(7)` does not implement the square root of an element, therefore in MuPAD you cannot compute the angle of any two vectors over $\mathbb{Z}_7$.

Parameters

u, v

Vectors of the same dimension; a vector is a $n \times 1$ or $1 \times n$ matrix of a domain of category `Cat::Matrix`

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`arccos` | `linalg::scalarProduct` | `linalg::vecdim`

linalg::basis

Basis for a vector space

Syntax

```
linalg::basis(S)
```

Description

`linalg::basis(S)` returns a basis for the vector space spanned by the vectors in the set or list *S*.

`linalg::basis(S)` removes those vectors in `S` that are linearly dependent on other vectors in `S`. The result is a basis for the vector space spanned by the vectors in `S`.

For an ordered basis of vectors, `S` should be a list of vectors.

The vectors in `S` must be defined over the same component ring.

The component ring of the vectors in `S` must be a field, i.e., it must be of category `Cat::Field`.

Examples

Example 1

We define the domain of matrices over $\mathbb{Q}$:

```
MatQ := Dom::Matrix(Dom::Rational):
```

and compute a basis for the vector space spanned by the vectors $\begin{pmatrix} 3 \\ -2 \end{pmatrix}$, $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\begin{pmatrix} 5 \\ -3 \end{pmatrix}$:

```
v1 := MatQ([3, -2]):
v2 := MatQ([1, 0]):
v3 := MatQ([5, -3]):
linalg::basis([v1, v2, v3])
```

$$\left[\begin{pmatrix} 3 \\ -2 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}\right]$$

If not a list but a set of vectors is given, then the basis returned may not consist of the same vectors as above. The order of the vectors in the set depends on the internal order (see `sysorder` and `DOM_SET`), i.e., the order of the vectors appears to be random:

```
b := linalg::basis({v1, v2, v3}):
op(b, 1)
```

$$\begin{pmatrix} 3 \\ -2 \end{pmatrix}$$

Parameters

S

A set or list of n-dimensional vectors; a vector is a $n \times 1$ or $1 \times n$ matrix of a domain of category `Cat::Matrix`

Return Values

Set or a list of vectors, respectively.

See Also

MuPAD Functions

`linalg::intBasis` | `linalg::sumBasis` | `lllint`

linalg::charmat

Characteristic matrix

Syntax

```
linalg::charmat(A, x)
```

Description

`linalg::charmat(A, x)` returns the characteristic matrix $x\,I_n$ - A of the $n \times n$ matrix A, where I_n denotes the $n \times n$ identity matrix.

The component ring of `A` must be a commutative ring, i.e., a domain of category `Cat::CommutativeRing`.

The characteristic matrix $M = x\,I_n - A$ of A can be evaluated at a point $x = u$ via `evalp(M, x = u)`. See "Example 2" on page 15-18.

Examples

Example 1

We define a matrix over the rational numbers:

```
A := Dom::Matrix(Dom::Rational)([[1, 2], [3, 4]])
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

and compute the characteristic matrix of A in the variable x:

```
MA := linalg::charmat(A, x)
```

$$\begin{pmatrix} x-1 & -2 \\ -3 & x-4 \end{pmatrix}$$

The determinant of the matrix MA is a polynomial in x, the characteristic polynomial of the matrix A:

```
pA := det(MA)
```

$$x^2 - 5\,x - 2$$

```
domtype(pA)
```

Dom::DistributedPolynomial([x], Dom::Rational, LexOrder)

Of course, we can compute the characteristic polynomial of A directly via `linalg::charpoly`:

```
linalg::charpoly(A, x)
```

$$x^2 - 5\,x - 2$$

The result is of the same domain type as the polynomial pA.

Example 2

We define a matrix over the complex numbers:

```
B := Dom::Matrix(Dom::Complex)([[1 + I, 1], [1, 1 - I]])
```

$$\begin{pmatrix} 1+i & 1 \\ 1 & 1-i \end{pmatrix}$$

The characteristic matrix of B in the variable z is:

```
MB := linalg::charmat(B, z)
```

$$\begin{pmatrix} z-1-i & -1 \\ -1 & z-1+i \end{pmatrix}$$

We evaluate MB at $z = i$ and get the matrix:

```
evalp(MB, z = I)
```

$$\begin{pmatrix} -1 & -1 \\ -1 & -1+2\,i \end{pmatrix}$$

Note that this is a matrix of the domain type `Dom::Matrix(Dom::Complex)`:

```
domtype(%)
```

Dom::Matrix(Dom::Complex)

Parameters

A

A square matrix of a domain of category `Cat::Matrix`

x

An identifier

Return Values

Matrix of the domain `Dom::Matrix(Dom::DistributedPolynomial([x], R))` or of `Dom::DenseMatrix(Dom::DistributedPolynomial([x], R))`, where `R` is the component ring of `A`.

See Also

MuPAD Functions

`linalg::charpoly`

linalg::charpoly

Characteristic polynomial of a matrix

Syntax

```
linalg::charpoly(A, x)
```

Description

`linalg::charpoly(A, x)` computes the characteristic polynomial of the matrix A. The characteristic polynomial of a $n \times n$ matrix is defined by $p_A(x) := \det\left(x\, I_n - A\right)$, where I_n denotes the $n \times n$ identity matrix.

The component ring of `A` must be a commutative ring, i.e., a domain of category `Cat::CommutativeRing`.

Examples

Example 1

We define a matrix over the rational numbers:

```
A := Dom::Matrix(Dom::Rational)([[1, 2], [3, 4]])
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Then the characteristic polynomial $p_A(x)$ is given by:

```
linalg::charpoly(A, x)
```

$$x^2 - 5\,x - 2$$

It is of the domain type:

```
domtype(%)
```

$$\text{Dom::DistributedPolynomial}([x], \text{Dom::Rational}, \text{LexOrder})$$

Example 2

We define a matrix over $\mathbb{Z}_7$:

```
B := Dom::Matrix(Dom::IntegerMod(7))([[1, 2], [3, 0]])
```

$$\begin{pmatrix} 1 \bmod 7 & 2 \bmod 7 \\ 3 \bmod 7 & 0 \bmod 7 \end{pmatrix}$$

The characteristic polynomial $p_B(x)$ of `B` is given by:

```
p := linalg::charpoly(B, x)
```

$$(1 \bmod 7)\, x^2 + (6 \bmod 7)\, x + (1 \bmod 7)$$

We compute the zeros of $p_B(x)$, i.e., the eigenvalues of the matrix `B`:

```
solve(p)
```

$$\{[x = 3 \bmod 7],\ [x = 5 \bmod 7]\}$$

Parameters

A

A square matrix of a domain of category `Cat::Matrix`

x

An identifier

Return Values

Polynomial of the domain `Dom::DistributedPolynomial([x],R)`, where `R` is the component ring of `A`.

Algorithms

`linalg::charpoly` implements Hessenberg's algorithm to compute the characteristic polynomial of a square matrix A. See: Henri Cohen: *A Course in Computational Algebraic Number Theory*, GTM 138, Springer Verlag.

This algorithm works for any field and requires only $O(n^3)$ field operations, in contrast to $O(n^4)$ when computing the determinant of the characteristic matrix of A.

Since the size of the components of A in intermediate computations of Hessenberg's algorithm can swell extremely, it is only applied for matrices over `Dom::Float` and `Dom::IntegerMod`.

For any other component ring, the characteristic polynomial is computed using the Berkowitz algorithm.

References

Reference: Jounaidi Abdeljaoued, *The Berkowitz Algorithm, Maple and Computing the Characteristic Polynomial in an Arbitrary Commutative Ring*, MapleTech Vol 4 No 3, pp 21-32, Birkhäuser, 1997.

See Also

MuPAD Functions

`det` | `linalg::charmat` | `linalg::hessenberg` | `linalg::minpoly`

linalg::col

Extract columns of a matrix

Syntax

```
linalg::col(A, c)
```

```
linalg::col(A, c1 .. c2)
```

```
linalg::col(A, list)
```

Description

`linalg::col(A, c)` extracts the *c*-th column vector of the matrix *A*.

`linalg::col(A, c1.. c2)` returns a list of column vectors whose indices are in the range `c1.. c2`. If `c2< c1` then the empty list `[]` is returned.

`linalg::col(A, list)` returns a list of column vectors whose indices are contained in `list` (in the same order).

Examples

Example 1

We define a matrix over $\mathbb{Q}$:

```
A := Dom::Matrix(Dom::Rational)(
  [[1, 1/5, 2], [-3/2, 0, 5]]
)
```

$$\begin{pmatrix} 1 & \frac{1}{5} & 2 \\ -\frac{3}{2} & 0 & 5 \end{pmatrix}$$

and illustrate the three different input formats for `linalg::col`:

```
linalg::col(A, 2)
```

$$\begin{pmatrix} \frac{1}{5} \\ 0 \end{pmatrix}$$

```
linalg::col(A, [2, 1, 3])
```

$$\left[\begin{pmatrix} \frac{1}{5} \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ -\frac{3}{2} \end{pmatrix}, \begin{pmatrix} 2 \\ 5 \end{pmatrix}\right]$$

```
linalg::col(A, 2..3)
```

$$\left[\begin{pmatrix} \frac{1}{5} \\ 0 \end{pmatrix}, \begin{pmatrix} 2 \\ 5 \end{pmatrix}\right]$$

Parameters

`A`

An $m \times n$ matrix of a domain of category `Cat::Matrix`

`c`

The column index: a positive integer less or equal to n

c_1 `..` c_2

A range of column indices (positive integers less or equal to n)

`list`

A list of column indices (positive integers less or equal to n)

Return Values

Single column vector or a list of column vectors; a column vector is an $m \times 1$ matrix of category `Cat::Matrix(R)`, where R is the component ring of `A`.

See Also

MuPAD Functions

`linalg::delCol` | `linalg::delRow` | `linalg::row` | `linalg::setCol` | `linalg::setRow`

linalg::companion

Companion matrix of a univariate polynomial

Syntax

```
linalg::companion(p, <x>)
```

Description

`linalg::companion(p)` returns the companion matrix associated with the polynomial *p*.

`p` must be monic and of degree one at least.

If `p` is a polynomial, i.e., an object of type `DOM_POLY`, then specifying `x` has no effect.

If `p` is a polynomial, then the component ring of the returned matrix is the coefficient ring of `p`, except in two cases for built-in coefficient rings: if the coefficient ring of `p` is `Expr` then the domain `Dom::ExpressionField()` is the component ring of the companion matrix. If it is `IntMod(m)` then the companion matrix is defined over the ring `Dom::IntegerMod(m)` (see “Example 2” on page 15-27).

If `p` is a polynomial expression, then the companion matrix is defined over `Dom::ExpressionField()`.

If `p` is a polynomial expression containing several symbolic indeterminates then `x` must be specified and distinguishes the indeterminate `x` from the other symbolic parameters.

Examples

Example 1

We start with the following polynomial expression:

```
delete a_0, a_1, a_2, a_3:
p := x^4 + a_3*x^3 + a_2*x^2 + a_1*x + a_0
```

$$x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

To compute the companion matrix of p with respect to x we must specify the second parameter x, because the expression `p` contains the indeterminates a_0, a_1, a_2, a_3 and x:

```
linalg::companion(p)
```

```
Error: The polynomial expression is multivariate. Specify the indeterminate as second a
```

```
linalg::companion(p, x)
```

$$\begin{pmatrix} 0 & 0 & 0 & -a_0 \\ 1 & 0 & 0 & -a_1 \\ 0 & 1 & 0 & -a_2 \\ 0 & 0 & 1 & -a_3 \end{pmatrix}$$

Of course, we can compute the companion matrix of p with respect to a_0 as well:

```
linalg::companion(p, a_0)
```

$$\left(-x^4 - a_3 x^3 - a_2 x^2 - a_1 x \right)$$

The following fails with an error message, because the polynomial p is not monic with respect to a_1:

```
linalg::companion(p, a_1)
```

```
Error: Polynomial is not monic. [linalg::companion]
```

Example 2

If we enter a polynomial over the built-in coefficient domain `Expr`, then the companion matrix is defined over the standard component ring for matrices (the domain `Dom::ExpressionField()`):

```
C := linalg::companion(poly(x^2 + 10*x + PI, [x]))
```

$$\begin{pmatrix} 0 & -\pi \\ 1 & -10 \end{pmatrix}$$

```
domtype(C)
```

Dom::Matrix()

If we define a polynomial over the build-in coefficient domain `IntMod(m)`, then the companion matrix is defined over the corresponding component ring `Dom::IntegerMod(m)`, as shown in the next example:

```
p := poly(x^2 + 10*x + 7, [x], IntMod(3))
```

$\mathrm{poly}(x^2 + x + 1, [x], \mathrm{IntMod}(3))$

```
C := linalg::companion(p)
```

$$\begin{pmatrix} 0 \bmod 3 & 2 \bmod 3 \\ 1 \bmod 3 & 2 \bmod 3 \end{pmatrix}$$

```
domtype(C)
```

Dom::Matrix(Dom::IntegerMod(3))

Parameters

p

An univariate polynomial, or a polynomial expression

x

An identifier

Return Values

Matrix of the domain `Dom::Matrix(R)`.

Algorithms

The companion matrix of the polynomial $x^n + a_{n_1} x^{n-1} + \ldots + a_1 x + a_0$ is the matrix:

$$C = \begin{pmatrix} 0 & \cdots & 0 & -a_0 \\ 1 & \ddots & & -a_1 \\ & \ddots & & \vdots \\ & & 0 & -a_{n-1} \\ & & 1 & -a_n \end{pmatrix}.$$

The companion matrix of a univariate polynomial p of degree n is an $n \times n$ matrix C with $p_C = p$, where p_C is the characteristic polynomial of C.

linalg::concatMatrix

Join matrices horizontally

Syntax

`linalg::concatMatrix(A, B1, <B2, …>)`

Description

`linalg::concatMatrix(A, B1, B2, dots )` returns the matrix formed by joining the matrices A, B_1, B_2, … horizontally.

The matrices `B1, B2, dots` are converted into the matrix domain `Dom::Matrix(R)`, where `R` is the component ring of `A`.

An error message is raised if one of these conversions fails, or if the matrices do not have the same number of rows as the matrix `A`.

A short form of `linalg::concatMatrix` is available through the dot operator `.`, i.e., instead of `linalg::concatMatrix(A, B)` one may use the short form `A . B`.

Examples

Example 1

We define the matrix:

```
A := matrix([[sin(x), x], [-x, cos(x)]])
```

$$\begin{pmatrix} \sin(x) & x \\ -x & \cos(x) \end{pmatrix}$$

and append the 2×2 identity matrix to the right of `A`:

```
I2 := matrix::identity(2):
linalg::concatMatrix(A, I2)
```

$$\begin{pmatrix} \sin(x) & x & 1 & 0 \\ -x & \cos(x) & 0 & 1 \end{pmatrix}$$

The short form for this operation is:

```
A . I2
```

$$\begin{pmatrix} \sin(x) & x & 1 & 0 \\ -x & \cos(x) & 0 & 1 \end{pmatrix}$$

Example 2

We define a matrix from the ring of 2×2 square matrices:

```
SqMatQ := Dom::SquareMatrix(2, Dom::Rational):
A := SqMatQ([[1, 2], [3, 4]])
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Note the following operation:

```
AA := A . A
```

$$\begin{pmatrix} 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 \end{pmatrix}$$

returns a matrix of a different domain type as the input matrix:

```
domtype(AA)
```

$$\text{Dom::Matrix(Dom::Rational)}$$

Parameters

A, B_1, B_2, ...

Matrices of a domain of category `Cat::Matrix`

Return Values

Matrix of the domain type `Dom::Matrix(R)`, where `R` is the component ring of `A`.

See Also

MuPAD Functions
`linalg::stackMatrix`

linalg::cond

Condition number of a matrix

Syntax

```
linalg::cond(M, <1 | 2 | Spectral | Infinity | Frobenius>)
```

Description

`linalg::cond(M)` computes the condition number of a matrix, defined by $\|M\|_\infty \left\|\frac{1}{M}\right\|_\infty$. By default the matrix norm Infinity is used by `linalg::cond`.

`linalg::cond(M)` is the short form of `linalg::cond(M, Infinity)`.

`linalg::cond(M, k)` computes the condition number of the matrix M, defined by $\|M\|_k \left\|\frac{1}{M}\right\|_k$.

For further details see the help page of `norm`.

Examples

Example 1

We define the 3×3 matrix A.

```
A := matrix(3,3, [[1,0,3],[-4,2,0],[0,3,-2]])
```

$$\begin{pmatrix} 1 & 0 & 3 \\ -4 & 2 & 0 \\ 0 & 3 & -2 \end{pmatrix}$$

Now we calculate the condition number of A for some matrix norms.

```
linalg::cond(A)
```

$$\frac{33}{10}$$

```
linalg::cond(A, Infinity)
```

$$\frac{33}{10}$$

```
linalg::cond(A, 1)
```

$$3$$

```
linalg::cond(A, Frobenius)
```

$$\frac{\sqrt{2}\,\sqrt{43}\,\sqrt{251}}{40}$$

The result for the spectral norm is too complex, so we want the floating valuation. The tiny imaginary part is a rounding artifact:

```
linalg::cond(A, 2);
float(%)
```

$$\sqrt{\frac{343}{9\,\sigma_1}+\sigma_1+\frac{43}{3}}\,\sqrt{\frac{11401}{5760000\,\sigma_2}+\sigma_2+\frac{251}{2400}}$$

where

$$\sigma_1=\left(\frac{3970}{27}+\frac{\sqrt{27}\,\sqrt{910841}\,i}{27}\right)^{1/3}$$

$$\sigma_2=\left(\frac{705851}{13824000000}+\frac{\sqrt{910841}\,\sqrt{176947200000000}\,i}{176947200000000}\right)^{1/3}$$

$$2.223147175+1.386849201\,10^{-18}\,i$$

If *A* contains at least one floating-point number, the result will be computed numerically.

```
B := A:
B[1,1] := float(B[1,1]):
linalg::cond(B, 2)
```

$$2.223147175$$

Example 2

We define the 2×2 matrix C.

```
C := matrix([[1,-2],[3,-4]])
```

$$\begin{pmatrix} 1 & -2 \\ 3 & -4 \end{pmatrix}$$

Now we calculate the condition number of C for some matrix norms.

```
linalg::cond(C,1)
```

$$21$$

```
linalg::cond(C,Infinity)
```

$$21$$

```
linalg::cond(C,Frobenius); Simplify(%);
```

$$\frac{\sqrt{2}\,\sqrt{15}\,\sqrt{30}}{2}$$

$$15$$

Example 3

Hilbert matrices are very ill-conditioned:

```
linalg::cond( linalg::hilbert(3) )
```

$$748$$

```
linalg::cond( linalg::hilbert(5) )
```

$$943656$$

```
linalg::cond( linalg::hilbert(7) )
```

$$\frac{1970389773}{2}$$

Parameters

M

Square matrix of domain type `Dom::Matrix`

Options

`Frobenius`, **`Infinity`**, **`Spectral`**

The index of the matrix norm.

Return Values

Arithmetical expression.

See Also

MuPAD Functions
`norm`

linalg::crossProduct

Cross product of three-dimensional vectors

Syntax

```
linalg::crossProduct(u, v)
```

Description

`linalg::crossProduct(u, v)` computes the cross product of the three-dimensional vectors $\vec{u}$ and $\vec{v}$. This is the vector

$$\vec{u} \times \vec{v} = \begin{pmatrix} u_2\, v_3 - u_3\, v_2 \\ u_3\, v_1 - u_1\, v_3 \\ u_1\, v_2 - u_2\, v_1 \end{pmatrix}.$$

The vectors must be defined over the same component ring.

Examples

Example 1

We define two vectors:

```
a := matrix([[1, 2, 3]]);
b := matrix([[-1, 0, 1]])
```

$$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

$$\begin{pmatrix} -1 & 0 & 1 \end{pmatrix}$$

The cross product of these two vectors is a vector $\vec{c}$ which is orthogonal to $\vec{a}$ and $\vec{b}$:

```
c:= linalg::crossProduct(a, b)
```

$$(2 \quad -4 \quad 2)$$

```
linalg::scalarProduct(a, c), linalg::scalarProduct(b, c)
```

$$0, 0$$

Parameters

u, v

3-dimensional vectors, i.e., either two 3 ×1 or two 1 ×3 matrices of a domain of category `Cat::Matrix`

Return Values

Vector of the same domain type as `u`.

See Also

MuPAD Functions

`linalg::scalarProduct`

linalg::delCol

Delete matrix columns

Syntax

```
linalg::delCol(A, c)
```

```
linalg::delCol(A, c1 .. c2)
```

```
linalg::delCol(A, list)
```

Description

`linalg::delCol(A, c)` returns a copy of the matrix *A* in which the column with index *c* is deleted.

`linalg::delCol(A, c1.. c2)` deletes those columns whose indices are in the range `c1.. c2`. If `c2< c1` then the input matrix `A` is returned.

`linalg::delCol(A, list)` deletes those columns whose indices are contained in `list`.

If all columns are deleted then `NIL` is returned.

Examples

Example 1

We define the following matrix:

```
A := matrix([[1, 2, 3, 4], [5, 6, 7, 8]])
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

and demonstrate the three different input formats for `linalg::delCol`:

```
linalg::delCol(A, 2)
```

$$\begin{pmatrix} 1 & 3 & 4 \\ 5 & 7 & 8 \end{pmatrix}$$

```
linalg::delCol(A, [1, 3])
```

$$\begin{pmatrix} 2 & 4 \\ 6 & 8 \end{pmatrix}$$

```
linalg::delCol(A, 2..4)
```

$$\begin{pmatrix} 1 \\ 5 \end{pmatrix}$$

Example 2

We compute the inverse of the 2×2 matrix:

```
MatQ := Dom::Matrix(Dom::Rational):
A := MatQ([[3, 2], [5, -4]])
```

$$\begin{pmatrix} 3 & 2 \\ 5 & -4 \end{pmatrix}$$

by appending the 2×2 identity matrix to the right side of A and applying the Gauss-Jordan algorithm provided by the function `linalg::gaussJordan`:

```
B := linalg::gaussJordan(A . MatQ::identity(2))
```

$$\begin{pmatrix} 1 & 0 & \frac{2}{11} & \frac{1}{11} \\ 0 & 1 & \frac{5}{22} & -\frac{3}{22} \end{pmatrix}$$

We get the inverse of `A` by deleting the first two columns of the matrix `B`:

```
AI := linalg::delCol(B, 1..2)
```

$$\begin{pmatrix} \frac{2}{11} & \frac{1}{11} \\ \frac{5}{22} & -\frac{3}{22} \end{pmatrix}$$

Finally, we check the result:

```
A * AI, AI * A
```

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Note: The inverse of `A` can be computed directly by entering `1/A`.

Parameters

`A`

An $m \times n$ matrix of a domain of category `Cat::Matrix`

`c`

The column index: a positive integer less or equal to n

c_1 `..` c_2

A range of column indices (positive integers less or equal to n)

`list`

A list of column indices (positive integers less or equal to n)

Return Values

Matrix of a domain of category `Cat::Matrix(R)`, where `R` is the component ring of `A`, or `NIL`.

See Also

MuPAD Functions

`linalg::col` | `linalg::delRow` | `linalg::row`

More About

- "Swap and Delete Rows and Columns"

linalg::delRow

Delete matrix rows

Syntax

```
linalg::delRow(A, r)
```

```
linalg::delRow(A, r1 .. r2)
```

```
linalg::delRow(A, list)
```

Description

`linalg::delRow(A, r)` returns a copy of the matrix A in which the row with index r is deleted.

`linalg::delRow(A, r1.. r2)` deletes those rows whose indices are in the range `r1.. r2`. If `r2< r1` then the input matrix `A` is returned.

`linalg::delRow(A, list)` deletes those rows whose indices are contained in `list`.

If all rows are deleted then `NIL` is returned.

Examples

Example 1

We define the following matrix:

```
A := matrix([[1, 2], [3, 4], [5, 6], [7, 8]])
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{pmatrix}$$

and illustrate the three different input formats for `linalg::delRow`:

```
linalg::delRow(A, 2)
```

$$\begin{pmatrix} 1 & 2 \\ 5 & 6 \\ 7 & 8 \end{pmatrix}$$

```
linalg::delRow(A, [1, 4])
```

$$\begin{pmatrix} 3 & 4 \\ 5 & 6 \end{pmatrix}$$

```
linalg::delRow(A, 2..4)
```

$$\begin{pmatrix} 1 & 2 \end{pmatrix}$$

Parameters

`A`

An $m \times n$ matrix of a domain of category `Cat::Matrix`

`r`

The row index: a positive integer less or equal to m

r_1 `..` r_2

A range of row indices (positive integers less or equal to m)

`list`

A list of row indices (positive integers less or equal to m)

Return Values

Matrix of a domain of category `Cat::Matrix(R)`, where `R` is the component ring of `A` or `NIL`.

See Also

MuPAD Functions

`linalg::col` | `linalg::delCol` | `linalg::row`

More About

- "Swap and Delete Rows and Columns"

linalg::eigenvalues

Eigenvalues of a matrix

Syntax

```
linalg::eigenvalues(A, <Multiple>)
```

Description

`linalg::eigenvalues(A)` returns a list of the eigenvalues of the matrix A.

A floating-point approximation of the eigenvalues is computed with `numeric::eigenvalues`, if the matrix `A` is defined over the component ring `Dom::Float` (see "Example 1" on page 15-46). In this case it is recommended to call `numeric::eigenvalues` directly for a better efficiency.

The eigenvalues are obtained by computing the zeros of the characteristic polynomial of `A`. The solver `solve` must be able to compute the roots of the characteristic polynomial over the component ring of `A`.

Examples

Example 1

We compute the eigenvalues of the matrix

$$A = \begin{pmatrix} 1 & 4 & 2 \\ 1 & 4 & 2 \\ 2 & 5 & 3 \end{pmatrix}.$$

```
A := matrix([[1, 4, 2], [1, 4, 2], [2, 5, 3]]):
linalg::eigenvalues(A)
```

$$\{0, 4 - \sqrt{15}, \sqrt{15} + 4\}$$

If we consider the matrix over the domain `Dom::Float`, then the call of `linalg::eigenvalues(A)` results in a numerical computation of the eigenvalues of `A` via `numeric::eigenvalues`:

```
B := Dom::Matrix(Dom::Float)(A):
linalg::eigenvalues(B)
```

$$\left\{-1.370431546\ 10^{-15},\ 0.1270166538,\ 7.872983346\right\}$$

Example 2

With the option `Multiple` we get the information about the algebraic multiplicity of each eigenvalue:

```
C := Dom::Matrix(Dom::Rational)(4, 4, [[-3], [0, 6]])
```

$$\begin{pmatrix} -3 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

```
linalg::eigenvalues(C, Multiple)
```

$$[[-3,\ 1],\ [0,\ 2],\ [6,\ 1]]$$

Parameters

A

A square matrix of a domain of category `Cat::Matrix`

Options

Multiple

Returns a list of sublists, where each sublist contains an eigenvalue of `A` and its algebraic multiplicity. Note that due to rounding errors, this may lead to wrong results in cases where multiple eigenvalues exist and `numeric::eigenvalues` is used.

Return Values

Set of the eigenvalues of A, or a list of inner lists when the option `Multiple` is given (see below).

See Also

MuPAD Functions

`linalg::charpoly` | `linalg::eigenvectors` | `numeric::eigenvalues` | `solve`

linalg::eigenvectors

Eigenvectors of a matrix

Syntax

```
linalg::eigenvectors(A)
```

Description

`linalg::eigenvectors(A)` computes the eigenvalues and eigenvectors of the matrix A.

A floating-point approximation of the eigenvalues and the eigenvectors is computed using `numeric::eigenvectors`, if the matrix `A` is defined over the component ring `Dom::Float` (see "Example 1" on page 15-50). In this case it is recommended to call `numeric::eigenvalues` directly for a better efficiency.

`linalg::eigenvectors` works as follows: For each eigenvalue λ of the $n \times n$ matrix `A`, a basis for the kernel of ($\lambda\, I_n$ - A), the eigenspace of `A` with respect to the eigenvalue λ, is computed using the Gauss-Jordan algorithm (see `linalg::gaussJordan`). Here, I_n denotes the $n \times n$ identity matrix.

The eigenvectors are of the domain `Dom::Matrix(R)`, where `R` is the component ring of `A`.

The component ring of the matrix `A` must be a field, i.e., a domain of category `Cat::Field`, for which the solver `solve` is able to compute the zeros of a polynomial.

It can happen that a basis for the eigenspace of `A` with respect to a certain eigenvalue cannot be computed (e.g., if the component ring does not have a canonical representation of the zero element). In this case `linalg::eigenvectors` answers with a warning message and returns `FAIL`.

Examples

Example 1

We compute the eigenvalues and the eigenvectors of the matrix

$$A = \begin{pmatrix} 1 & -3 & 3 \\ 6 & -10 & 6 \\ 6 & 6 & 4 \end{pmatrix}$$:

```
A := Dom::Matrix(Dom::Rational)(
  [[1, -3, 3], [6, -10, 6], [6, 6, 4]]
):
Ev:= linalg::eigenvectors(A)
```

$$\left[\left[-11,\ 1,\ \left[\begin{pmatrix} -\frac{7}{10} \\ -\frac{9}{5} \\ 1 \end{pmatrix}\right]\right],\ \left[-2,\ 1,\ \left[\begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}\right]\right],\ \left[8,\ 1,\ \left[\begin{pmatrix} \frac{1}{4} \\ \frac{5}{12} \\ 1 \end{pmatrix}\right]\right]\right]$$

The matrix A is diagonalizable. Hence, we extract the eigenvectors and combine them to a matrix P such that P^-1 * A * P is the diagonal matrix whose diagonal entries are given by the corresponding eigenvalues:

```
Eigenvectors:= Ev[1][3][1], Ev[2][3][1], Ev[3][3][1]
```

$$\begin{pmatrix} -\frac{7}{10} \\ -\frac{9}{5} \\ 1 \end{pmatrix},\ \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix},\ \begin{pmatrix} \frac{1}{4} \\ \frac{5}{12} \\ 1 \end{pmatrix}$$

```
P:= Eigenvectors[1].Eigenvectors[2].Eigenvectors[3]
```

$$\begin{pmatrix} -\frac{7}{10} & -1 & \frac{1}{4} \\ -\frac{9}{5} & 0 & \frac{5}{12} \\ 1 & 1 & 1 \end{pmatrix}$$

```
P^-1 * A * P
```

$$\begin{pmatrix} -11 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 8 \end{pmatrix}$$

A more skillful way of extracting the above eigenvectors from the output generated by `linalg::eigenvectors` is the following:

```
map(Ev, op@op, 3)
```

$$\left[\begin{pmatrix} -\frac{7}{10} \\ -\frac{9}{5} \\ 1 \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} \frac{1}{4} \\ \frac{5}{12} \\ 1 \end{pmatrix}\right]$$

If we consider the matrix `A` over the domain `Dom::Float`, the call of `linalg::eigenvectors(A)` results in a numerical computation of the eigenvalues and the eigenvectors of `A` via the function `numeric::eigenvectors`:

```
B := Dom::Matrix(Dom::Float)(A):
linalg::eigenvectors(B)
```

$$\left[\left[8.0,\ 1,\ \left[\begin{pmatrix} 0.2248595067 \\ 0.3747658445 \\ 0.8994380268 \end{pmatrix}\right]\right], \left[-2.0,\ 1,\ \left[\begin{pmatrix} 0.7071067812 \\ 0.0 \\ -0.7071067812 \end{pmatrix}\right]\right],\right.$$
$$\left.\left[-11.0,\ 1,\ \left[\begin{pmatrix} 0.3218603429 \\ 0.8276408818 \\ -0.4598004899 \end{pmatrix}\right]\right]\right]$$

Parameters

A

A square matrix of a domain of category `Cat::Matrix`

Return Values

List of sublists, where each sublist consists of an eigenvalue λ of A, its algebraic multiplicity and a basis for the eigenspace of λ. If a basis of an eigenspace cannot be computed, FAIL is returned.

See Also

MuPAD Functions

`linalg::eigenvalues` | `linalg::nullspace` | `numeric::eigenvectors`

linalg::expr2Matrix

Construct a matrix from equations

Syntax

```
linalg::expr2Matrix(eqns, <vars, R>, <Include>)
```

Description

`linalg::expr2Matrix(eqns, vars)` constructs the extended coefficient matrix $\left(A, \vec{b}\right)$ of the system of m linear equations in `eqns` with respect to the n indeterminates in `vars`. The vector $\vec{b}$ is the right-hand side of this system.

`linalg::expr2Matrix` returns the extended coefficient matrix $M = \left(A, \vec{b}\right)$. The right-hand side vector $\vec{b}$ can be extracted from the matrix M by `linalg::col(M, n + 1)`.

The coefficient matrix A can be extracted by `linalg::delCol(M, n + 1)`.

Arithmetical expressions in `eqns` are considered as equations with right hand-sides zero.

If no variables are given, then the indeterminates of the equations are determined with the function `indets` and the option `PolyExpr`, i.e., the left-hand sides of the equations are considered as polynomial expressions.

If no component ring `R` is given then the standard domain `Dom::ExpressionField()` is chosen as the component ring of the extended coefficient matrix.

The coefficients of the linear equations are converted into elements of the component ring `R`. An error message is returned if this is not possible.

Examples

Example 1

The extended coefficient matrix of the system $x + y + z = 1$, $2\,y - z + 5 = 0$ of linear equations in the variables x, y, z is the following 2×4 matrix:

```
delete x, y, z:
Ab := linalg::expr2Matrix(
  [x + y + z = 1, 2*y - z + 5], [x, y, z], Dom::Real
)
```

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 2 & -1 & -5 \end{pmatrix}$$

We use `linalg::matlinsolve` to compute the general solution of this system:

```
linalg::matlinsolve(Ab)
```

$$\left[\begin{pmatrix} \frac{7}{2} \\ -\frac{5}{2} \\ 0 \end{pmatrix}, \left[\begin{pmatrix} -\frac{3}{2} \\ \frac{1}{2} \\ 1 \end{pmatrix}\right]\right]$$

The coefficient matrix or the right-hand side vector can be be extracted from the matrix `Ab` in the following way:

```
A := linalg::delCol(Ab, 4); b := linalg::col(Ab, 4)
```

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 2 & -1 \end{pmatrix}$$

$$\begin{pmatrix} 1 \\ -5 \end{pmatrix}$$

Example 2

The following two inputs lead to different linear systems:

```
delete x, y, z:
linalg::expr2Matrix([x + y + z = 1, 2*y - z + 5 = x]),
linalg::expr2Matrix([x + y + z = 1, 2*y - z + 5 = x], [x, y])
```

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ -1 & 2 & -1 & -5 \end{pmatrix}, \begin{pmatrix} 1 & 1 & 1-z \\ -1 & 2 & z-5 \end{pmatrix}$$

Example 3

Note the difference between calling `linalg::expr2Matrix` with and without option `Include`:

```
delete x, y:
linalg::expr2Matrix([x + y = 1, 2*x - y = 3], [x, y])
```

$$\begin{pmatrix} 1 & 1 & 1 \\ 2 & -1 & 3 \end{pmatrix}$$

```
linalg::expr2Matrix([x + y = 1, 2*x - y = 3], [x, y], Include)
```

$$\begin{pmatrix} 1 & 1 & -1 \\ 2 & -1 & -3 \end{pmatrix}$$

Parameters

`eqns`

The system of linear equations, i.e. a set or list of expressions of type `"_equal"`

`vars`

A set or list of indeterminates

`R`

A commutative ring, i.e., a domain of category `Cat::CommutativeRing`

Options

`Include`

Appends the negative of the right-hand side vector $\vec{b}$ to the coefficient matrix A of the given system of linear equations. The result is the $m \times (n+1)$ matrix $\left(A, -\vec{b}\right)$.

Return Values

$m \times (n+1)$ matrix of the domain `Dom::Matrix(R)`.

See Also

MuPAD Functions

`indets` | `linalg::matlinsolve` | `linsolve`

linalg::factorCholesky

The Cholesky decomposition of a matrix

Syntax

```
linalg::factorCholesky(A, <NoCheck>, <Real>)
```

Description

`linalg::factorCholesky(A)` computes the Cholesky factorization of a Hermitian positive definite matrix A and returns a lower triangular matrix L, such that $L\,L^H = A$. Here, L^H is the Hermitian conjugate of L (the complex conjugate of the transpose).

The component ring of `A` must be a field (a domain of category `Cat::Field`).

If A is not a Hermitian positive definite matrix, then `linalg::factorCholesky` throws an error. If you use `NoCheck`, `linalg::factorCholesky` does not check whether the matrix is Hermitian positive definite. See "Example 2" on page 15-58.

If you use `Real`, then `linalg::factorCholesky` assumes that `A` is real and symmetric and, therefore, does not apply complex conjugate in the course of the algorithm.

`linalg::factorCholesky` returns `FAIL` if it fails to compute the matrix L over the component ring of `A`. (The algorithm requires the computation of square roots of some elements in L).

Environment Interactions

Properties of identifiers are taken into account.

Examples

Example 1

Define matrix `S` as follows:

```
S := Dom::Matrix(Dom::Rational)(
  [[4, -2, 4, 2], [-2, 10, -2, -7], [4, -2, 8, 4], [2, -7, 4, 7]]
)
```

$$\begin{pmatrix} 4 & -2 & 4 & 2 \\ -2 & 10 & -2 & -7 \\ 4 & -2 & 8 & 4 \\ 2 & -7 & 4 & 7 \end{pmatrix}$$

Compute the Cholesky factorization of S:

```
L := linalg::factorCholesky(S)
```

$$\begin{pmatrix} 2 & 0 & 0 & 0 \\ -1 & 3 & 0 & 0 \\ 2 & 0 & 2 & 0 \\ 1 & -2 & 1 & 1 \end{pmatrix}$$

Verify the result:

```
is(L * htranspose(L) = S)
```

$$\text{TRUE}$$

Example 2

Define matrix H as follows:

```
H := matrix([[a, b], [b, a]])
```

$$\begin{pmatrix} a & b \\ b & a \end{pmatrix}$$

`linalg::factorCholesky` cannot compute the Cholesky factorization because it cannot prove that H is a Hermitian matrix:

```
linalg::factorCholesky(H)
```

```
Error: A Hermitian matrix is expected. [linalg::factorCholesky]
```

If you assume that `a` and `b` are real, then matrix `H` is Hermitian. Still, `linalg::factorCholesky` cannot compute the Cholesky factorization:

```
linalg::factorCholesky(H) assuming a in R_ and b in R_
```

```
Error: Cannot check whether the matrix component is positive. [linalg::factorCholesky]
```

Use the `NoCheck` option to skip checking whether this matrix is Hermitian positive definite. Now, `linalg::factorCholesky` computes the factorization:

```
L := linalg::factorCholesky(H, NoCheck)
```

$$\begin{pmatrix} \sqrt{a} & 0 \\ \frac{b}{\sqrt{a}} & \frac{\sqrt{a\,|a| - |b|^2}}{\sqrt{|a|}} \end{pmatrix}$$

This result is not generally valid:

```
L*htranspose(L) = H
```

$$\begin{pmatrix} \sqrt{a}\,\overline{\sqrt{a}} & \frac{\sqrt{a}\,\overline{b}}{\overline{\sqrt{a}}} \\ \frac{b\,\overline{\sqrt{a}}}{\sqrt{a}} & \frac{\overline{\sqrt{a\,|a| - |b|^2}}\,\sqrt{a\,|a| - |b|^2}}{|a|} + \frac{b\,\overline{b}}{\sqrt{a}\,\overline{\sqrt{a}}} \end{pmatrix} = \begin{pmatrix} a & b \\ b & a \end{pmatrix}$$

It is valid for `0 < a < b`:

```
simplify(L*htranspose(L) = H) assuming 0 < b < a
```

$$\begin{pmatrix} a & b \\ b & a \end{pmatrix} = \begin{pmatrix} a & b \\ b & a \end{pmatrix}$$

Example 3

Compute the Cholesky factorization of matrix `H` using `NoCheck` to skip checking whether it is Hermitian positive definite. By default, `linalg::factorCholesky` computes a Hermitian factorization $A = L\,L^H$. Thus, the result contains complex conjugates (implied by $|a| = a^*\bar{a}$).

```
H := matrix([[a, b], [b, a]]):
L := linalg::factorCholesky(H, NoCheck)
```

$$\begin{pmatrix} \sqrt{a} & 0 \\ \frac{b}{\sqrt{a}} & \frac{\sqrt{a\,|a| - |b|^2}}{\sqrt{|a|}} \end{pmatrix}$$

To avoid complex conjugates in the result, use `Real`:

```
L := linalg::factorCholesky(H, NoCheck, Real)
```

$$\begin{pmatrix} \sqrt{a} & 0 \\ \frac{b}{\sqrt{a}} & \sqrt{\frac{a^2 - b^2}{a}} \end{pmatrix}$$

With this option, `linalg::factorCholesky` computes a symmetric factorization $A = L\,L^t$ instead of a Hermitian factorization $A = L\,L^H$:

```
simplify(L*transpose(L) = H)
```

$$\begin{pmatrix} a & b \\ b & a \end{pmatrix} = \begin{pmatrix} a & b \\ b & a \end{pmatrix}$$

Parameters

A

Square matrix of a domain of category `Cat::Matrix`.

Options

NoCheck

Skip checking whether A is Hermitian positive definite. When you use this option, the identity $L\,L^H = A$ is guaranteed to hold only if A is a Hermitian positive definite matrix.

`Real`

Compute the Cholesky factorization assuming that matrix A is symmetric and all its symbolic parameters are real. In this case, the transpose of the matrix is its Hermitian transpose. Use this option if A contains symbolic parameters, and you want to avoid complex conjugates. When using this option, the identity $L\,L^T = A$ is guaranteed to hold.

Return Values

Matrix of the same domain type as `A`, or the value `FAIL`.

Algorithms

The Cholesky factorization of a Hermitian positive definite $n \times n$ matrix A is a decomposition of A in a product $L\,L^H = A$, such that L is a lower triangular matrix with positive entries on the main diagonal. L is called the "Cholesky factor" of A.

If $L = (l_{i,j})$, where $1 \le i \le n$, $1 \le j \le n$, is the Cholesky factor of A, then $\det A = \left(\prod_{i=1}^{n} l_{ii}\right)^2$.

See Also

MuPAD Functions

`linalg::isHermitian` | `linalg::isPosDef`

linalg::factorLU

LU-decomposition of a matrix

Syntax

```
linalg::factorLU(A)
```

Description

`linalg::factorLU(A)` computes an LU-decomposition of an $m \times n$ matrix A, i.e., a decomposition of the A into an $m \times m$ lower triangular matrix L and an $m \times n$ upper triangular matrix U such that $PA = LU$, where P is a permutation matrix.

The diagonal entries of the lower triangular matrix L are equal to one (*Doolittle*-decomposition). The diagonal entries of U are the pivot elements used during the computation.

The matrices L and U are unique.

`pivindex` is a list `[ r1, r2, ...]` representing the row exchanges of A in the pivoting steps, i.e., $B = PA = LU$, where $b_{ij} = a_{r_i, j}$.

A floating-point approximation of the decomposition is computed using `numeric::factorLU`, if the matrix `A` is defined over the component ring `Dom::Float`. In this case it is recommended to call `numeric::factorLU` directly for a better efficiency.

The algorithm also works for singular A. In this case either L or U is singular.

L and U are nonsingular if and only if A is nonsingular.

The component ring of the matrix `A` must be a field, i.e., a domain of category `Cat::Field`.

Examples

Example 1

We compute an LU-decomposition of the real matrix:

```
A := Dom::Matrix(Dom::Real)(
  [[2, -3, -1], [1, 1, -1], [0, 1, -1]]
)
```

$$\begin{pmatrix} 2 & -3 & -1 \\ 1 & 1 & -1 \\ 0 & 1 & -1 \end{pmatrix}$$

```
[L, U, pivlist] := linalg::factorLU(A)
```

$$\left[\begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ 0 & \frac{2}{5} & 1 \end{pmatrix}, \begin{pmatrix} 2 & -3 & -1 \\ 0 & \frac{5}{2} & -\frac{1}{2} \\ 0 & 0 & -\frac{4}{5} \end{pmatrix}, [1, 2, 3]\right]$$

The lower triangular matrix L is the first element und the upper triangular matrix U is the second element of the list LU. The product of these two matrices is equal to the input matrix A:

```
L * U
```

$$\begin{pmatrix} 2 & -3 & -1 \\ 1 & 1 & -1 \\ 0 & 1 & -1 \end{pmatrix}$$

Example 2

An LU-decomposition of the 3×2 matrix:

```
A := Dom::Matrix(Dom::Real)([[2, -3], [1, 2], [2, 3]])
```

$$\begin{pmatrix} 2 & -3 \\ 1 & 2 \\ 2 & 3 \end{pmatrix}$$

gives a 3×3 lower triangular matrix and a 3×2 upper triangular matrix:

```
[L, U, pivlist] := linalg::factorLU(A)
```

$$\left[\begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ 1 & \frac{12}{7} & 1 \end{pmatrix}, \begin{pmatrix} 2 & -3 \\ 0 & \frac{7}{2} \\ 0 & 0 \end{pmatrix}, [1, 2, 3]\right]$$

```
L * U
```

$$\begin{pmatrix} 2 & -3 \\ 1 & 2 \\ 2 & 3 \end{pmatrix}$$

Example 3

To compute the LU-decomposition of the matrix:

```
A := matrix([[1, 2, -1], [0, 0, 3], [0, 2, -1]])
```

$$\begin{pmatrix} 1 & 2 & -1 \\ 0 & 0 & 3 \\ 0 & 2 & -1 \end{pmatrix}$$

one row interchange is needed, and we therefore get a non-trivial permutation list:

```
[L, U, pivlist] := linalg::factorLU(A)
```

$$\left[\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 2 & -1 \\ 0 & 2 & -1 \\ 0 & 0 & 3 \end{pmatrix}, [1, 3, 2]\right]$$

The corresponding permutation matrix is the following:

```
P := linalg::swapRow(matrix::identity(3), 3, 2)
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Hence, we have a decomposition of A into the product of the three matrices $\frac{1}{P}$, L and U as follows:

```
P^(-1) * L * U
```

$$\begin{pmatrix} 1 & 2 & -1 \\ 0 & 0 & 3 \\ 0 & 2 & -1 \end{pmatrix}$$

Example 4

You may compute an LU-decomposition of a matrix with symbolic components, such as:

```
delete a, b, c, d:
A := matrix([[a, b], [c, d]])
```

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

The diagonal entries of the matrix U are the pivot elements used during the computation. They must be non-zero, if the inverse of U is needed:

```
[L, U, pivlist] := linalg::factorLU(A)
```

$$\left[\begin{pmatrix} 1 & 0 \\ \frac{c}{a} & 1 \end{pmatrix}, \begin{pmatrix} a & b \\ 0 & d - \frac{b\,c}{a} \end{pmatrix}, [1, 2]\right]$$

For example, if we use this decomposition to solve the linear system $A\,\vec{x} = \vec{b}$ for arbitrary vectors $\vec{b} = (b_1, b_2)^t$, then the following result is only correct for $a \neq 0$ and $d - \frac{b\,c}{a} \neq 0$:

```
delete b1, b2:
linalg::matlinsolveLU(L, U, matrix([b1, b2]))
```

$$\begin{pmatrix} \dfrac{b1 - \dfrac{b\,(a\,b2 - b1\,c)}{a\,d - b\,c}}{a} \\ \dfrac{a\,b2 - b1\,c}{a\,d - b\,c} \end{pmatrix}$$

Parameters

A

A matrix of a domain of category `Cat::Matrix`

Return Values

List `[L, U, pivindex]` with the two matrices L and U of the domain `Dom::Matrix(R)` and a list `pivindex` of positive integers. `R` is the component ring of `A`.

Algorithms

The following algorithm for solving the system $A\,\vec{x} = \vec{b}$ with a nonsingular matrix A uses LU-decomposition:

1. Compute a LU-decomposition of A: $A = L\,U$.
2. Solve $\vec{y} = \dfrac{\vec{b}}{L}$ by forward substitution.
3. Solve $\vec{x} = \dfrac{\vec{y}}{R}$ by backward substitution.

The LU-decomposition of a matrix A is useful for solving several systems of linear equations $A\,\vec{x} = \vec{b}$ with the same coefficient matrix A and several right-hand side vectors $\vec{b}$, because then step one of the algorithm above needs to be done only once.

See Also

MuPAD Functions

`linalg::factorCholesky` | `linalg::factorQR` | `linalg::inverseLU` | `linalg::matlinsolveLU` | `lllint` | `numeric::factorLU`

linalg::factorQR

QR-decomposition of a matrix

Syntax

```
linalg::factorQR(A)
```

Description

`linalg::factorQR(A)` computes an QR-decomposition of an $m \times n$ matrix A, i.e., a decomposition of A into an $m \times m$ unitary matrix Q and an $m \times n$ upper triangular matrix R such that $Q\,R = A$.

`linalg::factorQR` uses Gram-Schmidt orthonormalization to compute the decomposition.

For a singular or non-square matrix `A` the QR-decomposition of `A` is not unique.

The columns of Q form an orthonormal basis with respect to the scalar product of two vectors, defined by `linalg::scalarProduct`, and the 2-norm of two vectors (see the method `"norm"` of the domain constructor `Dom::Matrix`).

If the component ring of `A` does not define the method `"conjugate"`, then the factor Q is orthogonal instead of unitary.

If the columns of `A` cannot be orthonormalized then `FAIL` is returned.

If `A` is a matrix over the domain `Dom::Float` and the computations are based on the standard scalar product, then the use of the corresponding function from the numeric library (`numeric::factorQR`) is recommended.

Even if `A` is defined over the real or the complex numbers the call of `numeric::factorQR` with the option `Symbolic` is recommended for better efficiency.

The component ring of the matrix `A` must be a field, i.e., a domain of category `Cat::Field`.

Examples

Example 1

We compute the QR-decomposition of a real matrix:

```
A := Dom::Matrix(Dom::Real)(
  [[2, -3, -1], [1, 1, -1], [0, 1, -1]]
)
```

$$\begin{pmatrix} 2 & -3 & -1 \\ 1 & 1 & -1 \\ 0 & 1 & -1 \end{pmatrix}$$

```
QR := linalg::factorQR(A)
```

$$\left[\begin{pmatrix} \frac{2\sqrt{5}}{5} & -\frac{\sqrt{6}}{6} & -\frac{\sqrt{2}\sqrt{15}}{30} \\ \frac{\sqrt{5}}{5} & \frac{\sqrt{6}}{3} & \frac{\sqrt{2}\sqrt{15}}{15} \\ 0 & \frac{\sqrt{6}}{6} & -\frac{\sqrt{2}\sqrt{15}}{6} \end{pmatrix}, \begin{pmatrix} \sqrt{5} & -\sqrt{5} & -\frac{3\sqrt{5}}{5} \\ 0 & \sqrt{6} & -\frac{\sqrt{6}}{3} \\ 0 & 0 & \frac{2\sqrt{2}\sqrt{15}}{15} \end{pmatrix} \right]$$

The orthogonal matrix Q is the first element und the upper triangular matrix R is the second element of the list `QR`. The product of these two matrices is equal to the input matrix `A`:

```
QR[1] * QR[2]
```

$$\begin{pmatrix} 2 & -3 & -1 \\ 1 & 1 & -1 \\ 0 & 1 & -1 \end{pmatrix}$$

Example 2

The QR-decomposition of the 3×2 matrix:

```
B := Dom::Matrix(Dom::Real)(
  [[2, -3], [1, 2], [2, 3]]
)
```

$$\begin{pmatrix} 2 & -3 \\ 1 & 2 \\ 2 & 3 \end{pmatrix}$$

yields a 3×3 orthogonal matrix and a 3×2 upper triangular matrix:

```
QR := linalg::factorQR(B)
```

$$\left[\begin{pmatrix} \frac{2}{3} & -\frac{31\sqrt{194}}{582} & \frac{\sqrt{194}}{194} \\ \frac{1}{3} & \frac{8\sqrt{194}}{291} & \frac{6\sqrt{194}}{97} \\ \frac{2}{3} & \frac{23\sqrt{194}}{582} & -\frac{7\sqrt{194}}{194} \end{pmatrix}, \begin{pmatrix} 3 & \frac{2}{3} \\ 0 & \frac{\sqrt{194}}{3} \\ 0 & 0 \end{pmatrix}\right]$$

```
QR[1] * QR[2]
```

$$\begin{pmatrix} 2 & -3 \\ 1 & 2 \\ 2 & 3 \end{pmatrix}$$

For this example we may call `numeric::factorQR(B, Symbolic)` instead, which in general is faster than `linalg::factorQR`:

```
QR := numeric::factorQR(B, Symbolic)
```

$$\left[\begin{pmatrix} \frac{2}{3} & -\frac{31\sqrt{194}}{582} & \frac{\sqrt{194}}{194} \\ \frac{1}{3} & \frac{8\sqrt{194}}{291} & \frac{6\sqrt{194}}{97} \\ \frac{2}{3} & \frac{23\sqrt{194}}{582} & -\frac{7\sqrt{194}}{194} \end{pmatrix}, \begin{pmatrix} 3 & \frac{2}{3} \\ 0 & \frac{\sqrt{194}}{3} \\ 0 & 0 \end{pmatrix}\right]$$

Parameters

A

A matrix of a domain of category `Cat::Matrix`

Return Values

List `[Q, R]` of the two matrices Q and R (of the same domain type as `A`), or the value `FAIL`.

Algorithms

The QR-decomposition can be used to generate a least square solution to an overdetermined system of linear equations. If $A\,\vec{x} = \vec{b}$, then $R\,\vec{x} = Q^t\,\vec{b}$ can be solved via backward substitution.

See Also

MuPAD Functions

`linalg::factorCholesky` | `linalg::factorLU` | `lllint` | `numeric::factorQR`

linalg::frobeniusForm

Frobenius form of a matrix

Syntax

```
linalg::frobeniusForm(A, <All>)
```

Description

`linalg::frobeniusForm(A)` returns the Frobenius form of the matrix A, also called the Rational Canonical form of A.

`linalg::frobeniusForm(A, All)` computes the Frobenius form R of `A` and a transformation matrix P such that PRP^{-1}.

The Frobenius form as computed by `linalg::frobeniusForm` is unique (see below).

The component ring of `A` must be a field, i.e., a domain of category `Cat::Field`.

Examples

Example 1

The Frobenius form of the following matrix over $\mathbb{C}$:

```
A := Dom::Matrix(Dom::Complex)(
  [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
)
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

is the matrix:

```
R := linalg::frobeniusForm(A)
```

$$\begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 18 \\ 0 & 1 & 15 \end{pmatrix}$$

The transformation matrix P can be selected from the list `[R, P]`, which is the result of `linalg::frobeniusForm` with option `All`:

```
P := linalg::frobeniusForm(A, All)[2]
```

$$\begin{pmatrix} 1 & 1 & 30 \\ 0 & 4 & 66 \\ 0 & 7 & 102 \end{pmatrix}$$

We check the result:

```
P * R * P^(-1)
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Parameters

A

A square matrix of a domain of category `Cat::Matrix`

Options

All

Returns the list `[R, P]` with the Frobenius form R of `A` and a transformation matrix P such that $A = P\,R\,P^{-1}$.

Return Values

Matrix of the same domain type as `A`, or the list `[R, P]` when the option `All` is given.

Algorithms

The Frobenius form of a square matrix A is the matrix

$$R = \begin{pmatrix} R_1 & & \\ & \ddots & \\ & & R_r \end{pmatrix},$$

where $R_1, \ldots, R_r$ are known as companion matrices and have the form:

$$R_i = \begin{pmatrix} 0 & \cdots & 0 & -a_0 \\ 1 & & & -a_1 \\ & \ddots & & \vdots \\ & & 0 & \vdots \\ & & 1 & -a_{n_i-1} \end{pmatrix}, \; i = 1, \ldots, r$$

.

In the last column of the companion matrix R_i, you see the coefficients of its minimal polynomial in ascending order, i.e., the polynomial $m_i := X^{n_i} + a_{n_i-1} X^{n_i-1} + \ldots + a_1 X + a_0$ is the minimal polynomial of the matrix R_i.

For these polynomials the following holds: m_{i+1} divides m_i for $i = 1, \ldots, r - 1$, and the product of all m_i for $i = 1, \ldots, r$ gives a factorization of the characteristic polynomial of the matrix A. The Frobenius form defined in this way is unique.

References

Reference: P. Ozello: *Calcul exact des formes de Jordan et de Frobenius d'une matrice*, pp. 30–43. Thèse de l'Universite Scientifique Technologique et Medicale de Grenoble, 1987

See Also

MuPAD Functions

`linalg::hermiteForm` | `linalg::jordanForm` | `linalg::minpoly` | `linalg::smithForm`

linalg::gaussElim

Gaussian elimination

Syntax

```
linalg::gaussElim(A, <All>)
```

Description

`linalg::gaussElim(A)` performs Gaussian elimination on the matrix A to reduce A to a similar matrix in upper row echelon form.

A row echelon form of `A` returned by `linalg::gaussElim` is not unique. See `linalg::gaussJordan` for computing the *reduced* row echelon form.

The component ring R of `A` must be an integral domain, i.e., a domain of category `Cat::IntegralDomain`.

If R is a field, i.e., a domain of category `Cat::Field`, ordinary Gaussian elimination is used. Otherwise, `linalg::gaussElim` applies fraction-free Gaussian elimination to `A`.

`linalg::gaussElim` serves as an interface function for the method `"gaussElim"` of the matrix domain of `A`, i.e., one may call `A::dom::gaussElim(A)` directly instead of `linalg::gaussElim(A, All)`

Refer to the help page of `Dom::Matrix` for details about the computation strategy of `linalg::gaussElim`.

Examples

Example 1

We apply Gaussian elimination to the following matrix:

```
A := Dom::Matrix(Dom::Rational)(
```

```
  [[1, 2, 3, 4], [-1, 0, 1, 0], [3, 5, 6, 9]]
)
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ -1 & 0 & 1 & 0 \\ 3 & 5 & 6 & 9 \end{pmatrix}$$

which reduces A to the following row echelon form:

```
linalg::gaussElim(A)
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 0 & 2 & 4 & 4 \\ 0 & 0 & -1 & -1 \end{pmatrix}$$

Example 2

We apply Gaussian elimination to the matrix:

```
B := Dom::Matrix(Dom::Integer)(
  [[1, 2, -1], [1, 0, 1], [2, -1, 4]]
)
```

$$\begin{pmatrix} 1 & 2 & -1 \\ 1 & 0 & 1 \\ 2 & -1 & 4 \end{pmatrix}$$

and get the following result:

```
linalg::gaussElim(B, All)
```

$$\left[\begin{pmatrix} 1 & 2 & -1 \\ 0 & -2 & 2 \\ 0 & 0 & -2 \end{pmatrix}, 3, -2, \{1, 2, 3\}\right]$$

We see that $rank(B) = 3$ and $\det B = -2$.

Parameters

A

A matrix of a domain of category `Cat::Matrix`

Options

All

Returns a list $[T, \text{rank}(A), \det A, \{j_1, \ldots, j_r\}]$ where T is a row echelon form of A and $\{j_1, \ldots, j_r\}$ is the set of characteristic column indices of T.

If A is not square, then the value `FAIL` is given instead of $\det A$.

Return Values

a matrix of the same domain type as A, or the list `[T, rank(A), det(A), {j_1,dots,j_r}]` when the option `All` is given (see below).

Algorithms

Let $T = (t_{i,j})_{1 \le i \le m,\, 1 \le j \le n}$ be an $m \times n$ matrix. Then T is a matrix in an upper *row echelon form*, if $r \in \{0, 1, \ldots, n\}$ and indices $j_1, j_2, \ldots, j_r \in \{1, \ldots, n\}$ exist with:

1. $j_1 < j_2 < \cdots < j_r$.
2. For each $i \in \{1, \ldots, r\}$: $t_{i,1} = t_{i,2} = \cdots = t_{i,j_i - 1} = 0$.
3. For each $i \in \{r + 1, \ldots, m\}$: $t_{i,j} = 0$ for each $j \in \{1, \ldots, n\}$.

The indices $j_1, j_2, \ldots, j_r$ are the *characteristic column indices* of the matrix T.

See Also

MuPAD Functions

`linalg::gaussJordan` | `lllint`

linalg::gaussJordan

Gauss-Jordan elimination

Syntax

```
linalg::gaussJordan(A, <All>)
```

Description

`linalg::gaussJordan(A)` performs Gauss-Jordan elimination on the matrix A, i.e., it returns the reduced row echelon form of A.

The component ring R of `A` must be an integral domain, i.e., a domain of category `Cat::IntegralDomain`.

If R is a field, i.e., a domain of category `Cat::Field`, then the leading entries of the matrix T in reduced row echelon form are equal to one.

If R is a ring providing the method `"gcd"`, then the components of each row of T do not have a non-trivial common divisor.

If the component ring of `A` is a field, then the reduced row echelon form is unique.

Examples

Example 1

We apply Gauss-Jordan elimination to the following matrix:

```
A := Dom::Matrix(Dom::Rational)(
  [[1, 2, 3, 4], [-5, 0, 3, 0], [3, 5, 6, 9]]
)
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ -5 & 0 & 3 & 0 \\ 3 & 5 & 6 & 9 \end{pmatrix}$$

```
linalg::gaussJordan(A, All)
```

$$\left[\begin{pmatrix} 1 & 0 & 0 & \frac{1}{2} \\ 0 & 1 & 0 & \frac{1}{2} \\ 0 & 0 & 1 & \frac{5}{6} \end{pmatrix}, 3, \text{FAIL}, \{1, 2, 3\}\right]$$

We see that $rank(B) = 3$. Because the determinant of a matrix is only defined for square matrices, the third element of the returned list is the value `FAIL`.

Example 2

If we consider the matrix from "Example 1" on page 15-79 as an integer matrix and apply the Gauss-Jordan elimination we get the following matrix:

```
B := Dom::Matrix(Dom::Integer)(
  [[1, 2, 3, 4], [-5, 0, 3, 0], [3, 5, 6, 9]]
):
linalg::gaussJordan(B)
```

$$\begin{pmatrix} 2 & 0 & 0 & 1 \\ 0 & -2 & 0 & -1 \\ 0 & 0 & -6 & -5 \end{pmatrix}$$

Parameters

A

A matrix of a domain of category `Cat::Matrix`

Options

All

Returns a list $[T, \text{rank}(A), \det A, \{j_1, \ldots, j_r\}]$ where T is the reduced row echelon form of `A` and $\{j_1, \ldots, j_r\}$ is the set of characteristic column indices of `T`.

If A is not square, then the value FAIL is given instead of $\det A$.

Return Values

a matrix of the same domain type as A, or the list `[T, rank(A), det(A), {j_1,dots,j_r}]` when the option `All` is given (see below).

Algorithms

Let $T = (t_{i,j})_{1 \le i \le m,\, 1 \le j \le n}$ be an $m \times n$ matrix. Then T is a matrix in *reduced row echelon form*, if $r \in \{0, 1, \ldots, n\}$ and indices $j_1, j_2, \ldots, j_r \in \{1, \ldots, n\}$ exist with:

1. $j_1 < j_2 < \cdots < j_r$.
2. For each $i \in \{1, \ldots, r\}$: $t_{i,1} = t_{i,2} = \cdots = t_{i,j_i - 1} = 0$. In addition, if A is defined over a field: $t_{i,j_i} = 1$.
3. For each $i \in \{r + 1, \ldots, m\}$: $t_{i,j} = 0$ for each $j \in \{1, \ldots, n\}$.
4. For each $i \in \{1, \ldots, r\}$: $t_{k,j_i} = 0$ for each $k \in \{1, \ldots, i - 1\}$.

The indices $j_1, j_2, \ldots, j_r$ are the *characteristic column indices* of the matrix T.

See Also

MuPAD Functions

`linalg::gaussElim`

linalg::hermiteForm

Hermite normal form of a matrix

Syntax

```
linalg::hermiteForm(A, <All>)
```

Description

`linalg::hermiteForm(A)` computes the Hermite normal form of a non-singular integer square matrix A. This is an upper-triangular matrix H such that $H_{jj} \geq 0$ and $-\frac{H_{jj}}{2} < H_{ij} \leq \frac{H_{jj}}{2}$ for $j > i$. In the case, A is not a square matrix or a singular matrix, the matrix H is simply an upper-triangular matrix.

If the matrix `A` is not of the domain `Dom::Matrix(Dom::Integer)` then `A` is converted into a matrix of this domain for intermediate computations.

If this conversion fails, then an error message is returned.

`linalg::hermiteForm(A, All)` computes a transformation matrix U and a matrix H such that $H = UA$.

Examples

Example 1

We compute the Hermite normal form of the matrix:

```
A := Dom::Matrix(Dom::Rational)(
   [[9, -36, 30], [-36, 192, -180], [30, -180, 180]]
)
```

$$\begin{pmatrix} 9 & -36 & 30 \\ -36 & 192 & -180 \\ 30 & -180 & 180 \end{pmatrix}$$

```
linalg::hermiteForm(A)
```

$$\begin{pmatrix} 3 & 0 & 30 \\ 0 & 12 & 0 \\ 0 & 0 & 60 \end{pmatrix}$$

We may also compute the transformation matrix by giving the option `All`:

```
linalg::hermiteForm(A, All)
```

$$\left[\begin{pmatrix} 3 & 0 & 30 \\ 0 & 12 & 0 \\ 0 & 0 & 60 \end{pmatrix}, \begin{pmatrix} 13 & 9 & 7 \\ 6 & 4 & 3 \\ 20 & 15 & 12 \end{pmatrix}\right]$$

Let us check the result:

```
U := linalg::hermiteForm(A, All)[2]:
```

```
U * A
```

$$\begin{pmatrix} 3 & 0 & 30 \\ 0 & 12 & 0 \\ 0 & 0 & 60 \end{pmatrix}$$

Parameters

`A`

An integer matrix of category `Cat::Matrix`

Options

`All`

Returns the list `[H, U]` with the hermite normal form H of `A` and the corresponding transformation matrix U

Return Values

Either a matrix of the same domain type as `A` or the list `[H, U]` when the option `All` is given.

Algorithms

Let A be an $n \times n$ matrix with coefficients in $\mathbb{Z}$. Then there exists an $n \times n$ matrix $H = (h_{ij})$ in Hermite normal form such that $H = AU$ with $|U| = \pm 1$.

Note that H is unique, if A has full row rank. The matrix U is not unique. U may be computed by using the option `All`.

If A is a square matrix, then the product of the diagonal elements of its Hermite normal form is, up to the sign, the determinant of A.

See Also

MuPAD Functions

`linalg::frobeniusForm` | `linalg::jordanForm` | `linalg::smithForm` | `lllint`

linalg::hessenberg

Hessenberg matrix

Syntax

```
linalg::hessenberg(A, <All>)
```

Description

`linalg::hessenberg(A)` returns an (upper) Hessenberg matrix H.

`linalg::hessenberg` uses Gaussian elimination without pivoting. There is no special implementation for matrices with floating-point components.

The component ring of `A` must be a field, i.e., a domain of category `Cat::Field`.

Examples

Example 1

Consider the matrix:

```
A := Dom::Matrix(Dom::Rational)(
  [[0, 1, 0, -1], [-4/3, 2/3, 5/3, -1/3],
   [-1, 2, 0, 0], [-5/3, 4/3, 1/3, 1/3]]
)
```

$$\begin{pmatrix} 0 & 1 & 0 & -1 \\ -\frac{4}{3} & \frac{2}{3} & \frac{5}{3} & -\frac{1}{3} \\ -1 & 2 & 0 & 0 \\ -\frac{5}{3} & \frac{4}{3} & \frac{1}{3} & \frac{1}{3} \end{pmatrix}$$

The following Hessenberg matrix is similar to A:

```
H := linalg::hessenberg(A)
```

$$\begin{pmatrix} 0 & -\frac{1}{4} & -\frac{1}{7} & -1 \\ -\frac{4}{3} & \frac{3}{2} & \frac{34}{21} & -\frac{1}{3} \\ 0 & \frac{7}{8} & -\frac{17}{14} & \frac{1}{4} \\ 0 & 0 & -\frac{72}{49} & \frac{5}{7} \end{pmatrix}$$

If the corresponding transformation matrix is needed as well, call `linalg::hessenberg` with option `All`:

```
[H, P] := linalg::hessenberg(A, All)
```

$$\left[\begin{pmatrix} 0 & -\frac{1}{4} & -\frac{1}{7} & -1 \\ -\frac{4}{3} & \frac{3}{2} & \frac{34}{21} & -\frac{1}{3} \\ 0 & \frac{7}{8} & -\frac{17}{14} & \frac{1}{4} \\ 0 & 0 & -\frac{72}{49} & \frac{5}{7} \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -\frac{3}{4} & 1 & 0 \\ 0 & -\frac{8}{7} & -\frac{1}{7} & 1 \end{pmatrix}\right]$$

Then P is a nonsingular matrix such that the product PAP^{-1} is equal to H:

```
P * A * P^(-1)
```

$$\begin{pmatrix} 0 & -\frac{1}{4} & -\frac{1}{7} & -1 \\ -\frac{4}{3} & \frac{3}{2} & \frac{34}{21} & -\frac{1}{3} \\ 0 & \frac{7}{8} & -\frac{17}{14} & \frac{1}{4} \\ 0 & 0 & -\frac{72}{49} & \frac{5}{7} \end{pmatrix}$$

Parameters

A

A square matrix of a domain of category `Cat::Matrix`

Options

All

Returns the list `[H, P]` with a Hessenberg matrix H similar to `A` and the corresponding nonsingular transformation matrix P such that $H = PAP^{-1}$.

Return Values

Matrix of the same domain type as `A`, or the list `[H, P]` when the option `All` is given.

Algorithms

An $n \times n$ matrix $A = (a_{i,j})_{1 \le i \le n,\ 1 \le j \le n}$ is called an (upper) *Hessenberg matrix*, if the following holds: $a_{i,j} = 0$ for all $i, j \in \{1, \ldots, n\}$ with $i > j$.

For each square matrix A over a field there exists a Hessenberg matrix similar to A. In general, the upper Hessenberg matrix is not unique.

References

Reference: K.-H. Kiyek, F. Schwarz: *Lineare Algebra*. Teubner Studienbücher Mathematik, B.G. Teubner Stuttgart, Leipzig, 1999.

See Also

MuPAD Functions

`linalg::charpoly`

linalg::hilbert

Hilbert matrix

Syntax

```
linalg::hilbert(n, <R>)
```

Description

`linalg::hilbert(n)` returns the $n \times n$ Hilbert matrix $H = (h_{i,j})_{1 \le i \le m,\ 1 \le j \le n}$ defined by $h_{i,j} = \frac{1}{(i+j-1)}$.

The entries of Hilbert matrices are rational numbers. Note, however, that the returned matrix is not defined over the component domain `Dom::Rational`, but over the standard component domain `Dom::ExpressionField()`. Thus, no conversion is necessary when working with other functions that expect or return matrices over that component domain.

Use `linalg::hilbert(n, Dom::Rational)` to define the $n \times n$ Hilbert matrix over the field of rational numbers.

Examples

Example 1

We construct the 3×3 Hilbert matrix:

```
H := linalg::hilbert(3)
```

$$\begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{pmatrix}$$

This is a matrix of the domain `Dom::Matrix()`.

If you prefer a different component ring, the matrix may be converted to the desired domain after construction (see `coerce`, for example). Alternatively, one can specify the component ring when creating the Hilbert matrix. For example, specification of the domain `Dom::Float` generates floating-point entries:

```
H := linalg::hilbert(3, Dom::Float)
```

$$\begin{pmatrix} 1.0 & 0.5 & 0.3333333333 \\ 0.5 & 0.3333333333 & 0.25 \\ 0.3333333333 & 0.25 & 0.2 \end{pmatrix}$$

```
domtype( H )
```

Dom::Matrix(Dom::Float)

Parameters

`n`

The dimension of the matrix: a positive integer

`R`

The component ring: a domain of category `Cat::Rng`; default: `Dom::ExpressionField()`

Return Values

$n \times n$ matrix of the domain `Dom::Matrix(R)`.

Algorithms

Hilbert matrices are symmetric and positive definite.

Hilbert matrices of large dimension are notoriously ill-conditioned challenging any numerical inversion scheme. However, their inverse can also be computed by a closed formula (see `linalg::invhilbert`).

See Also

MuPAD Functions

`linalg::invhilbert` | `linalg::invpascal` | `linalg::invvandermonde` | `linalg::pascal` | `linalg::toeplitz` | `linalg::toeplitzSolve` | `linalg::vandermonde` | `linalg::vandermondeSolve`

linalg::htranspose

Hermitian transpose of a matrix

Syntax

```
linalg::htranspose(M)
```

Description

`linalg::htranspose(M)` computes the Hermitian transpose of the matrix *M*.

The Hermitian transpose of *M* is computed. The result is an $n \times m$ matrix.

The [i, j]th element of the result is equal to the conjugate of the [j, i]th element of M.

Examples

Example 1

We define a 3×4 matrix:

```
A := matrix([[1, 2, 3, 4], [-I, 0, 1+I, 0], [3, 5, 6, 9]])
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ -i & 0 & 1+i & 0 \\ 3 & 5 & 6 & 9 \end{pmatrix}$$

Then the Hermitian transpose of A is the 4 ×3 matrix:

```
linalg::htranspose(A)
```

$$\begin{pmatrix} 1 & i & 3 \\ 2 & 0 & 5 \\ 3 & 1-i & 6 \\ 4 & 0 & 9 \end{pmatrix}$$

Parameters

M

$m \times n$ matrix of domain `Dom::Matrix`

Return Values

$n \times m$ matrix of domain `Dom::Matrix`.

Overloaded By

M

Algorithms

Let $A = (a_{i,j})_{1 \le i \le m,\ 1 \le j \le n}$ be an $m \times n$ matrix. Then the Hermitian transpose of A is the $n \times m$ matrix:

$$A^t = \begin{pmatrix} \overline{a_{1,1}} & \overline{a_{1,2}} & \cdots & \overline{a_{m,1}} \\ \overline{a_{1,2}} & \overline{a_{2,2}} & \cdots & \overline{a_{m,2}} \\ \vdots & \vdots & \cdots & \vdots \\ \overline{a_{1,n}} & \overline{a_{1,n}} & \cdots & \overline{a_{m,n}} \end{pmatrix}.$$

See Also

MuPAD Functions

`conjugate` | `linalg::transpose`

linalg::intBasis

Basis for the intersection of vector spaces

Syntax

`linalg::intBasis(`S_1`, `S_2`, …)`

Description

`linalg::intBasis( `S_1`, `S_2`, ...)` returns a basis for the intersection of the vector spaces spanned by the vectors in S_1, S_2, ….

The domain type of the vectors of the returned set is the domain type of the first parameter S_1.

A basis for the zero-dimensional space is the empty set or empty list, respectively.

The given vectors must be defined over the same component ring which must be a field, i.e., a domain of category `Cat::Field`.

Examples

Example 1

We define three vectors $\vec{v}_1, \vec{v}_2, \vec{v}_3$ in $\mathbb{Q}^2$:

```
MatQ := Dom::Matrix(Dom::Rational):
v1 := MatQ([[3, -2]]); v2 := MatQ([[1, 0]]); v3 := MatQ([[5, -3]])
```

$$\begin{pmatrix} 3 & -2 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 5 & -3 \end{pmatrix}$$

A basis for the vector space $V_1 \cap V_2 \cap V_3$ with

- V_1 generated by $\vec{v}_1, \vec{v}_2, \vec{v}_3$
- V_2 generated by $\vec{v}_1, \vec{v}_3$
- V_3 generated by $\vec{v}_1 + \vec{v}_2, \vec{v}_2, \vec{v}_1 + \vec{v}_3$

is:

```
linalg::intBasis([v1, v2, v3], [v1, v3], [v1 + v2, v2, v1 + v3])
```

$$[(4\ \ -2),(1\ \ 0)]$$

Example 2

The intersection of the two vector spaces spanned by the vectors in S_1 and S_2, respectively:

```
S1 := {matrix([[1, 0, 1, 0]]), matrix([[0, 1, 0, 1]])};
S2 := {matrix([[1, 2, 1, 1]]), matrix([[-1, -2, 1, 0]])}
```

$$\{(0\ 1\ 0\ 1),(1\ 0\ 1\ 0)\}$$

$$\{(-1\ -2\ 1\ 0),(1\ 2\ 1\ 1)\}$$

is the zero-dimensional space:

```
linalg::intBasis(S1, S2)
```

$$\varnothing$$

Parameters

`S1, S2, ...`

Either sets or lists of n-dimensional vectors (a vector is an $n \times 1$ or $1 \times n$ matrix of a domain of category `Cat::Matrix`)

Return Values

Set or a list of vectors, according to the domain type of the parameter S_1.

See Also

MuPAD Functions

`linalg::basis` | `linalg::sumBasis`

linalg::inverseLU

Computing the inverse of a matrix using LU-decomposition

Syntax

```
linalg::inverseLU(A)
```

```
linalg::inverseLU(L, U, pivindex)
```

Description

`linalg::inverseLU(A)` computes the inverse $\frac{1}{A}$ of the square matrix A using LU-decomposition.

`linalg::inverseLU(L, U, pivindex)` computes the inverse of the matrix $A = P^{-1} L U$ where `L`, `U` and `pivindex` are the result of an LU-deomposition of the (nonsingular) Matrix A, as computed by `linalg::factorLU`.

The matrix `A` must be nonsingular.

`pivindex` is a list `[r[1], r[2], ...]` representing a permutation matrix P such that $B = PA = LU$, where $b_{ij} = a_{r_i, j}$.

It is not checked whether `pivindex` has such a form.

The component ring of the input matrices must be a field, i.e., a domain of category `Cat::Field`.

Examples

Example 1

We compute the inverse of the matrix:

```
A := Dom::Matrix(Dom::Real)(
```

```
  [[2, -3, -1], [1, 1, -1], [0, 1, -1]]
)
```

$$\begin{pmatrix} 2 & -3 & -1 \\ 1 & 1 & -1 \\ 0 & 1 & -1 \end{pmatrix}$$

using LU-decomposition:

```
Ai := linalg::inverseLU(A)
```

$$\begin{pmatrix} 0 & 1 & -1 \\ -\frac{1}{4} & \frac{1}{2} & -\frac{1}{4} \\ -\frac{1}{4} & \frac{1}{2} & -\frac{5}{4} \end{pmatrix}$$

We check the result:

```
A * Ai, Ai * A
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

We can also compute the inverse of A in the usual way:

```
1/A
```

$$\begin{pmatrix} 0 & 1 & -1 \\ -\frac{1}{4} & \frac{1}{2} & -\frac{1}{4} \\ -\frac{1}{4} & \frac{1}{2} & -\frac{5}{4} \end{pmatrix}$$

`linalg::inverseLU` should be used for efficiency reasons in the case where an LU decomposition of a matrix already is computed, as the next example illustrates.

Example 2

If we already have an LU decomposition of a (nonsingular) matrix, we can compute the inverse of the matrix $A = P^{1} L U$ as follows:

```
LU := linalg::factorLU(linalg::hilbert(3))
```

$$\left[\begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{1}{3} & 1 & 1 \end{pmatrix}, \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ 0 & \frac{1}{12} & \frac{1}{12} \\ 0 & 0 & \frac{1}{180} \end{pmatrix}, [1, 2, 3]\right]$$

```
linalg::inverseLU(op(LU))
```

$$\begin{pmatrix} 9 & -36 & 30 \\ -36 & 192 & -180 \\ 30 & -180 & 180 \end{pmatrix}$$

`linalg::inverseLU` then only needs to perform forward and backward substitution to compute the inverse matrix (see also `linalg::matlinsolveLU`).

Parameters

`A, L, U`

A square matrix of a domain of category `Cat::Matrix`

`pivindex`

A list of positive integers

Return Values

Matrix of the same domain type as `A` or `L`, respectively.

See Also

MuPAD Functions

`_invert` | `linalg::factorLU` | `linalg::matlinsolveLU`

linalg::invhilbert

Inverse of a Hilbert matrix

Syntax

```
linalg::invhilbert(n, <R>)
```

Description

`linalg::invhilbert(n)` returns the inverse of the $n \times n$ Hilbert matrix H. The $n \times n$ Hilbert matrix $H = (h_{i,j})_{1 \le i \le m,\, 1 \le j \le n}$ is defined by $h_{i,j} = \frac{1}{(i+j-1)}$.

`linalg::invhilbert` uses an explicit formula for the inverse.

Note that the entries of the inverse of a Hilbert matrix are integers. But the returned matrix is defined over the standard component domain `Dom::ExpressionField()` so that no conversion is necessary when working with other functions that expect or return matrices over that component domain.

`linalg::invhilbert(n,Dom::Integer)` returns the inverse of the $n \times n$ Hilbert matrix defined over the integers.

Examples

Example 1

We compute the inverse of the 3×3 Hilbert matrix:

```
A := linalg::invhilbert(3)
```

$$\begin{pmatrix} 9 & -36 & 30 \\ -36 & 192 & -180 \\ 30 & -180 & 180 \end{pmatrix}$$

This is a matrix of the domain `Dom::Matrix()`.

If you prefer a different component ring, the matrix may be converted into the desired domain afterwards (see `coerce`, for example). Alternatively, one can specify the component ring when calling `linalg::invhilbert`, for example the domain `Dom::Float`:

```
A := linalg::invhilbert(3, Dom::Float)
```

$$\begin{pmatrix} 9.0 & -36.0 & 30.0 \\ -36.0 & 192.0 & -180.0 \\ 30.0 & -180.0 & 180.0 \end{pmatrix}$$

```
domtype( A )
```

Dom::Matrix(Dom::Float)

Parameters

n

The dimension of the matrix: a positive integer

R

The component ring: a domain of category `Cat::Rng`; default: `Dom::ExpressionField()`

Return Values

$n \times n$ matrix of the domain `Dom::Matrix(R)`.

Algorithms

Hilbert matrices of large dimension are notoriously ill-conditioned, challenging any numerical inversion scheme.

`linalg::invhilbert` uses the formula

$$\left(\frac{1}{H}\right)_{i,j} = \frac{(-1)^{i+j}\left(c_i\, c_j\right)}{i+j-1}$$

where

$$c_i = \frac{(n+i-1)!}{(n-i)!\,((i-1)!)^2}$$

for the inverse of the $n \times n$ Hilbert matrix H. All entries of $\frac{1}{H}$ are integers.

References

N.J. Higham, Accuracy and Stability of Numerical Algorithms, SIAM 1996

See Also

MuPAD Functions

`linalg::hilbert` | `linalg::invpascal` | `linalg::invvandermonde` | `linalg::pascal` | `linalg::toeplitz` | `linalg::toeplitzSolve` | `linalg::vandermonde` | `linalg::vandermondeSolve`

linalg::invpascal

Inverse of a Pascal matrix

Syntax

```
linalg::invpascal(n, <R>)
```

Description

`linalg::invpascal(n)` returns the inverse of the $n \times n$ Pascal matrix.

The entries of inverse Pascal matrices are integer numbers. Note, however, that the returned matrix is not defined over the component domain `Dom::Integer`, but over the standard component domain `Dom::ExpressionField()`. Thus, no conversion is necessary when working with other functions that expect or return matrices over that component domain.

The runtime to compute the inverse $n \times n$ Pascal matrix via `linalg::invpascal` is $O(n^2)$. This is much faster than inverting the Pascal matrix by a generic inversion algorithm.

The Pascal matrices are provided by `linalg::pascal`.

Examples

Example 1

We construct the inverse 3×3 Pascal matrix:

```
linalg::invpascal(3)
```

$$\begin{pmatrix} 3 & -3 & 1 \\ -3 & 5 & -2 \\ 1 & -2 & 1 \end{pmatrix}$$

This is a matrix of the domain `Dom::Matrix()`.

If you prefer a different component ring, the matrix may be converted to the desired domain after construction (see `coerce`, for example). Alternatively, one can specify the component ring when creating the inverse Pascal matrix. For example, specification of the domain `Dom::Float` generates floating-point entries:

```
linalg::invpascal(3, Dom::Float)
```

$$\begin{pmatrix} 3.0 & -3.0 & 1.0 \\ -3.0 & 5.0 & -2.0 \\ 1.0 & -2.0 & 1.0 \end{pmatrix}$$

```
domtype(%)
```

Dom::Matrix(Dom::Float)

Parameters

`n`

The dimension of the matrix: a positive integer

`R`

The component ring: a domain of category `Cat::Rng`; default: `Dom::ExpressionField()`

Return Values

$n \times n$ matrix of the domain `Dom::Matrix(R)`.

Algorithms

Pascal matrices and their inverses are symmetric and positive definite.

The determinant of a Pascal matrix and its inverse is 1.

The inverse of a Pascal matrix has integer entries.

If λ is an eigenvalue of a Pascal matrix/inverse Pascal matrix, then $\frac{1}{\lambda}$ is also an eigenvalue of the matrix.

The entries Q_{ij} of the inverse $n \times n$ Pascal matrix Q satisfy the linear relation

$$Q_{i,j} = Q_{i,j+1} + Q_{i+1,j} + (-1)^{i+j} \binom{n}{i} \binom{n}{j}.$$

This relation is used by `linalg::invpascal` to compute the matrix.

See Also

MuPAD Functions

`linalg::hilbert` | `linalg::invhilbert` | `linalg::invvandermonde` | `linalg::pascal` | `linalg::toeplitz` | `linalg::toeplitzSolve` | `linalg::vandermonde` | `linalg::vandermondeSolve`

linalg::isHermitian

Checks whether a matrix is Hermitian

Syntax

```
linalg::isHermitian(A)
```

Description

`linalg::isHermitian(A)` determines whether the matrix A is Hermitian, i.e., whether $A = \overline{A}^t$, where $\overline{A}$ denotes the conjugate matrix.

If the component ring of the matrix `A` does not provide the method `"conjugate"`, then `A` is tested for symmetry, i.e., `linalg::isHermitian` returns `TRUE` if and only if `A` satisfies the equation $A = A^t$.

Examples

Example 1

Here is an example of a Hermitian matrix:

```
A := Dom::Matrix(Dom::Complex)([[1, I], [-I, 1]])
```

$$\begin{pmatrix} 1 & i \\ -i & 1 \end{pmatrix}$$

```
linalg::isHermitian(A)
```

TRUE

The following matrix is not Hermitian:

```
B := Dom::Matrix(Dom::Complex)([[1, -I], [-I, 1]])
```

$$\begin{pmatrix} 1 & -i \\ -i & 1 \end{pmatrix}$$

```
linalg::isHermitian(B)
```

FALSE

The reason is the following:

```
linalg::transpose(conjugate(B)) <> B
```

$$\begin{pmatrix} 1 & i \\ i & 1 \end{pmatrix} \neq \begin{pmatrix} 1 & -i \\ -i & 1 \end{pmatrix}$$

Example 2

Here is an example of a symmetric matrix over the integers:

```
C := Dom::Matrix(Dom::Integer)([[1, 2], [2, -1]])
```

$$\begin{pmatrix} 1 & 2 \\ 2 & -1 \end{pmatrix}$$

```
linalg::isHermitian(C)
```

TRUE

Parameters

A

A square matrix of a domain of category `Cat::Matrix`

Return Values

Either TRUE or FALSE.

See Also

MuPAD Functions
`linalg::isPosDef`

linalg::isPosDef

Test a matrix for positive definiteness

Syntax

```
linalg::isPosDef(A)
```

Description

`linalg::isPosDef(A)` checks whether the matrix A is positive definite, so that $\vec{x}^t A \vec{x} > 0$ for arbitrary vectors $\vec{x} \neq \vec{0}$.

The component ring of `A` must be a field, i.e., a domain of category `Cat::Field`.

An error message is returned, if a result of an intermediate computation cannot be checked for being positive (which could happen, for example, if components of `A` are symbolic).

Environment Interactions

Properties of identifiers are taken into account.

Examples

Example 1

Here is an example of a positive definite matrix:

```
MatR := Dom::Matrix( Dom::Real ):
A := MatR([[14, 6, 9], [6, 17, -4], [9, -4, 13]])
```

$$\begin{pmatrix} 14 & 6 & 9 \\ 6 & 17 & -4 \\ 9 & -4 & 13 \end{pmatrix}$$

```
linalg::isPosDef(A)
```

TRUE

The following matrix is not positive definite:

```
B := MatR([[1, 2, 3], [2, 3, 4], [5, 6, 7]])
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 5 & 6 & 7 \end{pmatrix}$$

```
linalg::isPosDef(B)
```

FALSE

Example 2

`linalg::isPosDef` in general does not work for matrices with symbolic entries. It may respond with an error message (because the system in general cannot decide whether a symbolic component is positive), such as for the following matrix:

```
delete a, b:
C := matrix([[a, b], [b, a]])
```

$$\begin{pmatrix} a & b \\ b & a \end{pmatrix}$$

```
linalg::isPosDef(C)
```

```
Error: Cannot check whether the matrix component is positive. [linalg::factorCholesky]
```

However, properties of identifiers are taken into account, so that, for example, `linalg::isPosDef` is able to perform the test correctly for the following matrix:

```
assume(a > 1): C := matrix([[a, 1], [1, a]]):
linalg::isPosDef(C)
```

```
TRUE
```

Note that such computations depend on the power of the underlying property mechanism implemented in the `property` library.

Parameters

A

A matrix of a domain of category `Cat::Matrix`

Return Values

Either TRUE or FALSE.

See Also

MuPAD Functions

`linalg::factorCholesky` | `linalg::isHermitian`

linalg::isUnitary

Test whether a matrix is unitary

Syntax

```
linalg::isUnitary(A)
```

Description

`linalg::isUnitary` tests whether the matrix A is a unitary matrix. An $n \times n$ matrix A is unitary, if $A\,\overline{A}^t = I_n$, where I_n is the $n \times n$ identity matrix.

The square matrix `A` is a unitary matrix, if and only if the columns of `A` form an orthonormal basis with respect to the scalar product `linalg::scalarProduct` of two vectors.

The correctness of the result `FALSE` of `linalg::isUnitary` can only be guaranteed if the elements of the component ring R of the matrix `A` are canonically represented, i.e., if each element of R has only one unique representation.

The axiom `Ax::canonicalRep` states that a domain has this property. Hence, `linalg::isUnitary` returns `FALSE` or `UNKNOWN`, respectively, depending on whether the component ring of `A` has the axiom `Ax::canonicalRep`.

If the component ring of `A` does not define the method `"conjugate"` then it is checked whether `A` is an orthogonal matrix such that $A\,A^t = E_n$, where E_n is the $n \times n$ identity matrix.

Examples

Example 1

The following matrix is unitary:

```
A := 1/sqrt(5) * matrix([[1, 2], [2, -1]])
```

$$\begin{pmatrix} \frac{\sqrt{5}}{5} & \frac{2\sqrt{5}}{5} \\ \frac{2\sqrt{5}}{5} & -\frac{\sqrt{5}}{5} \end{pmatrix}$$

```
linalg::isUnitary(A)
```

TRUE

Parameters

A

A square matrix of a domain of category `Cat::Matrix`

Return Values

Either `TRUE`, `FALSE`, or `UNKNOWN`.

See Also

MuPAD Functions

`linalg::orthog` | `linalg::scalarProduct`

linalg::jordanForm

Jordan normal form of a matrix

Syntax

```
linalg::jordanForm(A, <All>)
```

Description

`linalg::jordanForm(A)` returns the Jordan normal form J of the matrix A.

`linalg::jordanForm` computes a nonsingular transformation matrix P and a matrix J such that $A = P\,J\,P^{-1}$ with $J = diag(J_1, \ldots, J_r)$ and Jordan matrices $J_1, \ldots, J_r$.

The Jordan normal form of a square matrix A over a field F exists if the characteristic polynomial of A splits over F into linear factors. If this is not the case for the matrix `A`, then `linalg::jordanForm` returns `FAIL`.

The Jordan normal form is unique up to permutations of the Jordan matrices $J_1, \ldots, J_r$.

The implemented method computes the eigenvalues of `A`. It returns `FAIL` if this is not possible (see `linalg::eigenvalues`).

The component ring of `A` must be a field, i.e., a domain of category `Cat::Field`.

Examples

Example 1

The Jordan normal form of the matrix:

```
A := Dom::Matrix(Dom::Complex)([[1, 2], [4, 5]])
```

$$\begin{pmatrix} 1 & 2 \\ 4 & 5 \end{pmatrix}$$

is the following matrix:

```
J := linalg::jordanForm(A)
```

$$\begin{pmatrix} -2\sqrt{3}+3 & 0 \\ 0 & 2\sqrt{3}+3 \end{pmatrix}$$

The corresponding transformation matrix P can be obtained from the result `[J, P]` of `linalg::jordanForm` with the option `All`:

```
P := linalg::jordanForm(A, All)[2]
```

$$\begin{pmatrix} -\frac{\sqrt{3}}{2}-\frac{1}{2} & \frac{\sqrt{3}}{2}-\frac{1}{2} \\ 1 & 1 \end{pmatrix}$$

We check the result:

```
map(P * J * P^(-1), radsimp)
```

$$\begin{pmatrix} 1 & 2 \\ 4 & 5 \end{pmatrix}$$

To get this result we must apply the function `radsimp` to each component of the matrix that is returned by the matrix product PJP^{-1}.

Parameters

`A`

A square matrix of a domain of category `Cat::Matrix`

Options

`All`

Returns the list `[J, P]` with the Jordan normal form J of `A` and the corresponding transformation matrix P such that $A = PJP^{-1}$.

Return Values

Either a matrix of the same domain type as A, the list `[J, P]` when the option `All` is given, or the value `FAIL`.

See Also

MuPAD Functions

`linalg::eigenvalues` | `linalg::frobeniusForm` | `linalg::hermiteForm` | `linalg::smithForm`

linalg::kroneckerProduct

Kronecker product of matrices

Syntax

```
linalg::kroneckerProduct(A, <B, …>)
```

Description

`linalg::kroneckerProduct(A, B)` computes the Kronecker product of two matrices A and B.

The Kronecker product (direct matrix product) of an $m \times n$ matrix A and a $p \times q$ matrix B is the $(m\,p) \times (n\,q)$ matrix $A \otimes B$ given in block form as

$$A \otimes B = \begin{pmatrix} a_{1,1}\,B & \dots & a_{1,n}\,B \\ \vdots & & \vdots \\ a_{m,1}\,B & \dots & a_{m,n}\,B \end{pmatrix}.$$

Componentwise: $A \otimes B_{I,J} = A_{i,j}\,B_{k,l}$ with $I = p\,(i - 1) + k$, $J = q\,(j - 1) + l$.

If A and B are matrices of the same matrix domain with the same component ring, the result is a matrix of the same type as A and B. If the domains or the component rings of A and B differ, `linalg::kroneckerProduct` tries to convert B into the domain type of A by `A::dom::coerce`. If this fails, conversion of A to the domain type of B is attempted. If no conversion is possible, an error is raised.

Note: Note that the Kronecker product is only implemented for matrices over the domains `Dom::Matrix`, `Dom::SquareMatrix` or `Dom::MatrixGroup`. In particular, this includes matrices created by `matrix`.

A call with more than two arguments produces `linalg::kroneckerProduct(A, B, C) = linalg::kroneckerProduct(linalg::kroneckerProduct(A, B), C)` etc.

A call with only one argument is possible. It returns the input matrix.

Examples

Example 1

We consider two matrices A and B with symbolic components:

```
A:= matrix([[a11, a12], [a21, a22]]);
B:= matrix([[b11, b12, b13], [b21, b22, b23]]);
```

$$\begin{pmatrix} a11 & a12 \\ a21 & a22 \end{pmatrix}$$

$$\begin{pmatrix} b11 & b12 & b13 \\ b21 & b22 & b23 \end{pmatrix}$$

The Kronecker product of A and B is computed by multiplying the matrix B with each of the components of the matrix A. The resulting block matrix is returned as a matrix of larger dimension:

```
linalg::kroneckerProduct(A, B);
```

$$\begin{pmatrix} a11\,b11 & a11\,b12 & a11\,b13 & a12\,b11 & a12\,b12 & a12\,b13 \\ a11\,b21 & a11\,b22 & a11\,b23 & a12\,b21 & a12\,b22 & a12\,b23 \\ a21\,b11 & a21\,b12 & a21\,b13 & a22\,b11 & a22\,b12 & a22\,b13 \\ a21\,b21 & a21\,b22 & a21\,b23 & a22\,b21 & a22\,b22 & a22\,b23 \end{pmatrix}$$

```
delete A, B:
```

Example 2

An $n \times n$ matrix H with components in {- 1, 1} is called a *Hadamard matrix* if H multiplied with its transpose equals n times the $n \times n$ identity matrix. The matrix H defined below is a Hadamard matrix:

```
H:= matrix([[1, 1], [1, -1]]);
H * linalg::transpose(H) = 2 * matrix::identity(2);
```

$$\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

$$\begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$$

Hadamard matrices play a role in the field of error correcting codes. A basic property of this type of matrices is that the Kronecker product of two Hadamard matrices is again a Hadamard matrix. We verify this statement for the matrix `H`:

```
H2:= linalg::kroneckerProduct(H, H);
```

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix}$$

Indeed, the matrix `H2` is a again a Hadamard matrix:

```
H2 * linalg::transpose(H2) = 4 * matrix::identity(4);
```

$$\begin{pmatrix} 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix} = \begin{pmatrix} 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}$$

```
delete H, H2:
```

Parameters

`A, B, …`

Matrices of the domains `Dom::Matrix`, `Dom::SquareMatrix` or `Dom::MatrixGroup`

Return Values

Matrix of the same type as A or B.

linalg::matdim

Dimension of a matrix

Syntax

```
linalg::matdim(A)
```

Description

`linalg::matdim(A)` returns the dimension of the matrix *A*, i.e., the number of rows and columns of *A*.

`linalg::matdim` is an interface function for the method `"matdim"` of the matrix domain of `A`, i.e., instead of `linalg::matdim(A)` one may call `A::dom::matdim(A)` directly.

Examples

Example 1

The dimension of the matrix:

```
A := matrix([[1, 2, 3, 4], [3, 1, 4], [5, 6]])
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 4 & 0 \\ 5 & 6 & 0 & 0 \end{pmatrix}$$

can be determined by:

```
linalg::matdim(A)
```

$$[3, 4]$$

Parameters

A

An $m \times n$ matrix of a domain of category `Cat::Matrix`

Return Values

List `[m, n]`, where `m` is the number of rows and `n` is the number of columns of `A`.

See Also

MuPAD Functions

`linalg::ncols` | `linalg::nrows` | `linalg::vecdim`

linalg::matlinsolve

Solving systems of linear equations

Syntax

```
linalg::matlinsolve(A, b, <list>, options)
linalg::matlinsolve(A, B, options)
linalg::matlinsolve(A, options)
```

Description

`linalg::matlinsolve(A, b)` computes the general solution of the equation $A\,\vec{x} = \vec{b}$.

`linalg::matlinsolve(A, b)` returns the solution vector $\vec{x}$ of the system $A\,\vec{x} = \vec{b}$ if it is a unique solution.

`linalg::matlinsolve(A, b)` returns a list $\left[\vec{w}, \left[\vec{v}_1, \ldots, \vec{v}_r\right]\right]$ if the system $A\,\vec{x} = \vec{b}$ has more than one solution, where $\vec{w}$ is one particular solution, i.e., $A\,\vec{w} = \vec{b}$ and $\vec{v}_1, \ldots, \vec{v}_r$ form a basis of the kernel of A, i.e., the solution space of the homogenous system $A\,\vec{x} = (\vec{0})$.

Each solution $\vec{x}$ has the form $\vec{x}_s + s_1\,\vec{v}_1 + \ldots + s_r\,\vec{v}_r (r \le n)$ with certain scalars $s_1, \ldots, s_r$.

A list of n scalars $[s_1, \ldots, s_n]$ may be passed as the additional parameter `list`. This extracts the solution $\vec{x}_s + s_{i_1}\,\vec{v}_1 + \ldots + s_{i_r}\,\vec{v}_r$ with $\{i_1, \ldots, i_r\} = \{1, \ldots, n\} \setminus \{j_1, \ldots, j_l\}$ from the solution space of the system $A\,\vec{x} = \vec{b}$, where $j_1, \ldots, j_l$ are the characteristic column indices of A (see `linalg::gaussJordan`).

The entries of `list` are converted to elements of the component ring of A (an error message is returned if this is not possible).

Note: This option should only be used for exact and symbolic computations. In the case that A or b contains floating-point entries, it should not be used.

If the system $A\vec{x} = \vec{b}$ has no solution, then the empty list `[]` is returned.

`linalg::matlinsolve(A)` solves the matrix equation $C\vec{x} = \vec{b}$, where $\vec{b}$ is the last column of A and C is A with the last column deleted.

`linalg::matlinsolve(A, B)` returns the solution X of the matrix equation $AX = B$, if it has exactly one solution. Otherwise the empty list `[]` is returned.

The vector b and the matrix B respectively, are converted into the domain `Dom::Matrix(R)`, where R is the component ring of A. Solution vectors also belong to this domain.

The component ring of A must be an integral domain, i.e., a domain of category `Cat::IntegralDomain`.

`linalg::matlinsolve` can compute the general solution for systems with more than one solution only over fields, i.e., component rings of category `Cat::Field`. If in this case the component ring of A does not have a canonical representation of the zero element, then it may happen that `linalg::matlinsolve` does not find a basis for the null space. In such a case, a wrong result is returned.

`linalg::matlinsolve` does exploit a sparse structure of A. (A matrix is *sparse* if it has many zero components). See “Example 5” on page 15-130.

Use the function `numeric::matlinsolve` to solve a linear system numerically.

Examples

Example 1

Solve the linear system:

$$\begin{pmatrix} 1 & 2 \\ -1 & 2 \end{pmatrix} \vec{x} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

over the reals. First, enter the coefficient matrix and the right side:

```
MatR := Dom::Matrix(Dom::Real):
A := MatR([[1, 2], [-1, 2]]); b := MatR([1, -1])
```

$$\begin{pmatrix} 1 & 2 \\ -1 & 2 \end{pmatrix}$$

$$\begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

Next, call `linalg::matlinsolve` to solve the system:

```
x := linalg::matlinsolve(A, b)
```

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

The system has exactly one solution. The vector `x` satisfies the matrix equation given above:

```
A * x
```

$$\begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

Example 2

The system:

$$\begin{pmatrix} 1 & 2 \\ -1 & -2 \end{pmatrix} \vec{x} = \begin{pmatrix} 1 \end{pmatrix}$$

does not have a solution over $\mathbb{R}$ (in fact, over no component domain):

```
MatR := Dom::Matrix(Dom::Real):
A := MatR([[1, 2], [-1, -2]]): b := MatR([1, 0]):
```

```
linalg::matlinsolve(A, b)
```

```
[]
```

Example 3

Solve the linear system:

$$\begin{pmatrix} 1 & 1 & -4 & -7 & -6 \\ & 1 & -3 & -5 & -7 \end{pmatrix} \vec{x} = \begin{pmatrix} 30 \\ 17 \end{pmatrix}$$

over the rational numbers. First, enter the coefficient matrix and the right side:

```
MatQ := Dom::Matrix(Dom::Rational):
A := MatQ([[1, 1, -4, -7, -6], [0, 1, -3, -5, -7]]);
b := MatQ([30, 17])
```

$$\begin{pmatrix} 1 & 1 & -4 & -7 & -6 \\ 0 & 1 & -3 & -5 & -7 \end{pmatrix}$$

$$\begin{pmatrix} 30 \\ 17 \end{pmatrix}$$

Next, call `linalg::matlinsolve` to solve the system:

```
sol:= linalg::matlinsolve(A, b)
```

$$\left[\begin{pmatrix} 13 \\ 17 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \left[\begin{pmatrix} 1 \\ 3 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 2 \\ 5 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ 7 \\ 0 \\ 0 \\ 1 \end{pmatrix} \right] \right]$$

The result is to be interpreted as follows: The first vector of the list `sol` is a particular solution of the linear system:

```
A * sol[1]
```

$$\begin{pmatrix} 30 \\ 17 \end{pmatrix}$$

The second entry of the list contains a basis for the null space of A, i.e., the solution space of the corresponding homogenous system $A\,\vec{x} = (\vec{0})$ (the kernel of A). The basis returned is given as a list of vectors.

The following input checks this fact by computing the product $A\,\vec{x}$ for each vector $\vec{x}$ of the list `sol[2]`:

```
map(sol[2], x -> A * x)
```

$$\left[\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \end{pmatrix}\right]$$

Any solution of the linear system can be represented as a sum of a particular solution (here: `sol[1]`) and a linear combination of the basis vectors of the kernel of A. Hence the input system has an infinite number of solutions.

For example, another solution of the system is given by:

```
x := sol[1] + 1*sol[2][1] + 1/2*sol[2][2] - 2*sol[2][3]
```

$$\begin{pmatrix} 17 \\ \frac{17}{2} \\ 1 \\ \frac{1}{2} \\ -2 \end{pmatrix}$$

```
A * x
```

$$\begin{pmatrix} 30 \\ 17 \end{pmatrix}$$

If you identify the columns of the coefficient matrix A of the linear system with the variables x_1, x_2, x_3, x_4, x_5, then you see from the general solution that the variables x_3,

x_4, x_5 act as free parameters. They can be assigned arbitrary rational values to obtain a unique solution.

By giving a list of values for these variables as a third parameter to `linalg::matlinsolve`, you can select a certain vector from the set of all solutions of the linear system. For example, to select the same vector `x` as chosen in the previous input, enter:

```
linalg::matlinsolve(A, b, [0, 0, 1, 1/2, -2])
```

$$\begin{pmatrix} 17 \\ \frac{17}{2} \\ 1 \\ \frac{1}{2} \\ -2 \end{pmatrix}$$

If you are only interested in a particular solution and do not need the general solution of the linear system, enter:

```
linalg::matlinsolve(A, b, Special)
```

$$\begin{pmatrix} 13 \\ 17 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

This call suppresses the computation of the kernel of A.

Example 4

If the linear system is given in the form of equations the function `linalg::expr2Matrix` can be used to form the corresponding matrix equation:

```
delete x, y, z:
Ab := linalg::expr2Matrix(
  [x + y + z = 6, 2*x + y + 2*z = 10, x + 3*y + z = 10]
```

```
)
```

$$\begin{pmatrix} 1 & 1 & 1 & 6 \\ 2 & 1 & 2 & 10 \\ 1 & 3 & 1 & 10 \end{pmatrix}$$

The result here is the extended coefficient matrix of the input system, that is, the right side vector $\vec{b}$ is the 4th column vector of the matrix `Ab`. Since you did not specify a component ring for this matrix, the standard component ring for matrices, the domain `Dom::ExpressionField()`, was chosen.

To solve the linear system, call:

```
linalg::matlinsolve(Ab)
```

$$\left[\begin{pmatrix} 4 \\ 2 \\ 0 \end{pmatrix}, \left[\begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}\right]\right]$$

The system has an infinite number of solutions. The third variable z acts as a free parameter and therefore can have any (complex) value.

To get the general solution in parameter form, you can use parameters for the variables x, y, z of the input system:

```
delete u, v, w:
sol := linalg::matlinsolve(Ab, [u, v, w])
```

$$\begin{pmatrix} 4-w \\ 2 \\ w \end{pmatrix}$$

This is possible here because you perform the matrix computations over `Dom::ExpressionField()` which lets you compute with symbolical (arithmetical) expressions.

To select a certain vector from the set of solutions, for example, the solution for $w = 1$, enter:

```
x := subs(sol, w = 1)
```

$$\begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix}$$

Example 5

Consider a system of linear equations with a sparse structure, that is, the coefficient matrix has many zero components:

```
eqs := {x1 + x5 = 0, x2 - x4 = 1, x3 + 2*x5 = 2, x4 - x5 = -1}:
Ab := linalg::expr2Matrix(eqs, [x1, x2, x3, x4, x5])
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 2 & 2 \\ 0 & 0 & 0 & 1 & -1 & -1 \end{pmatrix}$$

`linalg::matlinsolve` exploits the sparsity of the coefficient matrix if it is passed as a matrix of type `Dom::Matrix`. Alternatively, you can use the function `linsolve` which allows sparse input and output via symbolic equations:

```
linsolve(eqs)
```

$$[x1 = -x5,\ x2 = x5,\ x3 = 2 - 2\,x5,\ x4 = x5 - 1]$$

You also can use the function `numeric::matlinsolve` with the option `Symbolic` instead of `linalg::matlinsolve`:

```
A := linalg::delCol(Ab, 6):
b := linalg::col(Ab, 6):
numeric::matlinsolve(A, b, Symbolic)
```

$$\left[\begin{pmatrix} 0 \\ 0 \\ 2 \\ -1 \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ 1 \\ -2 \\ 1 \\ 1 \end{pmatrix}\right]$$

Note that the function `numeric::matlinsolve` always works over a subfield of the complex numbers and does not let you specify the domain of computation. Without the option `Symbolic`, `numeric::matlinsolve` converts input data to floating-point numbers.

Example 6

Check whether the matrix equation

$$\begin{pmatrix} 1 & 2 \\ -2 & 3 \end{pmatrix} \vec{x} = \begin{pmatrix} 4 & 2 \\ 6 & 3 \end{pmatrix}$$

has a unique solution over the integers.

Start by entering the coefficient matrix and the right side matrix:

```
MatZ := Dom::Matrix(Dom::Integer):
A := MatZ([[1, 2], [-2, 3]]); B := MatZ([[4, 2], [6, 3]])
```

$$\begin{pmatrix} 1 & 2 \\ -2 & 3 \end{pmatrix}$$

$$\begin{pmatrix} 4 & 2 \\ 6 & 3 \end{pmatrix}$$

Next, solve the matrix equation:

```
X := linalg::matlinsolve(A, B)
```

$$\begin{pmatrix} 0 & 0 \\ 2 & 1 \end{pmatrix}$$

The equation indeed has a unique solution (otherwise the answer of `linalg::matlinsolve` would be the empty list `[]`). Check the result:

```
A * X
```

$$\begin{pmatrix} 4 & 2 \\ 6 & 3 \end{pmatrix}$$

Example 7

If you use the `Normal` option, `linalg::matlinsolve` calls the `normal` function for final results. This call ensures that `linalg::matlinsolve` returns results in normalized form:

```
A := matrix([[1, s], [t, -1]]):
b := matrix([s + 1, t - 1]):
x := linalg::matlinsolve(A, b)
```

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

If you specify `Normal = FALSE`, `linalg::matlinsolve` does not call `normal` for the final result:

```
x := linalg::matlinsolve(A, b, Normal = FALSE)
```

$$\begin{pmatrix} s - \frac{s\,(t\,(s+1)-t+1)}{s\,t+1} + 1 \\ \frac{t\,(s+1)-t+1}{s\,t+1} \end{pmatrix}$$

Example 8

Solve this system:

```
A := matrix([[1, s], [1, t]]):
b := matrix([1, 1]):

linalg::matlinsolve(A, b)
```

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Note that more solutions exist for `t = s`. `linalg::matlinsolve` omits these solutions because it makes some additional assumptions on symbolic parameters of this system. To see the assumptions that `linalg::matlinsolve` made while solving this system, use the `ShowAssumptions` option:

```
linalg::matlinsolve(A, b, ShowAssumptions)
```

$$\left[\begin{pmatrix}1\\0\end{pmatrix}, [], [], [t-s\neq 0]\right]$$

Parameters

A

$m\times n$ matrix of a domain of category `Cat::Matrix`

B

$m\times k$ matrix of a domain of category `Cat::Matrix`

b

m-dimensional column vector, i.e., a $m\times 1$ matrix of a domain of category `Cat::Matrix`

list

List of n elements of the component ring of `A`

Options

Normal

Option, specified as `Normal = b`

Return normalized results. The value `b` must be `TRUE` or `FALSE`. By default, `Normal = TRUE`, meaning that `linalg::matlinsolve` guarantees normalization of the returned results. Normalizing results can be computationally expensive.

By default, `linalg::matlinsolve` calls `normal` before returning results. This additional internal call ensures that the final result is normalized. This call can be computationally expensive. This option affects the output only if the solution contains variables or exact expressions, such as `sqrt(5)` or `sin(PI/7)`.

To avoid this additional call, specify `Normal = FALSE`. In this case, `linalg::matlinsolve` also can return normalized results, but does not guarantee such normalization. See “Example 7” on page 15-132.

ShowAssumptions

Return information about internal assumptions that `linalg::matlinsolve` made on symbolic parameters in `eqs`.

With `ShowAssumptions`, `linalg::matlinsolve` returns a list `[S, KernelBasis, Constraints, Pivots]`. The lists `Constraints` and `Pivots` contain equations and inequalities involving symbolic parameters in `A` and `b` (or `B`). Internally, these were assumed to hold true when solving the system. See “Example 8” on page 15-132.

When Gaussian elimination produces an equation `0 = c` with nonzero `c`, `linalg::matlinsolve` without `ShowAssumptions` returns `[]`. If `c` involves symbolic parameters, try using `linalg::matlinsolve` with `ShowAssumptions` to solve such systems. If the system is solvable, you will get the solution. In this case, an equation `0 = c` is returned in the `Constraints` list. If the system is not solvable, `linalg::matlinsolve` with `ShowAssumptions` returns `[[], [], [], []]`.

Special

Only one particular solution `w` of the system $A\vec{x} = \vec{b}$ is returned. This supresses the computation of a basis for the kernel of `A`.

Unique

Checks whether the system has a unique solution and returns it. The return value `NIL` means that the system has more than one solution.

Return Values

Without `ShowAssumptions`, `linalg::matlinsolve` can return a vector or a list `[S, KernelBasis]` (possibly empty), where `S` is a solution vector and `KernelBasis` is a list of basis vectors for the kernel of `A`. It also can return a matrix or the value `NIL`.

The matrix and the vectors, respectively, are of the domain type `Dom::Matrix(R)`, where `R` is the component ring of `A`.

With `ShowAssumptions`, `linalg::matlinsolve` returns a list `[S, KernelBasis, Constraints, Pivots]`. The lists `Constraints` and `Pivots` contain equations and inequalities involving symbolic parameters in `A` and `b` (or `B`). Internally, these

were assumed to hold true when solving the system. If the system is not solvable, `linalg::matlinsolve` with `ShowAssumptions` returns `[[], [], [], []]`.

Algorithms

Let A be an $m \times n$ matrix with components from a field F and $\vec{b}$ an m-dimensional vector over F. Let $\left(A, \vec{b}\right)$ be the extended coefficient matrix of the linear system $A\,\vec{x} = \vec{b}$.

Then the following holds:

- The linear system $A\,\vec{x} = \vec{b}$ has a solution, if and only if $\mathrm{rank}\left(A, \vec{b}\right) = \mathrm{rank}(A)$.
- It has exactly one solution, if and only if $\mathrm{rank}\left(A, \vec{b}\right) = \mathrm{rank}(A) = n$.
- If $\vec{x}_s$ is a solution of the system $A\,\vec{x} = \vec{b}$ and $\left\{\vec{v}_1, \ldots, \vec{v}_r\right\}$ a basis of the kernel of A, then

  $$L\left(A, \vec{b}\right) = \left\{\vec{w} + \lambda_1\,\vec{v}_1 + \lambda_2\,\vec{v}_2 + \cdots + \lambda_r\,\vec{v}_r \;\middle|\; \lambda_1, \ldots, \lambda_r \in K\right\}$$

  is the set of all solutions of the linear system $A\,\vec{x} = \vec{b}$, the *general solution* of the (inhomogeneus) linear system.

The *kernel of the matrixA* is defined as:

$$\ker(A) = \left\{\vec{w} \;\middle|\; A\,\vec{w} = \vec{0}\right\}.$$

The kernel of A is a vector space over F of dimension n - *rank*(A).

See Also

MuPAD Functions

`linalg::expr2Matrix` | `linalg::matlinsolveLU` | `linalg::nullspace` | `linalg::wiedemann` | `linsolve` | `numeric::matlinsolve`

More About

- "Solve Algebraic Systems"

linalg::matlinsolveLU

Solving the linear system given by an LU decomposition

Syntax

```
linalg::matlinsolveLU(L, U, b)
```

```
linalg::matlinsolveLU(L, U, B)
```

Description

`linalg::matlinsolveLU(L, U, b)` solves the linear system $L\,U\,\vec{x} = \vec{b}$, where the matrices L and U form an LU-decomposition, as computed by `linalg::factorLU`.

If the third parameter is an $n \times k$ matrix `B` then the result is an $n \times k$ matrix X satisfying the matrix equation $L\,U\,X = B$.

The system to be solved always has a unique solution.

The diagonal entries of the lower diagonal matrix `L` must be equal to one (*Doolittle*-decomposition, see `linalg::factorLU`).

`linalg::matlinsolveLU` expects `L` and `U` to be nonsingular.

`linalg::matlinsolveLU` does not check on any of the required properties of `L` and `U`.

The component ring of the matrices `L` and `U` must be a field, i.e., a domain of category `Cat::Field`.

The parameters must be defined over the same component ring.

Examples

Example 1

We solve the system

$$\begin{pmatrix} 2 & -3 & -1 \\ 1 & 1 & -1 \\ & 1 & -1 \end{pmatrix} X = \begin{pmatrix} 1 & & \\ & 1 & \\ & & 1 \end{pmatrix}$$:

```
MatR := Dom::Matrix(Dom::Real):
A  := MatR([[2, -3, -1], [1, 1, -1], [0, 1, -1]]);
I3 := MatR::identity(3)
```

$$\begin{pmatrix} 2 & -3 & -1 \\ 1 & 1 & -1 \\ 0 & 1 & -1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

We start by computing an LU-decomposition of A:

```
LU := linalg::factorLU(A)
```

$$\left[\begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ 0 & \frac{2}{5} & 1 \end{pmatrix}, \begin{pmatrix} 2 & -3 & -1 \\ 0 & \frac{5}{2} & -\frac{1}{2} \\ 0 & 0 & -\frac{4}{5} \end{pmatrix}, [1, 2, 3] \right]$$

Now we solve the system $A\,X = I_3$, which gives us the inverse of A:

```
Ai := linalg::matlinsolveLU(LU[1], LU[2], I3)
```

$$\begin{pmatrix} 0 & 1 & -1 \\ -\frac{1}{4} & \frac{1}{2} & -\frac{1}{4} \\ -\frac{1}{4} & \frac{1}{2} & -\frac{5}{4} \end{pmatrix}$$

```
A * Ai, Ai * A
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Parameters

L

An $n \times n$ lower triangular matrix of a domain of category `Cat::Matrix`

U

An $n \times n$ upper triangular form matrix of the same domain as `L`

B

An $n \times k$ matrix of a domain of category `Cat::Matrix`

b

An n-dimensional column vector, i.e., an $n \times 1$ matrix of a domain of category `Cat::Matrix`

Return Values

n-dimensional solution vector or $n \times k$ dimensional solution matrix, respectively, of the domain type `Dom::Matrix(R)`, where `R` is the component ring of `A`.

See Also

MuPAD Functions

`linalg::factorLU` | `linalg::inverseLU` | `linalg::matlinsolve`

linalg::minpoly

Minimal polynomial of a matrix

Syntax

```
linalg::minpoly(A, x)
```

Description

`linalg::minpoly(A, x)` computes the minimal polynomial of the square matrix A in x, i.e., the monic polynomial of lowest degree annihilating the matrix A.

The minimal polynomial of `A` divides the characteristic polynomial of `A`, by Cayley-Hamilton theorem.

If the matrix is defined over `Dom::Float`, then due to numerical errors the computed polynomial can have a degree higher than the dimension of the matrix. In such cases, `linalg::minpoly` returns the value `FAIL`. See "Example 3" on page 15-142.

The component ring of `A` must be a field, i.e., a domain of category `Cat::Field`.

Examples

Example 1

We define the following matrix over the rational numbers:

```
A := Dom::Matrix(Dom::Rational)(
  [[0, 2, 0], [0, 0, 2], [2, 0, 0]]
)
```

$$\begin{pmatrix} 0 & 2 & 0 \\ 0 & 0 & 2 \\ 2 & 0 & 0 \end{pmatrix}$$

The minimal polynomial of the matrix A in the variable x is then given by:

```
delete x:  linalg::minpoly(A, x)
```

$$x^3 - 8$$

In this case, the minimal polynomial is in fact equal to the characteristic polynomial of A:

```
linalg::charpoly(A, x)
```

$$x^3 - 8$$

Example 2

The minimal polynomial of the matrix:

```
B := matrix([[0, 1, 0], [0, 0, 0], [0, 0, 0]])
```

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

is a polynomial of degree 2:

```
m := linalg::minpoly(B, x)
```

$$x^2$$

The characteristic polynomial of B has degree 3 and is divided by the minimal polynomial of B:

```
p := linalg::charpoly(B, x)
```

$$x^3$$

```
p / m
```

$$x$$

Example 3

For the following example, MuPAD is not able to compute the minimal polynomial, and thus `FAIL` is returned:

```
C := Dom::Matrix(Dom::Float)([
   [7, 0, 0, 0, 0], [1, 0, 0, 0, 0], [1, 2, 0, 0, 0],
   [1, 2, 3, 0, 0], [1, 2, 3, 4, 7]
])
```

$$\begin{pmatrix} 7.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 2.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 2.0 & 3.0 & 0.0 & 0.0 \\ 1.0 & 2.0 & 3.0 & 4.0 & 7.0 \end{pmatrix}$$

```
delete x:  linalg::minpoly(C, x)
```

```
Warning: Cannot compute the minimal polynomial. [linalg::minpoly]
```

FAIL

In fact, for this example MuPAD is not able to check for zero equivalence during Gaussian elimination and therefore chose a wrong pivot element.

If you perform the computation over the coefficient domain`Dom::ExpressionField(normal)` instead, then in most cases the minimal polynomial can be computed:

```
C := matrix([
   [7, 0, 0, 0, 0], [1, 0, 0, 0, 0], [1, 2, 0, 0, 0],
   [1, 2, 3, 0, 0], [1, 2, 3, 4, 7]
])
```

$$\begin{pmatrix} 7 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 \\ 1 & 2 & 3 & 0 & 0 \\ 1 & 2 & 3 & 4 & 7 \end{pmatrix}$$

```
linalg::minpoly(C, x)
```

$$x^5 - 14\,x^4 + 49\,x^3$$

However, in general this problem regarding zero recognition cannot be avoided.

Parameters

A

A square matrix of a domain of category `Cat::Matrix`

x

An indeterminate

Return Values

Polynomial of the domain `Dom::DistributedPolynomial([x],R)`, where `R` is the component ring of `A`, or the value `FAIL`.

See Also

MuPAD Functions

`linalg::charpoly` | `linalg::frobeniusForm`

linalg::multCol

Multiply columns with a scalar

Syntax

```
linalg::multCol(A, c, s)
```

```
linalg::multCol(A, c1 .. c2, s)
```

```
linalg::multCol(A, list, s)
```

Description

`linalg::multCol(A, c, s)` returns a copy of the matrix A resulting from A by multiplying the c-th column of A with the scalar s.

`linalg::multCol(A, c1.. c2, s)` returns a copy of the matrix `A` obtained from `A` by multiplying those columns whose indices are in the range `c1.. c2` with the scalar `s`.

`linalg::multCol(A, list, s)` returns a copy of the matrix `A` obtained from matrix `A` by multiplying those columns whose indices are contained in `list` with the scalar `s`.

The scalar `s` is converted into an element of the component ring of the matrix `A`. An error message is returned if the conversion fails.

Examples

Example 1

We define the following matrix:

```
A := matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

and illustrate the three different input formats for `linalg::multCol`:

```
linalg::multCol(A, 2, -1)
```

$$\begin{pmatrix} 1 & -2 & 3 \\ 4 & -5 & 6 \\ 7 & -8 & 9 \end{pmatrix}$$

```
linalg::multCol(A, 1..2, 2)
```

$$\begin{pmatrix} 2 & 4 & 3 \\ 8 & 10 & 6 \\ 14 & 16 & 9 \end{pmatrix}$$

```
linalg::multCol(A, [3, 1], 0)
```

$$\begin{pmatrix} 0 & 2 & 0 \\ 0 & 5 & 0 \\ 0 & 8 & 0 \end{pmatrix}$$

Parameters

A

An $m \times n$ matrix of a domain of category `Cat::Matrix`

c

The column index: a positive integer less or equal to n

c_1 .. c_2

A range of column indices (positive integers less or equal to n)

list

A list of column indices (positive integers less or equal to n)

Return Values

Matrix of the same domain type as A.

See Also

MuPAD Functions

`linalg::addCol` | `linalg::addRow` | `linalg::multRow`

linalg::multRow

Multiply rows with a scalar

Syntax

```
linalg::multRow(A, r, s)
```

```
linalg::multRow(A, r1 .. r2, s)
```

```
linalg::multRow(A, list, s)
```

Description

`linalg::multRow(A, r, s)` returns a copy of the matrix *A* resulting from *A* by multiplying the *r*-th row of *A* with the scalar *s*.

`linalg::multRow(A, r1.. r2, s)` returns a copy of the matrix `A` obtained from `A` by multiplying those rows whose indices are in the range $r_1 .. r_2$ with the scalar `s`.

`linalg::multRow(A, list, s)` returns a copy of the matrix `A` obtained from matrix `A` by multiplying those rows whose indices are contained in `list` with the scalar `s`.

The scalar `s` is converted into an element of the component ring of the matrix `A`. An error message is returned if the conversion fails.

Examples

Example 1

We define the following matrix:

```
A := matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

and illustrate the three different input formats for `linalg::multRow`:

```
linalg::multRow(A, 2, -1)
```

$$\begin{pmatrix} 1 & 2 & 3 \\ -4 & -5 & -6 \\ 7 & 8 & 9 \end{pmatrix}$$

```
linalg::multRow(A, 1..2, 2)
```

$$\begin{pmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \\ 7 & 8 & 9 \end{pmatrix}$$

```
linalg::multRow(A, [3, 1], 0)
```

$$\begin{pmatrix} 0 & 0 & 0 \\ 4 & 5 & 6 \\ 0 & 0 & 0 \end{pmatrix}$$

Parameters

`A`

An $m \times n$ matrix of a domain of category `Cat::Matrix`

`r`

The row index: a positive integer less or equal to m

r_1 `..` r_2

A range of row indices (positive integers less or equal to m)

`list`

A list of row indices (positive integers less or equal to m)

Return Values

Matrix of the same domain type as A.

See Also

MuPAD Functions

`linalg::addCol` | `linalg::addRow` | `linalg::multCol`

linalg::ncols

Number of columns of a matrix

Syntax

```
linalg::ncols(A)
```

Description

`linalg::ncols(A)` returns the number of columns of the matrix *A*.

Examples

Example 1

The matrix:

```
A:= matrix([[1, 2, 3, 4], [3, 1, 4], [5, 6]])
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 4 & 0 \\ 5 & 6 & 0 & 0 \end{pmatrix}$$

has four columns:

```
linalg::ncols(A)
```

$$4$$

Parameters

A

A matrix of a domain of category `Cat::Matrix`

Return Values

Positive integer.

See Also

MuPAD Functions

`linalg::matdim` | `linalg::nrows` | `linalg::vecdim`

linalg::nonZeros

Number of non-zero elements of a matrix

Syntax

```
linalg::nonZeros(A)
```

Description

`linalg::nonZeros(A)` returns the number of non-zero components of the matrix *A*.

Examples

Example 1

The matrix

```
MZ7 := Dom::Matrix(Dom::IntegerMod(7)):
A := MZ7([[18, -1], [4, 81]])
```

$$\begin{pmatrix} 4 \bmod 7 & 6 \bmod 7 \\ 4 \bmod 7 & 4 \bmod 7 \end{pmatrix}$$

has four non-zero entries:

```
linalg::nonZeros(A)
```

$$4$$

The matrix:

```
B := MZ7([[21, 2], [-1, 14]])
```

$$\begin{pmatrix} 0 \bmod 7 & 2 \bmod 7 \\ 6 \bmod 7 & 0 \bmod 7 \end{pmatrix}$$

has only two non-zero entries:

```
linalg::nonZeros(B)
```

$$2$$

Parameters

A

A matrix of a domain of category `Cat::Matrix`

Return Values

Nonnegative integer

linalg::normalize

Normalize a vector

Syntax

```
linalg::normalize(v)
```

Description

`linalg::normalize(v)` normalizes the vector $\vec{v}$ with respect to the 2-norm ($\|\vec{v}\|_2 = \sqrt{\langle \vec{v}, \vec{v} \rangle}$).

The result of `linalg::normalize(v)` is a vector that has norm 1 and the same direction as `v`.

The scalar product $\langle \vec{v}, \vec{v} \rangle$ for a vector $\vec{v}$ is implemented by the function `linalg::scalarProduct`.

The norm of a vector is computed with the function `norm`, which is overloaded for vectors. See the method `"norm"` of the domain constructor `Dom::Matrix` for details.

If the norm is an object that cannot be converted into an element of the component ring of `v`, then an error occurs (see “Example 2” on page 15-155).

Examples

Example 1

We define the following vector:

```
u := matrix([[1, 2]])
```

$$\begin{pmatrix} 1 & 2 \end{pmatrix}$$

Then the vector of norm 1 with the same direction as u is given by:

```
linalg::normalize(u)
```

$$\left(\frac{\sqrt{5}}{5}\ \frac{2\sqrt{5}}{5}\right)$$

Example 2

The following computation fails because the vector (1, 2) cannot be normalized over the rationals:

```
v := Dom::Matrix(Dom::Rational)([[1, 2]]):
linalg::normalize(v)
```

```
Error: Cannot normalize the given vector over its component ring. [linalg::normalize]
```

If we define v over the real numbers, then we get the normalized vector of v as follows:

```
w := Dom::Matrix(Dom::Real)(v): linalg::normalize(w)
```

$$\left(\frac{\sqrt{5}}{5}\ \frac{2\sqrt{5}}{5}\right)$$

Parameters

v

A vector, i.e., an $n\times 1$ or $1\times n$ matrix of a domain of category `Cat::Matrix`

Return Values

Vector of the same domain type as v.

See Also

MuPAD Functions

`linalg::scalarProduct` | `norm`

linalg::nrows

Number of rows of a matrix

Syntax

```
linalg::nrows(A)
```

Description

`linalg::nrows(A)` returns the number of rows of the matrix *A*.

Examples

Example 1

The matrix:

```
A := matrix([[1, 2, 3, 4], [3, 1, 4], [5, 6]])
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 4 & 0 \\ 5 & 6 & 0 & 0 \end{pmatrix}$$

has three rows:

```
linalg::nrows(A)
```

$$3$$

Parameters

A

A matrix of a domain of category `Cat::Matrix`

Return Values

Positive integer.

See Also

MuPAD Functions

`linalg::matdim` | `linalg::ncols` | `linalg::vecdim`

linalg::nullspace

Basis for the null space of a matrix

Syntax

```
linalg::nullspace(A)
```

Description

`linalg::nullspace(A)` returns a basis for the null space of the matrix A, i.e., a list B of linearly independent vectors such that $A\,\vec{x} = \vec{0}$ if and only if $\vec{x}$ is a linear combination of the vectors in B.

The component ring of the matrix `A` must be a field, i.e., a domain of category `Cat::Field`.

If the component ring of `A` does not have a canonical representation of the zero element, it can happen that `linalg::nullspace` does not find a basis for the null space. In such a case, a wrong result is returned.

Examples

Example 1

The kernel of the matrix:

```
A := Dom::Matrix(Dom::Real)(
  [[3^(1/2)*2 - 2, 2], [4, 3^(1/2)*2 + 2]]
)
```

$$\begin{pmatrix} 2\sqrt{3}-2 & 2 \\ 4 & 2\sqrt{3}+2 \end{pmatrix}$$

is one-dimensional, and a basis is $\left\{ \begin{pmatrix} -\frac{1}{\sqrt{3}-1} \\ 1 \end{pmatrix} \right\}$:

```
linalg::nullspace(A)
```

$$\left[\begin{pmatrix} -\frac{1}{\sqrt{3}-1} \\ 1 \end{pmatrix} \right]$$

Parameters

A

A matrix of a domain of category `Cat::Matrix`

Return Values

List of (column) vectors of the domain `Dom::Matrix(R)`, where `R` is the component ring of `A`.

See Also

MuPAD Functions

`linalg::basis` | `linalg::matlinsolve` | `linsolve` | `numeric::matlinsolve`

linalg::ogCoordTab

Table of orthogonal coordinate transformations

Syntax

```
linalg::ogCoordTab[ogName](u1, u2, u3, <c>)

linalg::ogCoordTab[ogName,
  Transformation](u1, u2, u3, <c>)

linalg::ogCoordTab[ogName,
  InverseTransformation](u1, u2, u3, <c>)

linalg::ogCoordTab[ogName,
  UnitVectors](u1, u2, u3, <c>)

linalg::ogCoordTab[ogName,
  Scales](u1, u2, u3, <c>)

linalg::ogCoordTab[ogName, Ranges](<c>)

linalg::ogCoordTab[ ogName , Dimension]
```

Description

`linalg::ogCoordTab` is a table of predefined orthogonal coordinate transformations in $\mathbb{R}^3$.

The entry associated with `ogName` defines a coordinate transformation $\vec{x} = \vec{x}(\vec{u})$ which maps the orthogonal parameters $\vec{u} = (u_1, u_2, u_3)$ to a vector $\vec{x} = (x_1, x_2, x_3)$ in Cartesian coordinates.

The coordinate systems `EllipticCylindrical` and `Torus` are defined with a constant parameter `c` which has to be passed as an additional argument. See “Example 2” on page 15-164.

The following coordinate transformations are stored in `linalg::ogCoordTab`. They are invertible for the indicated parameter values:

- `Cartesian`:

$u_1 \in \mathbb{R}$, $u_2 \in \mathbb{R}$, $u_3 \in \mathbb{R}$:

$$x_1 = u_1,\ x_2 = u_2,\ x_3 = u_3$$

- `Spherical` and its equivalent `Spherical[RightHanded]`:

  $0 < u_1 < \infty$, $0 \le u_2 \le \pi$, $0 \le u_3 < 2\pi$:

  $$x_1 = u_1 \sin(u_2) \cos(u_3),\ x_2 = u_1 \sin(u_2) \sin(u_3),\ x_3 = u_1 \cos(u_2)$$

- `Spherical[LeftHanded]`:

  $0 < u_1 < \infty$, $0 \le u_2 < 2\pi$, $0 \le u_3 \le \pi$:

  $$x_1 = u_1 \cos(u_2) \sin(u_3),\ x_2 = u_1 \sin(u_2) \sin(u_3),\ x_3 = u_1 \cos(u_3)$$

- `Cylindrical`:

  $0 < u_1 < \infty$, $0 \le u_2 < 2\pi$, $u_3 \in \mathbb{R}$:

  $$x_1 = u_1 \cos(u_2),\ x_2 = u_1 \sin(u_2),\ x_3 = u_3$$

- `EllipticCylindrical`:

  $0 < u_1 < \infty$, $0 \le u_2 < 2\pi$, $u_3 \in \mathbb{R}$ (with a real constant c):

  $$x_1 = c \cos(u_1) \cos(u_2),\ x_2 = c \sinh(u_1) \sin(u_2),\ x_3 = u_3$$

- `ParabolicCylindrical`:

  $0 < u_1 < \infty$, $u_2 \in \mathbb{R}$, $u_3 \in \mathbb{R}$:

  $$x_1 = \frac{\left({u_1}^2 - {u_2}^2\right)}{2},\ x_2 = u_1\, u_2,\ x_3 = u_3$$

- `RotationParabolic`:

  $0 < u_1 < \infty$, $0 < u_2 < \infty$, $0 \le u_3 < 2\pi$:

$$x_1 = u_1\, u_2 \cos(u_3),\ x_2 = u_1\, u_2 \sin(u_3),\ x_3 = \frac{\left(u_1{}^2 - u_2{}^2\right)}{2}$$

- `Torus`:

  $0 < u_1 < c$, $0 \le u_2 < 2\,\pi$, $0 \le u_3 < 2\,\pi$ (with a positive constant c):

$$x_1 = \left(c - u_1 \cos(u_2)\right) \cos(u_3),\ x_2 = \left(c - u_1 \cos(u_2)\right) \sin(u_3),\ x_3 = u_1 \sin(u_2)$$

`linalg::ogCoordTab` is used by functions such as `curl`, `divergence`, `gradient`, and `laplacian` to perform computations in non-Cartesian coordinates.

Examples

Example 1

The following call returns the Cartesian vector $\vec{x} = [x, y, z]$ in terms of the right-handed spherical coordinates $\vec{u} = [u_1, u_2, u_3]$:

```
linalg::ogCoordTab[Spherical[RightHanded],
                           Transformation](u1, u2, u3)
```

$$[u1 \cos(u3) \sin(u2),\ u1 \sin(u2) \sin(u3),\ u1 \cos(u2)]$$

The spherical coordinates expressed by the Cartesian coordinates:

```
linalg::ogCoordTab[Spherical[RightHanded],
                    InverseTransformation](x, y, z)
```

$$\left[\sqrt{x^2 + y^2 + z^2},\ \arccos\left(\frac{z}{\sqrt{x^2 + y^2 + z^2}}\right),\ \arccos\left(\frac{x}{\sqrt{x^2 + y^2}}\right)\right.$$

$$\left. + \operatorname{sign}(y)\,(\operatorname{sign}(y) - 1)\left(\pi - \arccos\left(\frac{x}{\sqrt{x^2 + y^2}}\right)\right)\right]$$

Note the `sign(y)` in the expression for u_3. This ensures that the correct angle is returned for any value of y:

```
assume(y > 0):
linalg::ogCoordTab[Spherical[RightHanded],
                   InverseTransformation](x, y, z)
```

$$\left[\sqrt{x^2+y^2+z^2},\ \arccos\left(\frac{z}{\sqrt{x^2+y^2+z^2}}\right),\ \arccos\left(\frac{x}{\sqrt{x^2+y^2}}\right)\right]$$

```
linalg::ogCoordTab[Spherical[RightHanded],
                   InverseTransformation](1, 1, 0),
linalg::ogCoordTab[Spherical[RightHanded],
                   InverseTransformation](-1, 0, 1),
linalg::ogCoordTab[Spherical[RightHanded],
                   InverseTransformation](1, 0, 2),
linalg::ogCoordTab[Spherical[RightHanded],
                   InverseTransformation](1, -1, 3)
```

$$\left[\sqrt{2},\ \frac{\pi}{2},\ \frac{\pi}{4}\right],\ \left[\sqrt{2},\ \frac{\pi}{4},\ \pi\right],\ \left[\sqrt{5},\ \arccos\left(\frac{2\sqrt{5}}{5}\right),\ 0\right],\ \left[\sqrt{11},\ \arccos\left(\frac{3\sqrt{11}}{11}\right),\ \frac{7\pi}{4}\right]$$

These parameter values are from the following ranges:

```
linalg::ogCoordTab[Spherical[RightHanded], Ranges]()
```

$$[0..\infty,\ 0..\pi,\ 0..2\,\pi]$$

The following orthonormal vectors are tangent to the spherical parameter lines:

```
linalg::ogCoordTab[Spherical[RightHanded], UnitVectors](u1, u2, u3)
```

$$[[\cos(u3)\sin(u2),\ \sin(u2)\sin(u3),\ \cos(u2)],\ [\cos(u2)\cos(u3),\ \cos(u2)\sin(u3),\ -\sin(u2)],\ [-\sin(u3),\ \cos(u3),\ 0]]$$

The `scaling factors' are:

```
linalg::ogCoordTab[Spherical[RightHanded],Scales](u1, u2, u3)
```

$$[1, u1, u1 \sin(u2)]$$

There is the following relationship between the Jacobian of the transformation $\vec{u} \to \vec{x}$ from the orthogonal coordinates to the Cartesian coordinates:

```
xyz:= linalg::ogCoordTab[Spherical[RightHanded],
                                  Transformation](u1, u2, u3):
unitvectors:= linalg::ogCoordTab[Spherical[RightHanded],
                                     UnitVectors](u1, u2, u3):
scales:= linalg::ogCoordTab[Spherical[RightHanded],
                                        Scales](u1, u2, u3):

linalg::transpose(jacobian(xyz, [u1, u2, u3])) =
  matrix(3, 3, scales, Diagonal)* matrix(unitvectors)
```

$$\sigma_1 = \sigma_1$$

where

$$\sigma_1 = \begin{pmatrix} \cos(u3)\sin(u2) & \sin(u2)\sin(u3) & \cos(u2) \\ u1\cos(u2)\cos(u3) & u1\cos(u2)\sin(u3) & -u1\sin(u2) \\ -u1\sin(u2)\sin(u3) & u1\cos(u3)\sin(u2) & 0 \end{pmatrix}$$

```
delete y, xyz, unitvectors, scales:
```

Example 2

The following call returns the Cartesian vector $\vec{x} = [x, y, z]$ in terms of elliptic cylindrical coordinates $\vec{u} = [u, v, w]$ involving a parameter c:

```
linalg::ogCoordTab[EllipticCylindrical, Transformation](u, v, z, c)
```

$$[c\cosh(u)\cos(v),\ c\sinh(u)\sin(v),\ z]$$

We compute the gradient of the function $f(u, v, w) = x(u, v, w)$ in elliptic cylindrical coordinates $\vec{u} = [u, v, w]$.

```
f:= (c*cos(v)*cosh(u))^2:
```

For computing the components of the gradient with respect to an orthogonal system, it is sufficient to know the 'scale parameters':

```
linalg::ogCoordTab[EllipticCylindrical, Scales](u, v, w, c)
```

$$\left[c\sqrt{\cosh(u)^2-\cos(v)^2},\ c\sqrt{\cosh(u)^2-\cos(v)^2},\ 1\right]$$

```
gradf := gradient(f, [u, v, w], %)
```

$$\begin{pmatrix} \frac{2\,c\cosh(u)\cos(v)^2\sinh(u)}{\sqrt{\cosh(u)^2-\cos(v)^2}} \\ -\frac{2\,c\cosh(u)^2\cos(v)\sin(v)}{\sqrt{\cosh(u)^2-\cos(v)^2}} \\ 0 \end{pmatrix}$$

These are the coefficients of the gradient with respect to the orthonormal basis $\vec{e}_u, \vec{e}_v, \vec{e}_w$ returned via the option `UnitVectors`:

```
[e_u, e_v, e_w] :=
linalg::ogCoordTab[EllipticCylindrical, UnitVectors](u, v, w, c)
```

$$[[\sigma_1, \sigma_2, 0], [-\sigma_2, \sigma_1, 0], [0, 0, 1]]$$

where

$$\sigma_1 = \frac{\cos(v)\sinh(u)}{\sqrt{\cosh(u)^2-\cos(v)^2}}$$

$$\sigma_2 = \frac{\cosh(u)\sin(v)}{\sqrt{\cosh(u)^2-\cos(v)^2}}$$

We convert the lists `e_u`, `e_v`, `e_w` into column vectors via `matrix`. Thus, in the standard basis of $\mathbb{R}^3$, the gradient vector field is:

```
G :=   gradf[1]*matrix(e_u)
     + gradf[2]*matrix(e_v)
     + gradf[3]*matrix(e_w)
```

$$\begin{pmatrix} \frac{2\,c\,\cosh(u)\,\cos(v)^3\,\sinh(u)^2}{\cosh(u)^2-\cos(v)^2} + \frac{2\,c\,\cosh(u)^3\,\cos(v)\,\sin(v)^2}{\cosh(u)^2-\cos(v)^2} \\ 0 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} \frac{2\,c\,\cosh(u)\,\cos(v)^3\,\sinh(u)^2}{\cosh(u)^2-\cos(v)^2} + \frac{2\,c\,\cosh(u)^3\,\cos(v)\,\sin(v)^2}{\cosh(u)^2-\cos(v)^2} \\ 0 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} \frac{2\,c\,\cosh(u)\,\cos(v)^3\,\sinh(u)^2}{\cosh(u)^2-\cos(v)^2} + \frac{2\,c\,\cosh(u)^3\,\cos(v)\,\sin(v)^2}{\cosh(u)^2-\cos(v)^2} \\ 0 \\ 0 \end{pmatrix}$$

We simplify this expression using the identities $sin^2(v) = 1 - cos^2(v)$, $sinh^2(u) = cosh^2(u) - 1$:

```
normal(subs(G, sin(v)^2 = 1 - cos(v)^2,
               sinh(u)^2 = cosh(u)^2 - 1))
```

$$\begin{pmatrix} 2\,c\,\cosh(u)\,\cos(v) \\ 0 \\ 0 \end{pmatrix}$$

This is the gradient of the function $f(x, y, z) = x^2$ with x expressed by elliptic cylindrical coordinates:

```
G := gradient(x^2, [x, y, z])
```

$$\begin{pmatrix} 2\,x \\ 0 \\ 0 \end{pmatrix}$$

```
[x, y, z] :=
linalg::ogCoordTab[EllipticCylindrical, Transformation](u, v, w, c)
```

$$[c\,\cosh(u)\,\cos(v),\ c\,\sinh(u)\,\sin(v),\ w]$$

```
map(G, eval)
```

$$\begin{pmatrix} 2\,c\,\cosh(u)\,\cos(v) \\ 0 \\ 0 \end{pmatrix}$$

```
delete f, gradf, e_u, e_v, e_w, G, x, y, z:
```

Parameters

`ogName`

The name of a predefined coordinate system. The following 3 dimensional coordinate systems are available: `Cartesian`, `Spherical` (equivalent to `Spherical[RightHanded]`), `Spherical[LeftHanded]`, `Cylindrical`, `EllipticCylindrical`, `ParabolicCylindrical`, `RotationParabolic`, `Torus`.

u_1, u_2, u_3

The coordinates of the orthogonal system: identifiers, indexed identifiers, or arithmetical expressions.

x_1, x_2, x_3

Cartesian coordinates: identifiers, indexed identifiers, or arithmetical expressions.

`c`

An arithmetical expression. The default value is `c` = 1.

Options

Transformation

`linalg::ogCoordTab [ogName, Transformation]( u1, u2, u3, c )` returns a list of arithmetical expressions `[ x1( u1, u2, u3), x2( u1, u2, u3), x3( u1, u2, u3)]` defining the transformation from the orthogonal coordinates u_i to the Cartesian coordinates x_j. The transformation is invertible if the coordinates u_i are from the range $a_i < u_i < b_i$ where `[a1..b1, a2..b2, a3..b3]` = `linalg::ogCoordTab [ogName, Ranges](c)`.

InverseTransformation

`linalg::ogCoordTab [ogName, InverseTransformation]( x1, x2, x3, c )` returns a list of arithmetical expressions `[ u1( x1, x2, x3), u2( x1, x2, x3), u3( x1, x2, x3)]` defining the inverse transformation. The inverse transformation produces parameter values u_i in the range $a_i \le u_i \le b_i$ given by `[a1..b1, a2..b2, a3..b3]` = `linalg::ogCoordTab [ogName, Ranges](c)`.

UnitVectors

`linalg::ogCoordTab [ogName, UnitVectors]( u1, u2, u3, c )` returns a list of orthonormal vectors $\left[\vec{e}_1, \vec{e}_2, \vec{e}_3\right]$, where each vector is represented by a list of three arithmetical expressions. These vectors $\vec{e}_i = \frac{1}{s_i}\frac{\partial}{\partial u_i}\vec{x}$ with $s_i = \left|\frac{\partial}{\partial u_i}\vec{x}\right|$ are the unit vectors tangent to the parameter lines `u`$_i$.

Scales

`linalg::ogCoordTab [ogName, Scales]( u1, u2, u3, c )` returns a list `[ s1, s2, s3]` of "scaling factors" of the transformation $\vec{u} \to \vec{x}$. The "scales" are the Euclidean lengths $s_i = \left|\frac{\partial}{\partial u_i}\vec{x}\right|$ of the vectors $\frac{\partial}{\partial u_i}\vec{x}$ tangent to the parameter lines `u`$_i$.

Return Values

Most of the entries in `linalg::ogCoordTab` are functions:

`linalg::ogCoordTab [ogName, Transformation]( u1, u2, u3, c )` returns a list of arithmetical expressions `[ x1( u1, u2, u3), x2( u1, u2, u3), x3( u1, u2, u3)]` defining the transformation from the orthogonal coordinates u_i to the Cartesian coordinates x_j.

`linalg::ogCoordTab [ogName, InverseTransformation]( x1, x2, x3, c )` returns a list of arithmetical expressions `[ u1( x1, x2, x3), u2( x1, x2, x3), u3( x1, x2, x3)]` defining the inverse transformation.

`linalg::ogCoordTab [ogName, UnitVectors]( u1, u2, u3, c )` returns a list of orthogonal unit "vectors." The "vectors" are given as lists of arithmetical expressions.

`linalg::ogCoordTab [ogName, Scales]( u1, u2, u3, c )` returns a list of arithmetical expressions.

`linalg::ogCoordTab [ogName, Ranges]( c )` returns a list of ranges `[ a1.. b1, a2.. b2, a3.. b3]`. The transformation is invertible for parameter values $a_i < u_i < b_i$.

`linalg::ogCoordTab [ogName, Dimension]` yields the dimension of the space parametrized by the orthogonal coordinates. Presently, all predefined systems parametrize $\mathbb{R}^3$, i.e., the dimension is 3 in all cases.

The call `linalg::ogCoordTab [ogName]( u1, u2, u3, c )` is identical to the call `linalg::ogCoordTab [ogName, UnitVectors]( u1, u2, u3, c )`.

See Also

MuPAD Functions

`curl` | `divergence` | `gradient` | `hessian` | `jacobian` | `laplacian`

linalg::orthog

Orthogonalization of vectors

Syntax

```
linalg::orthog(S, <Real>)
```

Description

`linalg::orthog(S)` orthogonalizes the vectors in S using the Gram-Schmidt orthogonalization algorithm.

The vectors in `S` are orthogonalized with respect to the scalar product `linalg::scalarProduct`.

If O is the returned set, then the vectors of O span the same subspace as the vectors in `S`, and they are pairwise orthogonal, i.e.: $\vec{v}\,\vec{w} = 0$ for all $\vec{v}, \vec{w} \in O$ with $\vec{v} \neq \vec{w}$.

The vectors returned are not normalized. To normalize them use `map(O, linalg::normalize)`.

For an ordered set of orthogonal vectors, `S` should be a list.

The vectors in `S` must be defined over the same component ring.

The component ring of the vectors in `S` must be a field, i.e., a domain of category `Cat::Field`.

If you use the `Real` option, `linalg::orthog` computes an orthogonal basis using a real scalar product in the orthogonalization process.

Examples

Example 1

The following list of vectors is a basis of the vector space $\mathbb{R}^3$:

```
MatR := Dom::Matrix(Dom::Real):
S := [MatR([2, 1, 0]), MatR([-3, 1, 1]), MatR([-1, -1, -1])]
```

$$\left[\begin{pmatrix}2\\1\\0\end{pmatrix}, \begin{pmatrix}-3\\1\\1\end{pmatrix}, \begin{pmatrix}-1\\-1\\-1\end{pmatrix}\right]$$

The Gram-Schmidt algorithm then returns an orthogonal basis for $\mathbb{R}^3$. We get an orthonormal basis with the following input:

```
ON:= linalg::orthog(S)
```

$$\left[\begin{pmatrix}2\\1\\0\end{pmatrix}, \begin{pmatrix}-1\\2\\1\end{pmatrix}, \begin{pmatrix}-\frac{2}{15}\\\frac{4}{15}\\-\frac{2}{3}\end{pmatrix}\right]$$

The vectors can be normalized using `linalg::normalize`:

```
map(ON, linalg::normalize)
```

$$\left[\begin{pmatrix}\frac{2\sqrt{5}}{5}\\\frac{\sqrt{5}}{5}\\0\end{pmatrix}, \begin{pmatrix}-\frac{\sqrt{6}}{6}\\\frac{\sqrt{6}}{3}\\\frac{\sqrt{6}}{6}\end{pmatrix}, \begin{pmatrix}-\frac{\sqrt{2}\sqrt{15}}{30}\\\frac{\sqrt{2}\sqrt{15}}{15}\\-\frac{\sqrt{2}\sqrt{15}}{6}\end{pmatrix}\right]$$

We may also build a matrix from the vectors in `S` an apply `linalg::orthog` to this matrix. The result is the matrix whose columns are given by the above elements of the list `ON`:

```
A:= S[1].S[2].S[3]
```

$$\begin{pmatrix}2 & -3 & -1\\1 & 1 & -1\\0 & 1 & -1\end{pmatrix}$$

```
linalg::orthog(A)
```

$$\begin{pmatrix} 2 & -1 & -\frac{2}{15} \\ 1 & 2 & \frac{4}{15} \\ 0 & 1 & -\frac{2}{3} \end{pmatrix}$$

Example 2

The orthogonalization of the vectors:

```
T := {matrix([[-2, 5, 3]]), matrix([[0, 2, 1]])}
```

$$\{(0\ 2\ 1), (-2\ 5\ 3)\}$$

gives:

```
linalg::orthog(T)
```

$$\left\{(0\ 2\ 1), \left(-2\ -\frac{1}{5}\ \frac{2}{5}\right)\right\}$$

Example 3

The result of `linalg::orthog` is a list or set of linearly independent vectors, even if the input contains linearly dependent vectors:

```
MatQ := Dom::Matrix(Dom::Rational):
S := [MatQ([2, 1]), MatQ([3, 4]), MatQ([-1, 1])]
```

$$\left[\begin{pmatrix} 2 \\ 1 \end{pmatrix}, \begin{pmatrix} 3 \\ 4 \end{pmatrix}, \begin{pmatrix} -1 \\ 1 \end{pmatrix}\right]$$

```
linalg::orthog(S)
```

$$\left[\begin{pmatrix} 2 \\ 1 \end{pmatrix}, \begin{pmatrix} -1 \\ 2 \end{pmatrix}\right]$$

Example 4

Compute an orthogonal basis of this matrix:

```
A := matrix([[a, 1], [1, a]]):
linalg::orthog(A)
```

$$\begin{pmatrix} a & -\frac{a^2-1}{a\,\overline{a}+1} \\ 1 & -\frac{\overline{a}-a^2\,\overline{a}}{a\,\overline{a}+1} \end{pmatrix}$$

To avoid complex conjugates, use the `Real` option:

```
linalg::orthog(A, Real)
```

$$\begin{pmatrix} a & -\frac{a^2-1}{a^2+1} \\ 1 & \frac{a\left(a^2-1\right)}{a^2+1} \end{pmatrix}$$

Parameters

`S`

A set or list of vectors of the same dimension (a vector is an $n \times 1$ or $1 \times n$ matrix of a domain of category `Cat::Matrix`) or a matrix

Options

`Real`

Avoid using a complex scalar product in the orthogonalization process.

Return Values

Set or a list of vectors, respectively.

See Also

MuPAD Functions

`linalg::factorQR` | `linalg::isUnitary` | `linalg::normalize` | `linalg::scalarProduct` | `lllint` | `norm`

linalg::pascal

Pascal matrix

Syntax

```
linalg::pascal(n, <R>)
```

Description

`linalg::pascal(n)` returns the $n \times n$ Pascal matrix P given by $P_{i,j} = \binom{i+j-2}{i-1}$, $1 \le i, j \le n$.

The entries of Pascal matrices are integer numbers. Note, however, that the returned matrix is not defined over the component domain `Dom::Integer`, but over the standard component domain `Dom::ExpressionField()`. Thus, no conversion is necessary when working with other functions that expect or return matrices over that component domain.

Use `linalg::pascal(n, Dom::Integer)` to define the $n \times n$ Pascal matrix over the ring of integer numbers.

Inverse Pascal matrices are provided by `linalg::invpascal`.

Examples

Example 1

We construct the 3×3 Pascal matrix:

```
linalg::pascal(3)
```

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 3 & 6 \end{pmatrix}$$

This is a matrix of the domain `Dom::Matrix()`.

If you prefer a different component ring, the matrix may be converted to the desired domain after construction (see `coerce`, for example). Alternatively, one can specify the component ring when creating the Pascal matrix. For example, specification of the domain `Dom::Float` generates floating-point entries:

```
linalg::pascal(3, Dom::Float)
```

$$\begin{pmatrix} 1.0 & 1.0 & 1.0 \\ 1.0 & 2.0 & 3.0 \\ 1.0 & 3.0 & 6.0 \end{pmatrix}$$

```
domtype(%)
```

Dom::Matrix(Dom::Float)

Example 2

The Cholesky factor of a Pascal matrix consists of the elements of Pascal's triangle:

```
linalg::factorCholesky(linalg::pascal(4))
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 2 & 1 & 0 \\ 1 & 3 & 3 & 1 \end{pmatrix}$$

Parameters

`n`

The dimension of the matrix: a positive integer

`R`

The component ring: a domain of category `Cat::Rng`; default: `Dom::ExpressionField()`

Return Values

$n \times n$ matrix of the domain `Dom::Matrix(R)`.

Algorithms

Pascal matrices are symmetric and positive definite.

The determinant of a Pascal matrix is 1.

The inverse of a Pascal matrix has integer entries.

If λ is an eigenvalue of a Pascal matrix, then $\frac{1}{\lambda}$ is also an eigenvalue of the matrix.

See Also

MuPAD Functions

`linalg::hilbert` | `linalg::invhilbert` | `linalg::invpascal` | `linalg::invvandermonde` | `linalg::toeplitz` | `linalg::toeplitzSolve` | `linalg::vandermonde` | `linalg::vandermondeSolve`

linalg::permanent

Permanent of a matrix

Syntax

```
linalg::permanent(A)
```

Description

`linalg::permanent(A)` computes the permanent of the square matrix A.

The component ring of the matrix `A` must be a commutative ring, i.e., a domain of category `Cat::CommutativeRing`.

Examples

Example 1

We compute the permanent of the following matrix:

```
delete a11, a12, a21, a22:
A := matrix([[a11, a12], [a21, a22]])
```

$$\begin{pmatrix} a11 & a12 \\ a21 & a22 \end{pmatrix}$$

which gives us the general formula for the permanent of an arbitrary 2 ×2 matrix:

```
linalg::permanent(A)
```

$$a11\ a22 + a12\ a21$$

Example 2

The permanent of a matrix can be computed over arbitrary commutative rings. Let us create a random matrix defined over the ring $\mathbb{Z}_6$, the integers modulo 6:

```
B := linalg::randomMatrix(5, 5, Dom::IntegerMod(6))
```

$$\begin{pmatrix} 3 \bmod 6 & 2 \bmod 6 & 3 \bmod 6 & 5 \bmod 6 & 4 \bmod 6 \\ 2 \bmod 6 & 5 \bmod 6 & 2 \bmod 6 & 1 \bmod 6 & 1 \bmod 6 \\ 1 \bmod 6 & 3 \bmod 6 & 3 \bmod 6 & 2 \bmod 6 & 2 \bmod 6 \\ 1 \bmod 6 & 0 \bmod 6 & 3 \bmod 6 & 3 \bmod 6 & 5 \bmod 6 \\ 0 \bmod 6 & 0 \bmod 6 & 0 \bmod 6 & 1 \bmod 6 & 3 \bmod 6 \end{pmatrix}$$

The permanent of this matrix is:

```
linalg::permanent(B)
```

$$4 \bmod 6$$

Its determinant is:

```
det(B)
```

$$0 \bmod 6$$

Parameters

A

A square matrix of a domain of category `Cat::Matrix`

Return Values

Element of the component ring of A.

Algorithms

The permanent of an $n \times n$ matrix $A = (a_{i,j})_{1 \le i \le n,\ 1 \le j \le n}$ is defined similary as the determinant of A, only the signs of the permutations do not enter the definition:

$$\mathrm{perm}(A) := \sum_{\sigma \in S_n} \prod_{j=1}^{n} a_{\sigma(j),\,j}$$

.

(S_n is the symmetric group of all permutations of {1, …, n}.)

In contrast to the computation of the determinant, the computation of the permanent takes time $O(n^2\,2^n)$.

See Also

MuPAD Functions

`det`

linalg::pseudoInverse

Moore-Penrose inverse of a matrix

Syntax

```
linalg::pseudoInverse(A)
```

Description

`linalg::pseudoInverse(A)` computes the Moore-Penrose inverse of A.

If the Moore-Penrose inverse of `A` does not exist, then `FAIL` is returned.

The component ring of the matrix `A` must be a field, i.e., a domain of category `Cat::Field`.

Examples

Example 1

The Moore-Penrose inverse of the 2×3 matrix:

```
A := Dom::Matrix(Dom::Complex)([[1, I, 3], [1, 3, 2]])
```

$$\begin{pmatrix} 1 & i & 3 \\ 1 & 3 & 2 \end{pmatrix}$$

is the 3×2 matrix:

```
Astar := linalg::pseudoInverse(A)
```

$$\begin{pmatrix} \frac{7}{96} + \frac{1}{32} i & \frac{1}{24} - \frac{1}{32} i \\ -\frac{7}{32} - \frac{5}{96} i & \frac{5}{16} + \frac{7}{96} i \\ \frac{7}{24} + \frac{1}{16} i & \frac{1}{96} - \frac{3}{32} i \end{pmatrix}$$

Note that in this example, only:

```
A * Astar
```

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

yields the identity matrix, but not (see “Backgrounds” below):

```
Astar * A
```

$$\begin{pmatrix} \frac{11}{96} & \frac{3}{32} - \frac{1}{48}\,i & \frac{29}{96} + \frac{1}{32}\,i \\ \frac{3}{32} + \frac{1}{48}\,i & \frac{95}{96} & -\frac{1}{32} - \frac{1}{96}\,i \\ \frac{29}{96} - \frac{1}{32}\,i & -\frac{1}{32} + \frac{1}{96}\,i & \frac{43}{48} \end{pmatrix}$$

Parameters

A

A matrix of category `Cat::Matrix`

Return Values

Matrix of the same domain type as `A`, or the value `FAIL`.

Algorithms

For an invertible matrix A, the Moore-Penrose inverse A^* of A coincides with the inverse of A. In general, only $A\,A^*\,A = A$ and $A^*\,A\,A^* = A^*$ holds.

If A is of dimension $m \times n$, then A^* is of dimension $n \times m$.

The computation of the Moore-Penrose inverse requires the existence of a scalar product on the vector space K^n, where K is the coefficient field of the matrix A. This is only the

case for some fields K in theory, but `linalg::scalarProduct` works also for vectors over other fields (e.g. finite fields). The computation of a Moore-Penrose inverse may fail in such cases.

See Also

MuPAD Functions

`_invert`

linalg::randomMatrix

Generate a random matrix

Syntax

```
linalg::randomMatrix(m, n, <R>, <bound>, <Diagonal | Unimodular>)
```

Description

The call `linalg::randomMatrix(m, n)` returns a random $m \times n$ matrix over the default component ring for matrices, i.e., over the domain `Dom::ExpressionField()`.

The matrix components are generated by the method `"random"` of the domain `R` (see “Example 2” on page 15-185).

The parameter `bound` is given as a parameter to the method `"random"` of the domain `R` in order to bound the size of the components of the random matrix. The correct type of `bound` is determined by the method `"random"`. The parameter has no effect if the slot `"random"` does not have a size argument.

Examples

Example 1

We create a random square matrix over the integers. Because the matrix is random the created matrix can vary:

```
linalg::randomMatrix(2, 2, Dom::Integer)
```

$$\begin{pmatrix} 824 & -65 \\ -814 & -741 \end{pmatrix}$$

If you want to bound the size of its components, say between -2 and 2, enter:

```
linalg::randomMatrix(2, 2, Dom::Integer, -2..2)
```

$$\begin{pmatrix} -1 & 1 \\ -2 & 1 \end{pmatrix}$$

Example 2

The following input creates a random vector over the component ring `Dom::FloatIV` of floating-point intervals. Because the vector is random the created vector can vary:

```
v := linalg::randomMatrix(1, 4, Dom::FloatIV)
```

```
[[0.2703581654 ... 0.831037179, 0.1531565158 ... 0.9948127811, 0.1801642274 ... 0.2662729024,
  0.4520830547 ... 0.6787819565]]
```

```
domtype(v)
```

```
Dom::Matrix(Dom::FloatIV)
```

The components of this matrix are random floating-point intervals created by the `"random"` method of the domain constructor `Dom::FloatIV`.

Example 3

To create a random diagonal matrix over the rationals we enter, for example:

```
linalg::randomMatrix(3, 3, Dom::Rational, Diagonal)
```

$$\begin{pmatrix} \frac{229}{220} & 0 & 0 \\ 0 & -\frac{535}{617} & 0 \\ 0 & 0 & -\frac{245}{597} \end{pmatrix}$$

Example 4

The following command creates a random unimodular matrix over the integers so that its determinant is either 1 or -1:

```
A := linalg::randomMatrix(3, 3, Dom::Integer, Unimodular)
```

$$\begin{pmatrix} -1 & -8 & 4 \\ 4 & 0 & -5 \\ -3 & 5 & 2 \end{pmatrix}$$

```
det(A)
```

$$-1$$

We can bound the size of the components. The following input returns a unimodular matrix $A = (a_{ij})$ with $|a_{ij}| \leq 2$ for $i, j = 1, 2, 3$:

```
A := linalg::randomMatrix(3, 3, 2, Unimodular)
```

$$\begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 2 \\ 1 & 0 & 1 \end{pmatrix}$$

Since we did not specifiy the component ring, the matrix is defined over the standard component ring for matrices (the domain `Dom::ExpressionField()`):

```
domtype(A)
```

$$\text{Dom::Matrix()}$$

Parameters

`m, n`

Positive integers

`R`

The component ring, i.e., a domain of category `Cat::Rng`; default: `Dom::ExpressionField()`

`bound`

An arithmetical expression

Options

`Diagonal`

Creates a random $m \times n$ diagonal matrix over `R`.

`Unimodular`

Creates a random $m \times m$ unimodular matrix over `R`, so that its determinant is a unit in `R`.

Note: Note that this option is only available for square matrices.

The norm of each component of the matrix returned does not exceed `bound`, which must be a positive integer, if specified. The default value of `bound` is 10.

Return Values

Matrix of the domain `Dom::Matrix(R)`.

References

For generating random unimodular matrices, see Jürgen Hansen: *Generating Problems in Linear Algebra*, MapleTech, Volume 1, No.2, 1994.

See Also

MuPAD Domains
`Dom::Matrix`

MuPAD Functions
`random`

linalg::rank

Rank of a matrix

Syntax

```
linalg::rank(A)
```

```
linalg::rank(S)
```

Description

`linalg::rank(A)` computes the rank of the matrix A.

`linalg::rank(S)` computes the rank of the matrix whose columns are the vectors in S.

The row rank of a matrix is the maximal number of linearly independent row vectors of that matrix. The column rank of a matrix is the maximal number of linearly independent column vectors of that matrix. For each matrix, its row rank is equal to its column rank. This number is called the rank of a matrix.

The component ring of `A` or of the vectors given in `S` must be an integral domain (a domain of category `Cat::IntegralDomain`).

`linalg::rank` replaces symbolic elements of a matrix by random integer numbers between 1 and 10^{10}. Then the function computes the rank of the resulting numeric matrix by Gaussian elimination (see `linalg::gaussElim`). This approach introduces a tiny chance of getting a wrong result.

Note: `linalg::rank` does not simplify special functions and algebraic numbers. For some matrices, this approach leads to wrong results. See “Example 3” on page 15-189.

Examples

Example 1

Define the following matrix `A` over $\mathbb{Z}$:

```
MatZ := Dom::Matrix(Dom::Integer):
A := MatZ([[1, 2, 3, 4], [-1, 0, 1, 0], [3, 5, 6, 9]])
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ -1 & 0 & 1 & 0 \\ 3 & 5 & 6 & 9 \end{pmatrix}$$

Compute the rank of the matrix A:

```
linalg::rank(A)
```

$$3$$

Example 2

Use the three vectors $\begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ to define the columns of the matrix A. Compute the rank of A:

```
MatZ := Dom::Matrix(Dom::Integer):
S:= { MatZ([0,1,1]), MatZ([0,1,0]), MatZ([0,0,1]) }:
linalg::rank(S)
```

$$2$$

Example 3

The `linalg::rank` function does not use any simplification rules for special functions, algebraic numbers (radicals), and so on. If `linalg::rank` computes intermediate expressions that can be simplified to zero, the function can return incorrect results. For example, create the following matrices:

```
A := matrix([[exp(x + y), exp(x)], [exp(y), 1]]);
B := matrix([[sin(x)^2 + cos(x)^2, 1], [1, 1]]);
C := matrix([[sqrt(6), sqrt(2)], [sqrt(3), 1]])
```

$$\begin{pmatrix} e^{x+y} & e^{x} \\ e^{y} & 1 \end{pmatrix}$$

$$\begin{pmatrix} \cos(x)^2 + \sin(x)^2 & 1 \\ 1 & 1 \end{pmatrix}$$

$$\begin{pmatrix} \sqrt{6} & \sqrt{2} \\ \sqrt{3} & 1 \end{pmatrix}$$

There is only one independent row in each of these matrices. The rank of the matrices `A`, `B`, and `C` is 1. The `linalg::rank` function returns 2 because it does not simplify the expressions $e^{x+y} - e^x e^y$, $sin^2 + cos^2$ - 1, and $\sqrt{2}\sqrt{3} - \sqrt{6}$:

```
linalg::rank(A), linalg::rank(B), linalg::rank(C)
```

$$2, 2, 2$$

Parameters

A

A matrix of a domain of category `Cat::Matrix`

S

A list or set of column vectors of the same dimension (a column vector is an $n \times 1$ matrix of a domain of category `Cat::Matrix`)

Return Values

Nonnegative integer

See Also

MuPAD Functions

`det` | `linalg::gaussElim`

linalg::row

Extract rows of a matrix

Syntax

```
linalg::row(A, r)
```

```
linalg::row(A, r1 .. r2)
```

```
linalg::row(A, list)
```

Description

`linalg::row(A, r)` extracts the *r*-th row vector of the matrix *A*.

`linalg::row(A, r1.. r2)` returns a list of row vectors whose indices are in the range `r1.. r2`. If `r2< r1` then the empty list `[]` is returned.

`linalg::row(A, list)` returns a list of row vectors whose indices are contained in `list` (in the same order).

Examples

Example 1

We define a matrix over $\mathbb{Q}$:

```
A := Dom::Matrix(Dom::Rational)(
  [[1, 1/5], [-3/2, 5], [2, -3]]
)
```

$$\begin{pmatrix} 1 & \frac{1}{5} \\ -\frac{3}{2} & 5 \\ 2 & -3 \end{pmatrix}$$

and illustrate the three different input formats for the function `linalg::row`:

```
linalg::row(A, 2)
```

$$\left(-\frac{3}{2}\ 5\right)$$

```
linalg::row(A, [2, 1, 3])
```

$$\left[\left(-\frac{3}{2}\ 5\right), \left(1\ \frac{1}{5}\right), (2\ -3)\right]$$

```
linalg::row(A, 2..3)
```

$$\left[\left(-\frac{3}{2}\ 5\right), (2\ -3)\right]$$

Parameters

`A`

An $m \times n$ matrix of a domain of category `Cat::Matrix`

`r`

The row index: a positive integer less or equal to m

r_1 `..` r_2

A range of row indices (positive integers less or equal to m)

`list`

A list of row indices (positive integers less or equal to m)

Return Values

Single row vector or a list of row vectors; a row vector is a $1 \times n$ matrix of category `Cat::Matrix(R)`, where R is the component ring of `A`.

See Also

MuPAD Functions

`linalg::col` | `linalg::delCol` | `linalg::delRow` | `linalg::setCol` | `linalg::setRow`

linalg::scalarProduct

Scalar product of vectors

Syntax

```
linalg::scalarProduct(u, v, <Real>)
```

Description

`linalg::scalarProduct(u, v)` computes the scalar product of the vectors $\vec{u} = (u_1, \ldots, u_n)$ and $\vec{v} = (v_1, \ldots, v_n)$ with respect to the standard basis, i.e., the sum $u_1 \overline{v_1} + \ldots + u_n \overline{v_n}$.

The scalar product is also called “inner product” or “dot product”.

If the component ring of the vectors `u` and `v` does not define the entry `"conjugate"` or if the option `Real` is specified, then `linalg::scalarProduct` uses the definition $u_1 v_1 + \ldots + u_n v_n$ of the scalar product.

The vectors `u` and `v` must be defined over the same component ring.

`linalg::scalarProduct` can be redefined to a different scalar product. This also affects the behaviour of functions such as `linalg::angle`, `linalg::factorQR`, `linalg::isUnitary`, `norm` (for vectors and matrices), `linalg::orthog` and `linalg::pseudoInverse` depend on the definition of `linalg::scalarProduct`. See “Example 3” on page 15-196.

Environment Interactions

Properties of identifiers are taken into account.

Examples

Example 1

We compute the scalar product of the vectors (i, 1) and (1, - i):

```
MatC := Dom::Matrix(Dom::Complex):
u := MatC([I, 1]): v := MatC([1, -I]):
linalg::scalarProduct(u, v)
```

$$2\,i$$

Example 2

We compute the scalar product of the vectors $\vec{u} = (u_1, u_2)$ and $\vec{v} = (v_1, v_2)$ with the symbolic entries u_1, u_2, v_1, v_2 over the standard component ring for matrices:

```
delete u1, u2, v1, v2:
u := matrix([u1, u2]): v := matrix([v1, v2]):
linalg::scalarProduct(u, v)
```

$$u1\,\overline{v1} + u2\,\overline{v2}$$

You can use `assume` to tell the system that the symbolic components are to represent real numbers:

```
assume([u1, u2, v1, v2], Type::Real):
```

Then the scalar product of $\vec{u}$ and $\vec{v}$ simplifies to:

```
linalg::scalarProduct(u, v)
```

$$u1\,v1 + u2\,v2$$

Alternatively, the option `Real` can be specified:

```
unassume(u1, u2, v1, v2):
```

```
linalg::scalarProduct(u, v, Real)
```

$$u1\ v1 + u2\ v2$$

Example 3

One particular scalar product in the real vector space of continuous functions on the interval [0, 1] is defined by

$$(f, g) = \int_0^1 f(t)\, g(t)\, \mathrm{d}t$$

.

To compute an orthogonal basis corresponding to the polynomial basis 1, t, t^2, t^3, ... with respect to this scalar product, we replace the standard scalar product by the following procedure:

```
standardScalarProduct := linalg::scalarProduct:
unprotect(linalg):
linalg::scalarProduct := proc(u, v)
    local F, f, t;
begin
    // (0)
    f := expr(u[1] * v[1]);

     // (1)
    t := indets(f);
    if t = {} then t := genident("t") else t := op(t, 1) end_if;

     // (2)
    F := int(f, t = 0..1);

     // (3)
    u::dom::coeffRing::coerce(F)
end:
```

We start with step `(0)` to convert $f(t)\, g(t)$ to an expression of a basic domain type, such that the system function `int` in step `(2)` can handle its input (this is not necessary if the elements of the component ring of the vectors are already represented by elements of basic domains).

Step `(1)` extracts the indeterminate of the polynomials, step `(2)` computes the scalar product as defined above and step `(3)` converts the result back to an element of the component ring of vectors `u` and `v`.

Note that we need to unprotect the write protected identifier `linalg`, otherwise the assignment would lead to an error message.

We next create the matrix which consists of the first five of the above polynomials:

```
P := matrix([[1, t, t^2, t^3, t^4]])
```

$$\begin{pmatrix} 1 & t & t^2 & t^3 & t^4 \end{pmatrix}$$

If we now perform the Gram-Schmidt orthogonalization procedure on the columns of `P` with the function `linalg::orthog`, we get:

```
S := linalg::orthog(linalg::col(P, 1..4))
```

$$\left[(1), \left(t - \frac{1}{2}\right), \left(t^2 - t + \frac{1}{6}\right), \left(t^3 - \frac{3\,t^2}{2} + \frac{3\,t}{5} - \frac{1}{20}\right)\right]$$

Each vector in `S` is orthogonal to the other vectors in `S` with respect to the modified scalar product. We check this for the first vector:

```
linalg::scalarProduct(S[1], S[j]) $ j = 2..nops(S)
```

$$0, 0, 0$$

Finally, we undo the redefinition of the scalar product, so as not to run into trouble with subsequent computations:

```
linalg::scalarProduct := standardScalarProduct:
protect(linalg, Error):
```

Parameters

`u, v`

Vectors of the same dimension (a vector is an $n \times 1$ or $1 \times n$ matrix of a domain of category `Cat::Matrix`)

Options

`Real`

Use $u_1 v_1 + \ldots + u_n v_n$ as the definition of the scalar product, i.e., suppress the use of `conjugate`.

Return Values

Element of the component ring of `u` and `v`.

See Also

MuPAD Functions

`linalg::angle` | `linalg::crossProduct` | `linalg::factorQR` | `linalg::isUnitary` | `linalg::orthog` | `norm`

linalg::setCol

Change a column of a matrix

Syntax

```
linalg::setCol(A, p, c)
```

Description

`linalg::setCol(A, p, c)` returns a copy of matrix A with the p-th column replaced by the column vector $\vec{c}$.

If `c` is a list with at most m elements, then `c` is converted into a column vector. An error message is returned if the conversion is not possible (e.g., if an element of the list cannot be converted into an object of the component ring of `A`; see "Example 2" on page 15-200).

Examples

Example 1

We define a matrix over the rationals:

```
MatQ := Dom::Matrix(Dom::Rational):
A := MatQ([[1, 2], [3, 2]])
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 2 \end{pmatrix}$$

and replace the 2nd column by the 2×1 zero vector:

```
linalg::setCol(A, 2, MatQ([0, 0]))
```

$$\begin{pmatrix} 1 & 0 \\ 3 & 0 \end{pmatrix}$$

Example 2

We create the 2×2 zero matrix over $\mathbb{Z}_6$:

```
B := Dom::Matrix(Dom::IntegerMod(6))(2, 2)
```

$$\begin{pmatrix} 0 \bmod 6 & 0 \bmod 6 \\ 0 \bmod 6 & 0 \bmod 6 \end{pmatrix}$$

and replace the 2nd column by the vector $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$. We give the column vector in form of a list. Its elements are converted implicitly into objects of the component ring of `B`:

```
linalg::setCol(B, 2, [1, -1])
```

$$\begin{pmatrix} 0 \bmod 6 & 1 \bmod 6 \\ 0 \bmod 6 & 5 \bmod 6 \end{pmatrix}$$

The following input leads to an error message because the number `1/3` can not be converted into an object of type `Dom::IntegerMod(6)`:

```
linalg::setCol(B, 1, [1/3, 0])
```

```
Error: The column vector is invalid. [linalg::setCol]
```

Parameters

A

An $m \times n$ matrix of a domain of category `Cat::Matrix`

c

A column vector, or a list that can be converted into a column vector of the domain `Dom::Matrix(R)`, where `R` is the component ring of `A` (a column vector is an $m \times 1$ matrix)

Return Values

Matrix of the same domain type as A.

See Also

MuPAD Functions

`linalg::col` | `linalg::delCol` | `linalg::delRow` | `linalg::row` | `linalg::setRow`

linalg::setRow

Change a row of a matrix

Syntax

```
linalg::setRow(A, p, r)
```

Description

`linalg::setRow(A, p, r)` returns a copy of the matrix A with the p-th row replaced by the row vector $\vec{r}$.

If `r` is a list with at most n elements, then `r` is converted into a row vector. An error message is returned if the conversion is not possible (e.g., if an element of the list cannot be converted into an object of the component ring of `A`; see "Example 2" on page 15-203).

Examples

Example 1

We define a matrix over the rationals:

```
MatQ := Dom::Matrix(Dom::Rational):
A := MatQ([[1, 2], [3, 2]])
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 2 \end{pmatrix}$$

and replace the 2nd row by the 1×2 zero vector:

```
linalg::setRow(A, 2, MatQ(1, 2, [0, 0]))
```

$$\begin{pmatrix} 1 & 2 \\ 0 & 0 \end{pmatrix}$$

Example 2

We create the 2×4 zero matrix over $\mathbb{Z}_6$:

```
B := Dom::Matrix(Dom::IntegerMod(6))(2, 4)
```

$$\begin{pmatrix} 0 \bmod 6 & 0 \bmod 6 & 0 \bmod 6 & 0 \bmod 6 \\ 0 \bmod 6 & 0 \bmod 6 & 0 \bmod 6 & 0 \bmod 6 \end{pmatrix}$$

and replace the 2nd row by the vector (1, - 1, 1, - 1). We give the row vector in form of a list. Its elements are converted implicitly into objects of the component ring of `B`:

```
linalg::setRow(B, 2, [1, -1, 1, -1])
```

$$\begin{pmatrix} 0 \bmod 6 & 0 \bmod 6 & 0 \bmod 6 & 0 \bmod 6 \\ 1 \bmod 6 & 5 \bmod 6 & 1 \bmod 6 & 5 \bmod 6 \end{pmatrix}$$

The following input leads to an error message because the number $\frac{1}{3}$ can not be converted into an object of type `Dom::IntegerMod(6)`:

```
linalg::setRow(B, 1, [1/3, 0, 1, 0])
```

```
Error: The row vector is invalid. [linalg::setRow]
```

Parameters

`A`

An $m \times n$ matrix of a domain of category `Cat::Matrix`

`r`

A row vector or a list that can be converted into a row vector the domain `Dom::Matrix(R)`, where `R` is the component ring of `A` (a row vector is a $1 \times n$ matrix)

Return Values

Matrix of the same domain type as `A`.

See Also

MuPAD Functions

`linalg::col` | `linalg::delCol` | `linalg::delRow` | `linalg::row` | `linalg::setCol`

linalg::smithForm

Smith canonical form of a matrix

Syntax

```
linalg::smithForm(A)
```

Description

`linalg::smithForm(A)` computes the Smith canonical form of the n-dimensional square matrix A, i.e., an $n \times n$ diagonal matrix S such that $S_{i-1,i-1}$ divides $S_{i,i}$ for $i = 2, \ldots, n$.

The Smith canonical form of a matrix A is unique.

The component ring of A must be a Euclidean ring, i.e., a domain of category `Cat::EuclideanDomain`.

Examples

Example 1

We define a matrix over the integers:

```
MatZ := Dom::Matrix(Dom::Integer):
A := MatZ([[9, -36, 30], [-36, 192, -180], [30, -180, 180]])
```

$$\begin{pmatrix} 9 & -36 & 30 \\ -36 & 192 & -180 \\ 30 & -180 & 180 \end{pmatrix}$$

The Smith canonical form of A is then given by:

```
linalg::smithForm(A)
```

$$\begin{pmatrix} 3 & 0 & 0 \\ 0 & 12 & 0 \\ 0 & 0 & 60 \end{pmatrix}$$

Example 2

We compute the Smith canonical form of a matrix over a ring of polynomials:

```
MatPoly := Dom::Matrix(Dom::DistributedPolynomial([x], Dom::Rational)):
B := MatPoly(
  [[-(x - 3)^2*(x - 2),(x - 3)*(x - 2)*(x - 4)],
   [(x - 3)*(x - 2)*(x - 4),-(x - 3)^2*(x - 4)]
])
```

$$\begin{pmatrix} -x^3+8x^2-21x+18 & x^3-9x^2+26x-24 \\ x^3-9x^2+26x-24 & -x^3+10x^2-33x+36 \end{pmatrix}$$

The Smith canonical form of the matrix `B` is the following matrix:

```
linalg::smithForm(B)
```

$$\begin{pmatrix} x-3 & 0 \\ 0 & x^3-9x^2+26x-24 \end{pmatrix}$$

Parameters

A

A square matrix of a domain of category `Cat::Matrix`

Return Values

Matrix of the same domain type as `A`.

Algorithms

An $n \times n$ matrix $S = (s_{ij})$ with coefficients in a Euclidean ring is said to be in Smith canonical form if S is a diagonal matrix (with nonnegative coefficients in case of the ring $\mathbb{Z}$) such that $s_{i,\,i}$ divides $s_{i+1,\,i+1}$ for all $i < n$.

See Also

MuPAD Functions

`linalg::frobeniusForm` | `linalg::hermiteForm` | `linalg::jordanForm`

linalg::sqrtMatrix

Square root of a matrix

Syntax

```
linalg::sqrtMatrix(A, <sqrtfunc>)
```

Description

`linalg::sqrtMatrix(A)` returns the square root of the matrix A.

If A has an eigenvalue 0 of algebraic multiplicity larger than its geometric multiplicity, then the square root of A does not exist.

Examples

Example 1

A square root of a diagonal matrix is given by the diagonal matrix, whose diagonal entries are just the square roots of the original matrix.

Compute the square root of the matrix

$$A = \begin{pmatrix} 4 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & -1 \end{pmatrix}.$$

```
A := matrix([[4, 0, 0], [0, 2, 0], [0, 0, -1]]):
S := linalg::sqrtMatrix(A)
```

$$\begin{pmatrix} 2 & 0 & 0 \\ 0 & \sqrt{2} & 0 \\ 0 & 0 & i \end{pmatrix}$$

Check the correctness of the result:

```
S^2
```

$$\begin{pmatrix} 4 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

Example 2

Compute the square root of the matrix

$$A = \begin{pmatrix} 2 & -2 & 0 \\ -1 & 3 & 0 \\ -\frac{1}{3} & \frac{5}{3} & 2 \end{pmatrix}:$$

```
A := matrix([[2, -2, 0], [-1, 3, 0], [-1/3, 5/3, 2]]):
S := linalg::sqrtMatrix(A)
```

$$\begin{pmatrix} \frac{4}{3} & -\frac{2}{3} & 0 \\ -\frac{1}{3} & \frac{5}{3} & 0 \\ \frac{2\sqrt{2}}{3} - 1 & 1 - \frac{\sqrt{2}}{3} & \sqrt{2} \end{pmatrix}$$

If you compute the square of the matrix S and simplify the result, you obtain the matrix A:

```
simplify(S^2)
```

$$\begin{pmatrix} 2 & -2 & 0 \\ -1 & 3 & 0 \\ -\frac{1}{3} & \frac{5}{3} & 2 \end{pmatrix}$$

Using the function `x -> -sqrt(x)` as second argument for the computation of the square root of the matrix `A`, obtain a different matrix, whose components are just the negative components of the original square root computed above:

```
S:= linalg::sqrtMatrix(A, x -> -sqrt(x)):
S, simplify(S^2);
```

$$\begin{pmatrix} -\frac{4}{3} & \frac{2}{3} & 0 \\ \frac{1}{3} & -\frac{5}{3} & 0 \\ 1-\frac{2\sqrt{2}}{3} & \frac{\sqrt{2}}{3}-1 & -\sqrt{2} \end{pmatrix}, \begin{pmatrix} 2 & -2 & 0 \\ -1 & 3 & 0 \\ -\frac{1}{3} & \frac{5}{3} & 2 \end{pmatrix}$$

Parameters

A

A square matrix of a domain of category `Cat::Matrix`

sqrtfunc

A function satisfying $sqrtfunc(a)^2 = a$ for every element a of the coefficient ring of A (i.e. the square root function of the coefficient domain of A).

Return Values

Matrix B with $B^2 = A$ such that the eigenvalues of B are the square roots of the eigenvalues of A or `FAIL` if the square root of the matrix does not exist

See Also

MuPAD Functions

`funm` | `linalg::eigenvalues` | `linalg::eigenvectors` | `linalg::jordanForm` | `numeric::eigenvalues` | `numeric::eigenvectors` | `numeric::expMatrix` | `numeric::fMatrix` | `solve`

linalg::stackMatrix

Join matrices vertically

Syntax

```
linalg::stackMatrix(A, <B1, B2, …>)
```

Description

`linalg::stackMatrix(A, B1, B2, ...)` returns the matrix formed by joining the matrices `A, B1, B2, ...` vertically.

The matrices `B1, B2, ...` are converted into the matrix domain `Dom::Matrix(R)`, where `R` is the component ring of `A`.

An error message is raised if one of these conversions fails, or if the matrices do not have the same number of columns as the matrix `A`.

Examples

Example 1

We define a matrix:

```
A:= matrix( [[sin(x),x], [-x,cos(x)]] )
```

$$\begin{pmatrix} \sin(x) & x \\ -x & \cos(x) \end{pmatrix}$$

and append the 2×2 identity matrix to the lower end of the matrix `A`:

```
linalg::stackMatrix(A, matrix::identity(2))
```

$$\begin{pmatrix} \sin(x) & x \\ -x & \cos(x) \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Example 2

We define a matrix from the ring of 2×2 square matrices:

```
SqMatQ := Dom::SquareMatrix(2,Dom::Rational):
A := SqMatQ([[1, 2], [3, 4]])
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Note that the following operation:

```
AA := linalg::stackMatrix(A, A)
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 1 & 2 \\ 3 & 4 \end{pmatrix}$$

returns a matrix of a different domain type as the input matrix:

```
domtype(AA)
```

Dom::Matrix(Dom::Rational)

Parameters

A, <B_1, B_2, …>

Matrices of a domain of category `Cat::Matrix`

Return Values

Matrix of the domain type `Dom::Matrix(R)`, where `R` is the component ring of `A`.

See Also

MuPAD Functions

`linalg::concatMatrix`

linalg::submatrix

Extract a submatrix or a subvector from a matrix or a vector, respectively

Syntax

```
linalg::submatrix(A, r1 .. r2, c1 .. c2)

linalg::submatrix(A, rlist, clist)

linalg::submatrix(v, i1 .. i2)

linalg::submatrix(v, list)
```

Description

`linalg::submatrix(A, r1.. r2, c1.. c2)` returns a copy of the submatrix of the matrix A obtained by selecting the rows r_1, $r_1 + 1$, …, r_2 and the columns c_1, $c_1 + 1$, …, c_2.

`linalg::submatrix(v, i1.. i2)` returns a copy of the subvector of the vector $\vec{v}$ obtained by selecting the components with indices i_1, $i_1 + 1$, …, i_2.

The index notation `A[ r1.. r2, c1.. c2]` and `v[ i1.. i2]`, respectively, can be used instead of `linalg::submatrix(A, r1.. r2, c1.. c2)` and `linalg::submatrix(v, i1.. i2)`.

`linalg::submatrix(A,rlist,clist)` returns the submatrix of the matrix `A` whose (i, j)-th component is $a_{rlist_i,\, clist_j}$.

`linalg::submatrix(v,list)` returns the subvector of the vector `v` whose i-th component is v_{list_i}.

If `v` is a row vector or a column vector, then `linalg::submatrix(v, 1..1, i1.. i2)` and `linalg::submatrix(v, i1.. i2, 1..1)`, respectively, are valid inputs, and they both are equivalent to the call `linalg::submatrix(v, i1.. i2)`.

Examples

Example 1

We define the following matrix:

```
A := matrix([[1, x, 0], [0, x^2, 1]])
```

$$\begin{pmatrix} 1 & x & 0 \\ 0 & x^2 & 1 \end{pmatrix}$$

The submatrix $(a_{1,j})_{1 \le j \le 2}$ of A is given by:

```
linalg::submatrix(A, 1..1, 1..2)
```

$$\begin{pmatrix} 1 & x \end{pmatrix}$$

Equivalent to the use of the index operator we obtain:

```
A[1..1, 1..2]
```

$$\begin{pmatrix} 1 & x \end{pmatrix}$$

We extract the first and the third column of A and get the 2×2 identity matrix:

```
linalg::submatrix(A, [1, 2], [1, 3])
```

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Example 2

Vector components can be accessed by a single index or a range of indices. For example, to extract the first two components of the following vector:

```
v := matrix([1, 2, 3])
```

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

just enter the command:

```
v[1..2]
```

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

Of course, the same subvector can be extracted with the command `linalg::submatrix(v, 1..2)`.

The following input returns the vector comprising the first and the third component of `v`:

```
linalg::submatrix(v, [1, 3])
```

$$\begin{pmatrix} 1 \\ 3 \end{pmatrix}$$

Parameters

`A`

An $m \times n$ matrix of a domain of category `Cat::Matrix`

`v`

A vector with k components, i.e., a $k \times 1$ or $1 \times k$ matrix of a domain of category `Cat::Matrix`

r_1 `..` r_2, c_1 `..` c_2

Ranges of row/column indices: positive integers less or equal to m and n, respectively

`rlist`, `clist`

Lists of row/column indices: positive integers less or equal to m and n, respectively

i_1 `..` i_2

A range of vector indices: positive integers less or equal to k

`list`

A list of vector indices: positive integers less or equal to k

Return Values

Matrix of the same domain type as A or a vector of the same domain type as v, respectively.

See Also

MuPAD Functions

`linalg::col` | `linalg::row` | `linalg::substitute`

linalg::substitute

Replace a part of a matrix by another matrix

Syntax

```
linalg::substitute(B, A, m, n)
```

Description

`linalg::substitute(B, A, m, n)` returns a copy of the matrix `B`, where entries starting at position $[m, n]$ are replaced by the entries of the matrix `A`, i.e., B_{mn} is A_{11}.

If the matrices are defined over different component domains, then the entries of `A` are converted into elements of the component domain of the matrix `B`. If one of these conversions fails, then an error message is returned.

Examples

Example 1

We define the following matrix:

```
B := matrix(
  [[1, 2, 3, 4], [5, 6, 7, 8],
   [9, 10, 11, 12], [13, 14, 15, 16]]
)
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$$

and copy the 2×2 zero matrix into the matrix `B`, beginning at position [3, 3]:

```
A := matrix(2, 2):
```

```
linalg::substitute(B, A, 3, 3)
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 0 & 0 \\ 13 & 14 & 0 & 0 \end{pmatrix}$$

Matrix entries out of range are ignored:

```
linalg::substitute(B, A, 4, 4)
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 0 \end{pmatrix}$$

Parameters

A, B

Matrices of a domain of category `Cat::Matrix`

m, n

Positive integers

Return Values

Matrix of the same domain type as `B`.

See Also

MuPAD Functions

`linalg::concatMatrix` | `linalg::setCol` | `linalg::setRow` | `linalg::stackMatrix` | `linalg::submatrix`

linalg::sumBasis

Basis for the sum of vector spaces

Syntax

`linalg::sumBasis(`S_1`, `S_2`, …)`

Description

`linalg::sumBasis( `S_1`, `S_2`, ...)` returns a basis of the vector space $V_1 + V_2 + \ldots$, where V_i denotes the vector space spanned by the vectors in S_i.

To obtain an ordered basis, ``S_1`, `S_2`, ...` should be given as lists of vectors.

A basis of the zero-dimensional space is the empty set or list, respectively.

The given vectors must be defined over the same component ring, which must be a field, i.e., a domain of category `Cat::Field`.

Examples

Example 1

We define three vectors $\vec{v}_1, \vec{v}_2, \vec{v}_3$ over $\mathbb{Q}$:

```
MatQ := Dom::Matrix(Dom::Rational):
v1 := MatQ([[3, -2]]); v2 := MatQ([[1, 0]]); v3 := MatQ([[5, -3]])
```

$$\begin{pmatrix} 3 & -2 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 5 & -3 \end{pmatrix}$$

A basis of the vector space $V_1 + V_2 + V_3$ with

- V_1 generated by $\vec{v}_1, \vec{v}_2, \vec{v}_3$
- V_2 generated by $\vec{v}_1, \vec{v}_3$
- V_3 generated by $\vec{v}_1 + \vec{v}_2, \vec{v}_2, \vec{v}_1 + \vec{v}_3$

is:

```
linalg::sumBasis([v1, v2, v3], [v1, v3], [v1 + v2, v2, v1 + v3])
```

$$[(3\ \ -2),(1\ \ 0)]$$

Example 2

The following set of two vectors:

```
MatQ := Dom::Matrix(Dom::Rational):
S1 := {MatQ([1, 2, 3]), MatQ([-1, 0, 2])}
```

$$\left\{\begin{pmatrix}-1\\0\\2\end{pmatrix},\begin{pmatrix}1\\2\\3\end{pmatrix}\right\}$$

is a basis of a two-dimensional subspace of $\mathbb{Q}^3$:

```
linalg::rank(S1)
```

$$2$$

The same holds for the following set:

```
S2 := {MatQ([0, 2, 3]), MatQ([2, 4, 6])};
linalg::rank(S2)
```

$$\left\{\begin{pmatrix}0\\2\\3\end{pmatrix},\begin{pmatrix}2\\4\\6\end{pmatrix}\right\}$$

$$2$$

The sum of the corresponding two subspaces is the vector space $\mathbb{Q}^3$:

```
Q3 := linalg::sumBasis(S1, S2)
```

$$\left\{\begin{pmatrix}-1\\0\\2\end{pmatrix}, \begin{pmatrix}0\\2\\3\end{pmatrix}, \begin{pmatrix}1\\2\\3\end{pmatrix}\right\}$$

Parameters

S_1, S_2, …

A set or list of vectors of the same dimension (a vector is a $n\times1$ or $1\times n$ matrix of a domain of category `Cat::Matrix`)

Return Values

Set or a list of vectors, according to the domain type of the parameter S_1.

See Also

MuPAD Functions

`linalg::basis` | `linalg::intBasis` | `linalg::rank`

linalg::swapCol

Swap two columns in a matrix

Syntax

`linalg::swapCol(A, c1, c2)`

`linalg::swapCol(A, c1, c2, r1 .. r2)`

Description

`linalg::swapCol(A, c1, c2)` returns a copy of the matrix *A* with the columns with indices c_1 and c_2 interchanged.

The effect of `linalg::swapCol(A, c1, c2, r1.. r2)` is that only the components from row r_1 to row r_2 of column c_1 are interchanged with the corresponding components of column c_2.

Examples

Example 1

We consider the following matrix:

```
A := matrix(3, 3, (i, j) -> 3*(i - 1) + j)
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

The following command interchanges the first and the second column of A. The result is the following matrix:

```
linalg::swapCol(A, 1, 2)
```

$$\begin{pmatrix} 2 & 1 & 3 \\ 5 & 4 & 6 \\ 8 & 7 & 9 \end{pmatrix}$$

If only the components in the first two rows should be affected, we enter:

```
linalg::swapCol(A, 1, 2, 1..2)
```

$$\begin{pmatrix} 2 & 1 & 3 \\ 5 & 4 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

The third row remains unchanged.

Parameters

A

An $m \times n$ matrix of a domain of category `Cat::Matrix`

c_1, c_2

The column indices: positive integers less or equal to n

r_1 .. r_2

A range of row indices (positive integers less or equal to m)

Return Values

Matrix of the same domain type as A.

See Also

MuPAD Functions

`linalg::col` | `linalg::delCol` | `linalg::delRow` | `linalg::row` | `linalg::setCol` | `linalg::setRow` | `linalg::swapRow`

More About

- “Swap and Delete Rows and Columns”

linalg::swapRow

Swap two rows in a matrix

Syntax

`linalg::swapRow(A, r1, r2)`

`linalg::swapRow(A, r1, r2, c1 .. c2)`

Description

`linalg::swapRow(A, r1, r2)` returns a copy of the matrix *A* with the rows with indices r_1 and r_2 interchanged.

The effect of `linalg::swapRow(A, r1, r2, c1.. c2)` is that only the components from column c_1 to column c_2 of row r_1 are interchanged with the corresponding components of row r_2.

Examples

Example 1

We consider the following matrix:

```
A := matrix(3, 3, (i, j) -> 3*(i - 1) + j)
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

The following command interchanges the first and the second row of A. The result is the following matrix:

```
linalg::swapRow(A, 1, 2)
```

$$\begin{pmatrix} 4 & 5 & 6 \\ 1 & 2 & 3 \\ 7 & 8 & 9 \end{pmatrix}$$

If only the components in the first two columns should be affected, we enter:

```
linalg::swapRow(A, 1, 2, 1..2)
```

$$\begin{pmatrix} 4 & 5 & 3 \\ 1 & 2 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

The third column remains unchanged.

Parameters

A

An $m \times n$ matrix of a domain of category `Cat::Matrix`

r_1, r_2

The row indices: positive integers less or equal to m

c_1 .. c_2

A range of column indices (positive integers less or equal to n)

Return Values

Matrix of the same domain type as A.

See Also

MuPAD Functions

`linalg::col` | `linalg::delCol` | `linalg::delRow` | `linalg::row` | `linalg::setCol` | `linalg::setRow` | `linalg::swapCol`

More About

- "Swap and Delete Rows and Columns"

linalg::sylvester

Sylvester matrix of two polynomials

Syntax

```
linalg::sylvester(p, q)
```

```
linalg::sylvester(f, g, x)
```

Description

`linalg::sylvester(p, q)` returns the Sylvester matrix of the two polynomials p and q.

If no variable is specified, then the polynomials `p` and `q` must be either of the domain `DOM_POLY` or from a domain of category `Cat::Polynomial`. Polynomial expressions are not allowed.

If the polynomials `p` and `q` are of the domain `DOM_POLY`, then they must be univariate polynomials. The component ring of the Sylvester matrix is the common coefficient ring R of `p` and `q`, except in the following two cases for built-in coefficient rings: If R is `Expr` then the domain `Dom::ExpressionField()` is the component ring of the Sylvester matrix. If R is `IntMod(m)`, then the Sylvester matrix is defined over the ring `Dom::IntegerMod(m)` (see “Example 2” on page 15-230).

Otherwise, if the polynomials `p` and `q` are from a domain of category `Cat::Polynomial`, then the Sylvester matrix is computed with respect to the main variable of `p` and `q` (see the method `"mainvar"` of the category `Cat::Polynomial`). In the case of univariate polynomials the Sylvester matrix is defined over the common coefficient ring of `p` and `q`. In the case of multivariate polynomials, the Sylvester matrix is defined over the component ring `Dom::DistributedPolynomial(ind, R)`, where `ind` is the list of all variables of `p` and `q` except `x`, and `R` is the common coefficient ring of the polynomials.

If `f` and `g` are polynomial expressions or multivariate polynomials of type `DOM_POLY`, then you must specifiy the variable `x`.

In the case of polynomial expressions, the component ring of the Sylvester matrix is the domain `Dom::ExpressionField()` (see “Example 3” on page 15-231).

In the case of multivariate polynomials the Sylvester matrix is defined over the component ring `Dom::DistributedPolynomial(ind, R)`, where `ind` is the list of all variables of `f` and `g` except `x`, and `R` is the common coefficient ring of the polynomials (see "Example 4" on page 15-231).

At least one of the input polynomials must have positive degree with respect to the main variable or `x`, respectively, but it is not necessary that both of them have positive degree.

Examples

Example 1

The Sylvester matrix of the two polynomials $p = x^2 + 2x - 1$ and $q = x^4 + 1$ over $\mathbb{Z}$ is the following 6×6 matrix:

```
delete x: Z := Dom::Integer:
S := linalg::sylvester(poly(x^2 + 2*x - 1, Z), poly(x^4 + 1, Z))
```

$$\begin{pmatrix} 1 & 2 & -1 & 0 & 0 & 0 \\ 0 & 1 & 2 & -1 & 0 & 0 \\ 0 & 0 & 1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 1 & 2 & -1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Example 2

If the polynomials have the built-in coefficient ring `IntMod(m)`, then the Sylvester matrix is defined over the domain `Dom::IntegerMod(m)`:

```
delete x:
S:= linalg::sylvester(
  poly(x + 1, IntMod(7)), poly(x^2 - 2*x + 2, IntMod(7))
)
```

$$\begin{pmatrix} 1 \bmod 7 & 1 \bmod 7 & 0 \bmod 7 \\ 0 \bmod 7 & 1 \bmod 7 & 1 \bmod 7 \\ 1 \bmod 7 & 5 \bmod 7 & 2 \bmod 7 \end{pmatrix}$$

```
domtype(S)
```

```
Dom::Matrix(Dom::IntegerMod(7))
```

Example 3

The Sylvester matrix of the following two polynomial expressions with respect to the variable x is:

```
delete x, y:
S := linalg::sylvester(x + y^2, 2*x^3 - 1, x)
```

$$\begin{pmatrix} 1 & y^2 & 0 & 0 \\ 0 & 1 & y^2 & 0 \\ 0 & 0 & 1 & y^2 \\ 2 & 0 & 0 & -1 \end{pmatrix}$$

```
domtype(S)
```

```
Dom::Matrix()
```

The Sylvester matrix of these two polynomials with respect to y is the following 2×2 matrix:

```
linalg::sylvester(x + y^2, 2*x^3 - 1, y)
```

$$\begin{pmatrix} 2x^3 - 1 & 0 \\ 0 & 2x^3 - 1 \end{pmatrix}$$

Example 4

Here is an example for computing the Sylvester matrix of multivariate polynomials:

```
delete x, y: Q := Dom::Rational:
T := linalg::sylvester(poly(x^2 - x + y, Q), poly(x + 2, Q), x)
```

$$\begin{pmatrix} 1 & -1 & y \\ 1 & 2 & 0 \\ 0 & 1 & 2 \end{pmatrix}$$

```
domtype( T )
```

$$\text{Dom::Matrix}(\text{Dom::DistributedPolynomial}([y], \text{Dom::Rational}, \text{LexOrder}))$$

The Sylvester matrix of these two multivariate polynomials with respect to `y` is:

```
linalg::sylvester(poly(x^2 - x + y, Q), poly(x + 2, Q), y)
```

$$\begin{pmatrix} x+2 \end{pmatrix}$$

Parameters

`p, q`

Polynomials

`f, g`

Polynomials or polynomial expressions of positive degree

`x`

A variable

Return Values

Matrix of the domain `Dom::Matrix(R)`, where `R` is the coefficient domain of the polynomials (see below).

See Also

MuPAD Functions

`polylib::discrim` | `polylib::resultant`

linalg::tr

Trace of a matrix

Syntax

```
linalg::tr(A)
```

Description

`linalg::tr(A)` returns the trace of the square matrix A, i.e., the sum of the diagonal elements of A.

Examples

Example 1

We compute the trace of the following matrix:

```
A := Dom::Matrix(Dom::Integer)
  (3, 3, (i, j) -> 3*(i - 1) + j)
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

```
linalg::tr(A)
```

$$15$$

Parameters

A

A square matrix of a domain of category `Cat::Matrix`

Return Values

Element of the component ring of A.

See Also

MuPAD Functions
`det`

More About

- “Compute Determinants and Traces of Square Matrices”

linalg::toeplitz

Toeplitz matrix

Syntax

```
linalg::toeplitz(m, n, [t_-k, …,t_k], <R>)
linalg::toeplitz(n, [t_-k, …,t_k], <R>)
linalg::toeplitz(c, r)
linalg::toeplitz(r)
```

Description

`linalg::toeplitz(m, n, [t_-k, ..., t_-1, t_0, t_1, ..., t_k])` returns the $m \times n$ Toeplitz matrix

$$\begin{pmatrix} t_0 & t_1 & t_2 & \cdot & \cdot & \cdot & \cdot & \cdot & t_k \\ t_{-1} & t_0 & t_1 & t_2 & \cdot & \cdot & \cdot & \cdot & t_{k-1} \\ t_{-2} & t_{-1} & t_0 & t_1 & t_2 & \cdot & \cdot & \cdot & t_{k-2} \\ \cdot & & \cdot & \cdot & \cdot & & & & \cdot \\ \cdot & & & \cdot & \cdot & \cdot & & & \cdot \\ \cdot & & & & \cdot & t_0 & t_1 & t_2 & \\ \cdot & & & & \cdot & t_{-1} & t_0 & t_1 & \\ t_{-k} & \cdot & \cdot & \cdot & \cdot & \cdot & t_{-2} & t_{-1} & t_0 \end{pmatrix}.$$

`linalg::toeplitz(n, [t_-k, ..., t_k])` returns the square Toeplitz matrix of dimension $n \times n$.

A number of entries $[t_{-k}, \ldots, t_k]$ must be an odd number $2k + 1$. There must be at least k diagonal bands above the diagonal and k diagonal bands below the diagonal: k must satisfy $k \le min(m, n) - 1$. Entries with matrix indices (i, j) satisfying $|i - j| > k$ are set to 0.

Toeplitz matrices of dimension $n \times n$ can be inverted with $O(n^2)$ operations. See `linalg::toeplitzSolve`.

`linalg::toeplitz(c, r)` generates a nonsymmetric Toeplitz matrix having `c` as its first column and `r` as its first row. If the first elements of `c` and `r` differ, `toeplitz` issues a warning and uses the first element of the column.

`linalg::toeplitz(r)` generates a symmetric Toeplitz matrix if `r` is real. If `r` is complex, but its first element is real, then this syntax generates the Hermitian Toeplitz matrix formed from `r`. If the first element of `r` is complex, then the resulting matrix is Hermitian off the main diagonal, meaning that T_{ij} = conjugate(T_{ji}) for $i \neq j$.

When you use matrices in MuPAD computations, both computational efficiency and memory use can depend on whether the matrix is sparse or dense. The first two syntaxes are optimized for generating sparse matrices and, therefore, these syntaxes are preferable. For details about improving performance when working with matrices, see "Use Sparse and Dense Matrices".

Examples

Example 1

Construct a 4×4 Toeplitz matrix with three bands:

```
linalg::toeplitz(4, [-1, 2, 1])
```

$$\begin{pmatrix} 2 & 1 & 0 & 0 \\ -1 & 2 & 1 & 0 \\ 0 & -1 & 2 & 1 \\ 0 & 0 & -1 & 2 \end{pmatrix}$$

Construct a 3×5 Toeplitz matrix with symbolic entries:

```
linalg::toeplitz(3, 5, [a, b, c])
```

$$\begin{pmatrix} b & c & 0 & 0 & 0 \\ a & b & c & 0 & 0 \\ 0 & a & b & c & 0 \end{pmatrix}$$

Example 2

Construct a Toeplitz matrix by using a vector to specify its first row. For a real vector, the resulting matrix is symmetric:

```
r := matrix([1, 2, 3]):
linalg::toeplitz(r)
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 2 \\ 3 & 2 & 1 \end{pmatrix}$$

For a complex vector, the resulting matrix is Hermitian off the main diagonal:

```
r := matrix([1 + I, 2 + I, 3 + I]):
T := linalg::toeplitz(r);
htranspose(T)
```

$$\begin{pmatrix} 1+i & 2+i & 3+i \\ 2-i & 1+i & 2+i \\ 3-i & 2-i & 1+i \end{pmatrix}$$

$$\begin{pmatrix} 1-i & 2+i & 3+i \\ 2-i & 1-i & 2+i \\ 3-i & 2-i & 1-i \end{pmatrix}$$

Example 3

Construct a Toeplitz matrix by using two vectors to specify its first column and first row:

```
c := matrix([1, a/2, b/2]):
r := [1, a, b]:
linalg::toeplitz(c, r)
```

$$\begin{pmatrix} 1 & \frac{a}{2} & \frac{b}{2} \\ a & 1 & \frac{a}{2} \\ b & a & 1 \end{pmatrix}$$

If the first elements of the vectors differ, `linalg::toeplitz` issues a warning and uses the first element of the column:

```
c := matrix([1, a/2, b/2]):
r := [2, a, b]:
linalg::toeplitz(c, r)
```

```
Warning: First element of input column does not match first element of input row. Colu
```

$$\begin{pmatrix} 2 & \frac{a}{2} & \frac{b}{2} \\ a & 2 & \frac{a}{2} \\ b & a & 2 \end{pmatrix}$$

Parameters

`m, n`

Row and column dimensions of the matrix: positive integers.

$t_{-k}, \ldots, t_k$

Arithmetical expressions or elements of the component ring `R`.

`R`

Component ring: a domain of category `Cat::Rng`. The default ring is `Dom::ExpressionField()`.

`c`

Vector specifying the first column of a Toeplitz matrix.

`r`

Vector specifying the first row of a Toeplitz matrix.

Return Values

Matrix of the domain `Dom::Matrix(R)`.

See Also

MuPAD Functions

`linalg::hilbert` | `linalg::invhilbert` | `linalg::invpascal` | `linalg::invvandermonde` | `linalg::pascal` | `linalg::toeplitzSolve` | `linalg::vandermonde` | `linalg::vandermondeSolve`

linalg::toeplitzSolve

Solve a linear Toeplitz system

Syntax

```
linalg::toeplitzSolve(t, y)
```

Description

`linalg::toeplitzSolve(t, y)` returns the solution $\vec{x}$ of the linear Toeplitz system $\sum_{j=1}^{n} t_{i-j}\, x_j = y_i$ with $i = 1, \ldots, n$.

`linalg::toeplitzSolve(t, y)` with `t` = $[t_k, \ldots, t_0, \ldots, t_{-k}]$ and `y` = $[y_1, \ldots, y_n]$ solves the $n \times n$ Toeplitz system

$$\begin{pmatrix} t_0 & t_{-1} & \cdot & \cdot & \cdot & t_{-k} & & \\ t_1 & t_0 & t_{-1} & \cdot & \cdot & t_{-k} & & \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & & \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & & \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & & \\ t_k & t_{k-1} & \cdot & \cdot & \cdot & t_{-k} & & \\ & t_k & t_{k-1} & \cdot & \cdot & \cdot & t_{-k} & \\ & & t_k & t_{k-1} & \cdot & \cdot & \cdot & t_{-k} \\ & & & & \cdot & \cdot & \cdot & \cdot \\ & & & & \cdot & \cdot & \cdot & \cdot \\ & & & & \cdot & \cdot & \cdot & \cdot \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

with $2k + 1$ bands.

`linalg::toeplitzSolve` implements the Levinson algorithm. It uses $O(n^2)$ elementary operations to solve the Toeplitz system. The memory requirements are

$O(n)$. For dense Toeplitz systems, it is faster than the general solver `solve` and the linear solvers `linsolve`, `numeric::linsolve`, `linalg::matlinsolve` and `numeric::matlinsolve`.

Note: Note that the Levinson algorithm requires that all principal minors

$$t_0, \begin{pmatrix} t_0 & t_{-1} \\ t_1 & t_0 \end{pmatrix}, \begin{pmatrix} t_0 & t_{-1} & t_{-2} \\ t_1 & t_0 & t_{-1} \\ t_2 & t_1 & t_0 \end{pmatrix}, \text{etc}$$

are non-singular.

If `linalg::toeplitzSolve` does not manage to find the solution due to this limitation, or if the system is very sparse with k smaller than $\sqrt{n}$, we recommend to generate the corresponding Toeplitz matrix via `linalg::toeplitz` and compute the solution via `linalg::matlinsolve` or `numeric::matlinsolve`, respectively. Cf. “Example 2” on page 15-242

`linalg::toeplitzSolve` can solve Toeplitz systems over arbitrary coefficient rings. Just make sure that both the Toeplitz entries t as well as the components of the 'right hand side' y are elements of the desired coefficient ring. Cf. “Example 3” on page 15-243.

Examples

Example 1

The Toeplitz entries t and the right hand side y of the linear system are entered as row vectors:

```
t := matrix([4, 2, 1, 3, 5]):
y := matrix([y1, y2, y3]):
```

The solution of the Toeplitz system is returned as a vector of the same type as the input vector y:

```
x := linalg::toeplitzSolve(t, y): x, domtype(x)
```

$$\begin{pmatrix} \frac{7\,y2}{25} - \frac{y1}{5} + \frac{4\,y3}{25} \\ \frac{2\,y1}{5} - \frac{19\,y2}{25} + \frac{7\,y3}{25} \\ \frac{2\,y2}{5} - \frac{y3}{5} \end{pmatrix}, \text{Dom::Matrix}()$$

If the input vector is a list, the output is a list, too:

```
x := linalg::toeplitzSolve(t, [y1, y2, y3]): x, domtype(x)
```

$$\left[\frac{7\,y2}{25} - \frac{y1}{5} + \frac{4\,y3}{25},\ \frac{2\,y1}{5} - \frac{19\,y2}{25} + \frac{7\,y3}{25},\ \frac{2\,y2}{5} - \frac{y3}{5}\right], \text{DOM\_LIST}$$

```
delete t, y, x:
```

Example 2

The Levinson algorithm cannot solve the following Toeplitz system because the first principal minor of the Toeplitz matrix (the central element of the Toeplitz entries) vanishes:

```
linalg::toeplitzSolve([1, 0, 1], [y1, y2, y3, y4])
```

FAIL

This does not necessarily imply that the Toeplitz system is not solvable. We generate the corresponding Toeplitz matrix and use a generic linear solver such as `linalg::matlinsolve`:

```
T := linalg::toeplitz(4, 4, [1, 0, 1])
```

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

```
linalg::matlinsolve(T, matrix([y1, y2, y3, y4]))
```

$$\begin{pmatrix} y2 - y4 \\ y1 \\ y4 \\ y3 - y1 \end{pmatrix}$$

Example 3

We solve a Toeplitz system over the field $\mathbb{Z}_7$ (the integers modulo 7) represented by the domain `Dom::IntegerMod(7)`:

```
R := Dom::IntegerMod(7):
t := [R(5), R(3), R(2), R(5), R(1)]:
y := [R(1), R(2), R(3)]:
linalg::toeplitzSolve(t, y)
```

$$[1 \bmod 7,\ 5 \bmod 7,\ 2 \bmod 7]$$

```
delete R, t, y:
```

Parameters

`t`

A vector or a list with $2k + 1$ elements. (A vector is a $(2k + 1)\times 1$ or a $1 \times (2k + 1)$ matrix of category `Cat::Matrix`).

`y`

A vector or a list with n elements

Return Values

Vector or list with n elements of the same domain type as the elements of `y`. `FAIL` is returned if the algorithm does not succeed in finding a solution.

See Also

MuPAD Functions

`linalg::hilbert` | `linalg::invhilbert` | `linalg::invpascal` | `linalg::invvandermonde` | `linalg::matlinsolve` | `linalg::pascal` | `linalg::toeplitz` | `linalg::vandermonde` | `linalg::vandermondeSolve` | `linsolve` | `numeric::linsolve` | `numeric::matlinsolve` | `solve`

linalg::transpose

Transpose of a matrix

Syntax

```
linalg::transpose(A)
```

Description

`linalg::transpose(A)` returns the transpose A^t of the matrix A.

`linalg::transpose` is an interface function for the method `"transpose"` of the matrix domain of `A`, i.e., instead of `linalg::transpose(A)` one may call `A::dom::transpose(A)` directly.

Examples

Example 1

We define a 3×4 matrix:

```
A := matrix([[1, 2, 3, 4], [-1, 0, 1, 0], [3, 5, 6, 9]])
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ -1 & 0 & 1 & 0 \\ 3 & 5 & 6 & 9 \end{pmatrix}$$

Then the transpose of `A` is the 4 ×3 matrix:

```
linalg::transpose(A)
```

$$\begin{pmatrix} 1 & -1 & 3 \\ 2 & 0 & 5 \\ 3 & 1 & 6 \\ 4 & 0 & 9 \end{pmatrix}$$

Parameters

A

A matrix of a domain of category `Cat::Matrix`

Return Values

Matrix of the same domain type as A.

Algorithms

Let $A = (a_{i,j})_{1 \le i \le m,\ 1 \le j \le n}$ be an $m \times n$ matrix. Then the transpose of A is the $n \times m$ matrix:

$$A^t = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{m,1} \\ a_{1,2} & a_{2,2} & \cdots & a_{m,2} \\ \vdots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots \\ a_{1,n} & a_{1,n} & \cdots & a_{m,n} \end{pmatrix}.$$

linalg::vandermonde

Vandermonde matrices and their inverses

Syntax

```
linalg::vandermonde([v1, v2, …], <R>)
```

Description

`linalg::vandermonde( v1, v2, ... , vn)` returns the $n \times n$ Vandermonde matrix V with entries $V_{ij} = v_i^{j-1}$.

Use `linalg::vandermonde([ v1, ..., vn], R)` to define the $n \times n$ Vandermonde matrix over the field `R`. Note that the Vandermonde nodes v_i must be elements of `R` or must be convertible to elements of `R`. The same holds true for the inverse Vandermonde matrix.

Vandermonde matrices of dimension $n \times n$ can be inverted with $O(n^2)$ operations. Linear equations with a Vandermonde coefficient matrix can be solved via `linalg::vandermondeSolve`.

Examples

Example 1

We construct a 3×3 Vandermonde matrix:

```
V := linalg::vandermonde([v1, v2, v3])
```

$$\begin{pmatrix} 1 & v1 & v1^2 \\ 1 & v2 & v2^2 \\ 1 & v3 & v3^2 \end{pmatrix}$$

The inverse of this matrix is:

```
linalg::invvandermonde([v1, v2, v3])
```

$$\begin{pmatrix} \frac{v2\,v3}{\sigma_3} & -\frac{v1\,v3}{\sigma_2} & \frac{v1\,v2}{\sigma_1} \\ -\frac{v2+v3}{\sigma_3} & \frac{v1+v3}{\sigma_2} & -\frac{v1+v2}{\sigma_1} \\ \frac{1}{\sigma_3} & -\frac{1}{\sigma_2} & \frac{1}{\sigma_1} \end{pmatrix}$$

where

$$\sigma_1 = (v1 - v3)\,(v2 - v3)$$

$$\sigma_2 = (v1 - v2)\,(v2 - v3)$$

$$\sigma_3 = (v1 - v2)\,(v1 - v3)$$

`V` and its inverse are matrices of the domain `Dom::Matrix()`. One can specify a special component ring for the matrices, provided the nodes can be converted to elements of the ring. For example, specification of the domain `Dom::Float` generates floating-point entries:

```
V := linalg::vandermonde([2, PI, 1/3], Dom::Float)
```

$$\begin{pmatrix} 1.0 & 2.0 & 4.0 \\ 1.0 & 3.141592654 & 9.869604401 \\ 1.0 & 0.3333333333 & 0.1111111111 \end{pmatrix}$$

```
domtype(V)
```

Dom::Matrix(Dom::Float)

It is faster to compute the inverse via `linalg::invvandermonde` than inverting the Vandermonde matrix by a generic inversion algorithm (as implemented by `V^(-1)`):

```
V^(-1) = linalg::invvandermonde([2, PI, 1/3], Dom::Float)
```

$$\begin{pmatrix} -0.5503876788 & 0.2079506905 & 1.342436988 \\ 1.826356876 & -0.7278274166 & -1.098529459 \\ -0.5255815182 & 0.3119260357 & 0.2136554825 \end{pmatrix} =$$

$$\begin{pmatrix} -0.5503876788 & 0.2079506905 & 1.342436988 \\ 1.826356876 & -0.7278274166 & -1.098529459 \\ -0.5255815182 & 0.3119260357 & 0.2136554825 \end{pmatrix}$$

```
delete V:
```

Parameters

v_1, v_2, …

The Vandermonde nodes: arithmetical expressions

R

The component ring: a domain of category `Cat::Rng`; default: `Dom::ExpressionField()`

Return Values

$n \times n$ matrix of the domain `Dom::Matrix(R)`.

Algorithms

Vandermonde matrices are notoriously ill-conditioned. The inverses of large floating-point Vandermonde matrices are subject to severe round-off effects.

See Also

MuPAD Functions

`linalg::hilbert` | `linalg::invhilbert` | `linalg::invpascal` | `linalg::invvandermonde` | `linalg::pascal` | `linalg::toeplitz` | `linalg::toeplitzSolve` | `linalg::vandermondeSolve`

linalg::invvandermonde

Vandermonde matrices and their inverses

Syntax

```
linalg::invvandermonde([v1, v2, …], <R>)
```

Description

`linalg::invvandermonde( v1, v2, ... , vn)` returns the inverse of the Vandermonde matrix with nodes v_i.

Use `linalg::vandermonde([ v1, ..., vn], R)` to define the $n \times n$ Vandermonde matrix over the field `R`. Note that the Vandermonde nodes v_i must be elements of `R` or must be convertible to elements of `R`. The same holds true for the inverse Vandermonde matrix.

Vandermonde matrices of dimension $n \times n$ can be inverted with $O(n^2)$ operations. Linear equations with a Vandermonde coefficient matrix can be solved via `linalg::vandermondeSolve`.

Examples

Example 1

We construct a 3×3 Vandermonde matrix:

```
V := linalg::vandermonde([v1, v2, v3])
```

$$\begin{pmatrix} 1 & v1 & v1^2 \\ 1 & v2 & v2^2 \\ 1 & v3 & v3^2 \end{pmatrix}$$

The inverse of this matrix is:

```
linalg::invvandermonde([v1, v2, v3])
```

$$\begin{pmatrix} \frac{v2\,v3}{\sigma_3} & -\frac{v1\,v3}{\sigma_2} & \frac{v1\,v2}{\sigma_1} \\ -\frac{v2+v3}{\sigma_3} & \frac{v1+v3}{\sigma_2} & -\frac{v1+v2}{\sigma_1} \\ \frac{1}{\sigma_3} & -\frac{1}{\sigma_2} & \frac{1}{\sigma_1} \end{pmatrix}$$

where

$$\sigma_1 = (v1 - v3)\,(v2 - v3)$$

$$\sigma_2 = (v1 - v2)\,(v2 - v3)$$

$$\sigma_3 = (v1 - v2)\,(v1 - v3)$$

`V` and its inverse are matrices of the domain `Dom::Matrix()`. One can specify a special component ring for the matrices, provided the nodes can be converted to elements of the ring. For example, specification of the domain `Dom::Float` generates floating-point entries:

```
V := linalg::vandermonde([2, PI, 1/3], Dom::Float)
```

$$\begin{pmatrix} 1.0 & 2.0 & 4.0 \\ 1.0 & 3.141592654 & 9.869604401 \\ 1.0 & 0.3333333333 & 0.1111111111 \end{pmatrix}$$

```
domtype(V)
```

Dom::Matrix(Dom::Float)

It is faster to compute the inverse via `linalg::invvandermonde` than inverting the Vandermonde matrix by a generic inversion algorithm (as implemented by `V^(-1)`):

```
V^(-1) = linalg::invvandermonde([2, PI, 1/3], Dom::Float)
```

$$\begin{pmatrix} -0.5503876788 & 0.2079506905 & 1.342436988 \\ 1.826356876 & -0.7278274166 & -1.098529459 \\ -0.5255815182 & 0.3119260357 & 0.2136554825 \end{pmatrix} =$$

$$\begin{pmatrix} -0.5503876788 & 0.2079506905 & 1.342436988 \\ 1.826356876 & -0.7278274166 & -1.098529459 \\ -0.5255815182 & 0.3119260357 & 0.2136554825 \end{pmatrix}$$

```
delete V:
```

Parameters

v_1, v_2, …

The Vandermonde nodes: arithmetical expressions

R

The component ring: a domain of category `Cat::Rng`; default: `Dom::ExpressionField()`

Return Values

$n \times n$ matrix of the domain `Dom::Matrix(R)`.

Algorithms

Vandermonde matrices are notoriously ill-conditioned. The inverses of large floating-point Vandermonde matrices are subject to severe round-off effects.

See Also

MuPAD Functions

`linalg::hilbert` | `linalg::invhilbert` | `linalg::invpascal` | `linalg::pascal` | `linalg::toeplitz` | `linalg::toeplitzSolve` | `linalg::vandermonde` | `linalg::vandermondeSolve`

linalg::vandermondeSolve

Solve a linear Vandermonde system

Syntax

```
linalg::vandermondeSolve(v, y, <Transposed>)
```

Description

`linalg::vandermondeSolve(v, y)` returns the solution $\vec{x}$ of the linear Vandermonde system $\sum_{j=1}^{n} v_i^{j-1} x_j = y_i$ with $i = 1, \ldots, n$.

`linalg::vandermondeSolve` uses $O(n^2)$ elementary operations to solve the Vandermonde system. It is faster than the general solver `solve` and the linear solvers `linsolve`, `numeric::linsolve`, `linalg::matlinsolve` and `numeric::matlinsolve`.

The solution $\vec{x} = (x_1, \ldots, x_n)$ returned by `linalg::vandermondeSolve([v[i] $ i=1..n], [y[i] $ i=1..n])` yields the coefficients of the polynomial $p(v) = x_1 + x_2\, v + \cdots + x_n\, v^{n-1}$ interpolating the data table $(v_1, y_1), \ldots, (v_n, y_n)$, i.e.,

$$p(v_1) = y_1, \ldots, p(v_n) = y_n .$$

See “Example 1” on page 15-255.

Examples

Example 1

The Vandermonde points v and the right hand side y of the linear system are entered as vectors:

```
delete y0, y1, y2:
v := matrix([[0, 1, 2]]);
y := matrix([[y0, y1, y2]])
```

$$\begin{pmatrix} 0 & 1 & 2 \end{pmatrix}$$

$$\begin{pmatrix} y0 & y1 & y2 \end{pmatrix}$$

The solution vector is:

```
x := linalg::vandermondeSolve(v, y)
```

$$\left(y0 \quad 2\,y1 - \frac{3\,y0}{2} - \frac{y2}{2} \quad \frac{y0}{2} - y1 + \frac{y2}{2} \right)$$

The solution yields the coefficients of the interpolating polynomial:

```
P := v -> _plus(x[i+1]*v^i $ i=0..2):
```

through the points $(0, y_0)$, $(1, y_1)$, $(2, y_2)$:

```
P(v[1]), P(v[2]), P(v[3])
```

$$y0,\ y1,\ y2$$

With the optional argument `Transposed`, the linear system with the transposed Vandermonde matrix corresponding to `v` is solved:

```
linalg::vandermondeSolve(v, y, Transposed)
```

$$\left(y0 - \frac{3\,y1}{2} + \frac{y2}{2} \quad 2\,y1 - y2 \quad \frac{y2}{2} - \frac{y1}{2} \right)$$

```
delete v, y, x, P:
```

Example 2

The Vandermonde points v and the right hand side y of the linear system are entered as 2×1 matrices:

```
Mat := Dom::Matrix(Dom::ExpressionField(normal)):

delete v1, v2, y1, y2:
v := Mat([v1, v2]):
y := Mat([y1, y2]):
```

We define the vectors over the domain `Dom::ExpressionField(normal)` in order to simplify intermediate computations.

Next, we compute the solution of the corresponding Vandermonde system:

```
x := linalg::vandermondeSolve(v, y)
```

$$\begin{pmatrix} \frac{v1\,y2 - v2\,y1}{v1 - v2} \\ \frac{y1 - y2}{v1 - v2} \end{pmatrix}$$

We construct the Vandermonde matrix `V` and verify the result:

```
V := Mat([[1, v[1]], [1, v[2]]])
```

$$\begin{pmatrix} 1 & v1 \\ 1 & v2 \end{pmatrix}$$

```
V * x
```

$$\begin{pmatrix} y1 \\ y2 \end{pmatrix}$$

```
delete Mat, v, y, x, V:
```

Example 3

We solve a Vandermonde system over the field $\mathbb{Z}_7$ (the integers modulo 7) represented by the domain `Dom::IntegerMod(7)`:

```
MatZ7 := Dom::Matrix(Dom::IntegerMod(7)):
v := MatZ7([1, 2, 3]): y := MatZ7([0, 1, 2]):
```

```
linalg::vandermondeSolve(v, y)
```

$$\begin{pmatrix} 6 \bmod 7 \\ 1 \bmod 7 \\ 0 \bmod 7 \end{pmatrix}$$

```
delete MatZ7, v, y:
```

Parameters

v

A vector with distinct elements (a vector is an $n \times 1$ or $1 \times n$ matrix of category `Cat::Matrix`). Alternatively, a list with n distinct elements is also accepted.

y

A vector of the same dimension and domain type as `v`. Alternatively, a list with n elements is also accepted.

Options

`Transposed`

Returns the solution $\vec{x}$ of the transposed system $\sum_{j=1}^{n} v_j^{\,i-1} x_j = y_i$ with $i = 1, \ldots, n$.

Return Values

Vector of the same domain type as `y`.

Algorithms

The Vandermonde matrix

$$V = \begin{pmatrix} 1 & v_1 & v_1^2 & \dots & v_1^{n-1} \\ 1 & v_2 & v_2^2 & \dots & v_2^{n-1} \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ 1 & v_n & v_n^2 & \dots & v_n^{n-1} \end{pmatrix}$$

generated by $v = [v_1, \ldots, v_n]$ is invertible if and only if the v_i are distinct.

The vector $\vec{x}$ returned by `linalg::vandermondeSolve(x, y)` is the unique solution of $V\,\vec{x} = \vec{y}$.

The vector x returned by `linalg::vandermondeSolve(x, y, Transposed)` is the unique solution of $V^t\,\vec{x} = \vec{y}$.

See Also

MuPAD Functions

`interpolate` | `linalg::invhilbert` | `linalg::invpascal` | `linalg::invvandermonde` | `linalg::matlinsolve` | `linalg::pascal` | `linalg::toeplitz` | `linalg::toeplitzSolve` | `linalg::vandermonde` | `linalg::vandermondeSolve` | `linsolve` | `numeric::linsolve` | `numeric::matlinsolve` | `solve`

linalg::vecdim

Number of components of a vector

Syntax

```
linalg::vecdim(v)
```

Description

`linalg::vecdim(v)` returns the number of elements of the vector $\vec{v}$.

Examples

Example 1

We define a column vector with two elements and a row vector with four elements:

```
v1 := matrix([1, 0]); v2 := matrix([[1, 2, 3, 4]])
```

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$(1\ 2\ 3\ 4)$$

`linalg::vecdim` gives us the number of elements, i.e., the dimension of these vectors:

```
linalg::vecdim(v1), linalg::vecdim(v2)
```

$$2, 4$$

In contrast, the function `linalg::matdim` returns the number of rows and columns of these vectors:

```
linalg::matdim(v1), linalg::matdim(v2)
```

```
[2, 1], [1, 4]
```

Parameters

v

A vector, i.e., an $n \times 1$ or $1 \times n$ matrix of a domain of category `Cat::Matrix`

Return Values

Positive integer.

See Also

MuPAD Functions

`linalg::matdim` | `linalg::ncols` | `linalg::nrows`

linalg::vectorOf

Type specifier for vectors

Syntax

```
linalg::vectorOf(R)
```

```
linalg::vectorOf(R, n)
```

Description

`linalg::vectorOf(R)` is a type specifier representing all objects of a domain of category `Cat::Matrix` with component ring `R` and number of rows or number of columns equal to one.

`linalg::vectorOf(R,n)` is a type specifier representing all objects of a domain of category `Cat::Matrix` with component ring `R` and number of rows equal to `n` and number of columns equal to one, or vice versa.

`linalg::vectorOf(Type::AnyType,n)` is a type specifier representing all objects of a domain of category `Cat::Matrix` with an arbitrary component ring `R` and number of rows equal to `n` and number of columns equal to one, or vice versa.

Examples

Example 1

`linalg::vectorOf` can be used together with `testtype` to check whether a MuPAD object is a vector:

```
MatZ := Dom::Matrix(Dom::Integer):
v := MatZ([1, 0, -1])
```

$$\begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}$$

The following yields FALSE because v is 3-dimensional vector:

```
testtype(v, linalg::vectorOf(Dom::Integer, 4))
```

```
    FALSE
```

The following yields FALSE because v is defined over the integers:

```
testtype(v, linalg::vectorOf(Dom::Rational))
```

```
    FALSE
```

Of course, v can be converted into a vector over the rationals, as shown by the following call:

```
bool(coerce(v, Dom::Matrix(Dom::Rational)) <> FAIL)
```

```
    TRUE
```

This shows that `testtype` in conjunction with `linalg::vectorOf(R)` does not check whether an object can be converted into a vector over the specified component ring R. It checks only if the object is a vector whose component ring is R.

The following test returns TRUE because v is a 3-dimensional vector:

```
testtype(v, linalg::vectorOf(Type::AnyType, 3))
```

```
    TRUE
```

Example 2

`linalg::vectorOf` can also be used for checking parameters of procedures. The following procedure computes the orthogonal complement of a 2-dimensional vector:

```
orth := proc(v:linalg::vectorOf(Type::AnyType, 2))
begin
    [v[1], v[2]] := [-v[2],v[1]];
    return(v)
end:
```

```
u := matrix([[1, 2]]); u_ := orth(u)
```

$$\begin{pmatrix} 1 & 2 \end{pmatrix}$$

$$\begin{pmatrix} -2 & 1 \end{pmatrix}$$

Calling the procedure `orth` with an invalid parameter leads to an error message:

```
orth([1, 2])
```

```
Error: The object '[1, 2]' is incorrect. The type of argument number 1 must be 'slot(Ty
  Evaluating: orth
```

Parameters

R

The component ring: a library domain

n

A positive integer

Return Values

Type expression of the domain type Type.

See Also

MuPAD Functions

`testtype`

linalg::wiedemann

Solving linear systems by Wiedemann's algorithm

Syntax

```
linalg::wiedemann(A, b, <mult>, <prob>)
```

Description

`linalg::wiedemann(A, b, mult, ...)` tries to find a vector $\vec{x}$ that satisfies the equation $A\,\vec{x} = \vec{b}$ by using Wiedemann's algorithm.

The parameter `mult` must be a function such that the result of `mult(A,y)` equals $A\,\vec{y}$ for every n-dimensional column vector $\vec{y}$. The parameter `y` is of the same domain type as `A`. The argument `mult` does not need to handle other types of parameters, nor does it need to handle other matrices than `A`.

`linalg::wiedemann` uses a probabilistic algorithm. For a deterministic variant enter `FALSE` for the optional parameter `prob`.

If the system $A\,\vec{x} = \vec{b}$ does not have a solution, then `linalg::wiedemann` returns `FAIL`.

If the system $A\,\vec{x} = \vec{b}$ has more than one solution, then a random one is returned.

Due to the probabilistic nature of Wiedemann's algorithm, the computation may fail with small probability. In this case `FAIL` is returned. If the deterministic variant is chosen, then the algorithm may be slower for a small number of matrices.

The vector `b` must be defined over the component ring of `A`.

The coefficient ring of `A` must be a field, i.e., a domain of category `Cat::Field`.

It is recommended to use `linalg::wiedemann` only if `mult` uses significantly less than $O(n^2)$ field operations.

Examples

Example 1

We define a matrix and a column vector over the finite field with 29 elements:

```
MatZ29 := Dom::Matrix(Dom::IntegerMod(29)):
A := MatZ29([[1, 2, 3], [4, 7, 8], [9, 12, 17]]);
b := MatZ29([1, 2, 3])
```

$$\begin{pmatrix} 1 \bmod 29 & 2 \bmod 29 & 3 \bmod 29 \\ 4 \bmod 29 & 7 \bmod 29 & 8 \bmod 29 \\ 9 \bmod 29 & 12 \bmod 29 & 17 \bmod 29 \end{pmatrix}$$

$$\begin{pmatrix} 1 \bmod 29 \\ 2 \bmod 29 \\ 3 \bmod 29 \end{pmatrix}$$

Since `A` does not have a special form that would allow a fast matrix-vector multiplication, we simply use `_mult`. Wiedemann's algorithm works in this case, although it is less efficient than Gaussian elimination:

```
linalg::wiedemann(A, b, _mult)
```

$$\left[\begin{pmatrix} 24 \bmod 29 \\ 21 \bmod 29 \\ 17 \bmod 29 \end{pmatrix}, \text{TRUE} \right]$$

Example 2

Now let us define another matrix that has a special form:

```
MatZ29 := Dom::Matrix(Dom::IntegerMod(29)):
A := MatZ29([[1, 0, 0], [0, 1, 2], [0, 0, 1]]);
```

```
b := MatZ29(3, 1, [1, 2, 3]):
```

$$\begin{pmatrix} 1 \bmod 29 & 0 \bmod 29 & 0 \bmod 29 \\ 0 \bmod 29 & 1 \bmod 29 & 2 \bmod 29 \\ 0 \bmod 29 & 0 \bmod 29 & 1 \bmod 29 \end{pmatrix}$$

For this particular matrix, it is easy to define an efficient multiplication method:

```
mult := proc(dummy, y)
begin
    y[2]:=y[2]+2*y[3];
    y
end:
linalg::wiedemann(A, b, mult)
```

$$\left[\begin{pmatrix} 1 \bmod 29 \\ 25 \bmod 29 \\ 3 \bmod 29 \end{pmatrix}, \text{TRUE} \right]$$

Parameters

A

An $n \times n$ matrix of a domain of category `Cat::Matrix`

b

An n-dimensional column vector, i.e., an $n \times 1$ matrix of a domain of category `Cat::Matrix`

mult

A matrix-vector multiplication method: function or functional expression; default: `_mult`

prob

TRUE or FALSE (default: TRUE)

Return Values

Either the list `[x, TRUE]` if a solution for the system $A\,\vec{x} = \vec{b}$ has been found, or the list `[x, FALSE]` if a non-zero solution for the corresponding homogeneous system $A\,\vec{x} = \vec{0}$ has been found, or the value `FAIL` (see below).

Algorithms

The expected running time for the probabilistic algorithm is $O(n^2 + n\,M)$, and the running time for the deterministic variant is $O(n^2\,M)$ in the worst case, but only $O(n^2 + n\,M)$ on average. Here, M is the number of field operations that the matrix-vector multiplication routine `mult` uses.

The basic idea of the algorithm is to solve a linear system $A\,\vec{x} = \vec{b}$ by finding the minimal polynomial $f(y)$ that solves $f(A)\,\vec{b} = \vec{0}$. If the constant coefficient $c = f(0)$ is non-zero and $g(y) := f(y) - c$, the equality $g(A)\,\vec{b} = -c\,\vec{b}$ implies that $\vec{x} = -\frac{1}{c}\left(\frac{g}{y}\right)(A)$ is the solution.

The polynomial f is found by looking for the minimal polynomial h satisfying $\vec{u}\,h(A)\,\vec{b} = \vec{0}$ for some randomly chosen row vector $\vec{u}$. This may yield $h \neq f$ in unlucky cases, but in general the probability for this is small.

References

[1] Douglas Wiedemann: “Solving Sparse Linear equations over Finite Fields”, IEEE Transactions on Information Theory, vol. 32, no.1, Jan. 1986.

See Also

MuPAD Functions

`linalg::matlinsolve` | `linalg::vandermondeSolve`

linopt – Linear Optimization

linopt::corners
linopt::maximize
linopt::minimize
linopt::plot_data
linopt::Transparent
linopt::Transparent::autostep
linopt::Transparent::clean_basis
linopt::Transparent::convert
linopt::Transparent::dual_prices
linopt::Transparent::phaseI_tableau
linopt::Transparent::phaseII_tableau
linopt::Transparent::result
linopt::Transparent::simplex
linopt::Transparent::suggest
linopt::Transparent::userstep

linopt::corners

Return the feasible corners of a linear program

Syntax

```
linopt::corners([constr, obj], vars, <All>, <Logic>)

linopt::corners([constr, obj, <NonNegative>, <seti>], vars, <All>, <Logic>)

linopt::corners([constr, obj, <NonNegative>, <All>], vars, <All>, <Logic>)

linopt::corners([constr, obj, <setn>, <seti>], vars, <All>, <Logic>)

linopt::corners([constr, obj, <setn>, <All>], vars, <All>, <Logic>)
```

Description

`linopt::corners([constr, obj], vars)` returns all valid corners of the linear program.

`linopt::corners([constr, obj], vars, All)` returns all corners of the linear program.

[`constr`, `obj`] is a linear program of the same structure like in `linopt::maximize`. The second parameter `vars` specifies the order in which the components of the corners found are printed; if e.g. {x=1, y=2} is a corner and [x,y] was entered, [1,2] will be returned.

As options, for finding the corners, one may use `All` and/or `Logic`. `All` causes the output of non-feasible corners, too, `Logic` allows the algorithm to search for corners in planes like x=0, too, although $x \geq 0$ is not part of the input. This guarantees that for all non-empty feasible regions a corner will be found.

As the result of `linopt::corners` a triple consisting of the set of corners, the maximal objective function value found and the corner associated to it is returned. If there is no feasible corner found, only the empty set is returned.

Examples

Example 1

We compute all valid corners of a small linear program:

```
k := [{4 <= 2*x + 2*y, -2 <= 4*y - 2*x, -8 <= y - 2*x,
       y - 2*x <= -2, y <= 6}, x + y]:
linopt::corners(k, [x, y])
```

$$\left[\left\{\left[\frac{4}{3}, \frac{2}{3}\right], \left[\frac{5}{3}, \frac{1}{3}\right], [4, 6], [5, 2], [7, 6]\right\}, 13, [7, 6]\right]$$

Now we compute all corners, also the ones which are not valid. We see that we now get e.g. also the corner which is given by the cut of $-2x + 4y = 2$ and $-2x + y \leq -2$. Here we see that the invalid corner (13,6) has the maximal objective function value 19:

```
k := [{4 <= 2*x + 2*y, -2 <= 4*y - 2*x, -8 <= y - 2*x,
       y - 2*x <= -2, y <= 6}, x + y]:
linopt::corners(k, [x, y], All)
```

$$\left[\left\{[-4, 6], [1, 0], \left[\frac{4}{3}, \frac{2}{3}\right], \left[\frac{5}{3}, \frac{1}{3}\right], \left[\frac{10}{3}, -\frac{4}{3}\right], [4, 6], [5, 2], [7, 6], [13, 6]\right\}, 19, [13, 6]\right]$$

```
delete k:
```

Example 2

As one can see the linear program given by the constraints $x + y \geq -1$ and $x + y \leq 3$ and the linear objective function $x + 2y$ has no corners:

```
l := [{-1 <= x + y, x + y <= 3}, x + 2*y]:
linopt::corners(l,[x,y]), linopt::corners(l,[x,y], All)
```

$$\varnothing, \varnothing$$

If one also assumes a cut with a coordinate plane as a corner, some corners exist. One can use `linopt::plot_data` to visualize this problem:

```
linopt::corners(l, [x,y], Logic)
```

```
[{[-1, 0], [0, 0], [0, -1], [0, 3], [3, 0]}, 6, [0, 3]]
```

```
delete l:
```

Parameters

`constr`

A set or list of linear constraints

`obj`

A linear expression

`seti`

A set which contains identifiers interpreted as indeterminants

`setn`

A set which contains identifiers interpreted as indeterminants

`vars`

A list containing the variables of the linear program described by `constr` and `obj` and the existing options

Options

`All`

This option can appear at two different places in the call of `linopt::corners`. If it is part of the linear program it means that all variables are constrained to be integers. If it appears as the third or forth argument of the call it means that all corners, not only the valid ones should be computed by `linopt::corners`.

`NonNegative`

All variables are constrained to be nonnegative

Logic

This allows the algorithm to search for corners in planes like x=0 too, although $x \geq 0$ is not part of the linear program.

Return Values

Set or a list with 3 elements.

References

Papadimitriou, Christos H; Steiglitz, Kenneth: Combinatorial Optimization; Algorithms and Complexity. Prentice-Hall, 1982.

Nemhauser, George L; Wolsey, Laurence A: Integer and Combinatorial Optimization. New York, Wiley, 1988.

Salkin, Harvey M; Mathur, Kamlesh: Foundations of Integer Programming. North-Holland, 1989.

Neumann, Klaus; Morlock, Martin: Operations-Research. Munich, Hanser, 1993.

Duerr, Walter; Kleibohm, Klaus: Operations Research; Lineare Modelle und ihre Anwendungen. Munich, Hanser, 1992.

Suhl, Uwe H: MOPS - Mathematical OPtimization System. European Journal of Operational Research 72(1994)312-322. North-Holland, 1994.

Suhl, Uwe H; Szymanski, Ralf: Supernode Processing of Mixed Integer Models. Boston, Kluwer Academic Publishers, 1994.

See Also

MuPAD Functions

`linopt::maximize` | `linopt::minimize` | `linopt::plot_data`

linopt::maximize

Maximize a linear or mixed-integer program

Syntax

```
linopt::maximize([constr, obj], <DualPrices>)

linopt::maximize([constr, obj, <NonNegative>, <seti>])

linopt::maximize([constr, obj, <NonNegative>, <All>])

linopt::maximize([constr, obj, <setn>, <seti>])

linopt::maximize([constr, obj, <setn>, <All>])

linopt::maximize([constr, obj, <NonNegative>], DualPrices)

linopt::maximize([constr, obj, <set>], DualPrices)
```

Description

`linopt::maximize([constr, obj])` returns the solution of the linear or mixed-integer program given by the constraints `constr` and the linear objective function `obj` which should be maximized.

The expression `obj` is the linear objective function to be maximized subject to the linear constraints `constr`. The function `linopt::maximize` returns a triple consisting of the state of the output, `OPTIMAL`, `EMPTY` or `UNBOUNDED`, a set of equations which describes the optimal solution of the specified linear program, which is empty or depends on a free variable Φ subject to the state, and finally the maximal objective function value, which can be either a number, `-infinity` or a linear function in Φ.

The states `OPTIMAL`, `EMPTY` or `UNBOUNDED` have the following meanings. `OPTIMAL` means an optimal solution for the linear program was found. If the state is `EMPTY` no optimal solution was found and if it is `UNBOUNDED` then the solution has no upper bound.

If the option `NonNegative` is used all variables are constrained to be nonnegative. If instead of `NonNegative` a set `setn` is given then only the variables from `setn` are constrained to be nonnegative.

If the option `All` is used all variables are constrained to be integers. If instead of `All` a set `seti` is given, then only the variables from `seti` are constrained to be integers.

As a second parameter for `linopt::maximize` the option `DualPrices` is provided for the linear case (the first parameter therefore must not have more than three elements). This option causes the output of the dual-prices in addition to the solution-tripel. In this case the result of `linopt::maximize` is a sequence of a list containing the solution-tripel and a set containing the dual prices. See “Example 4” on page 16-8.

Examples

Example 1

We try to solve the linear program

$$2\,c_1 \le 1$$

$$2\,c_2 \le 1$$

with the linear objective function $c_1 + 2\,c_2$:

```
linopt::maximize([{2*c1 <= 1, 2*c2 <= 1}, c1 + 2*c2])
```

$$\left[\text{OPTIMAL},\ \left\{c1 = \frac{1}{2},\ c2 = \frac{1}{2}\right\},\ \frac{3}{2}\right]$$

Example 2

Now let's have a look at the linear program

$$\begin{aligned} 3\,x + 4\,y - 3\,z &\le 23 \\ 5\,x - 4\,y - 3\,z &\le 10 \\ 7\,x + 4\,y + 11\,z &\le 30 \end{aligned}$$

with the linear objective function $-x + y + 2\,z$. If we make no restriction to the variables the result is unbounded:

```
c := [{3*x + 4*y - 3*z <= 23, 5*x - 4*y - 3*z <= 10,
       7*x + 4*y + 11*z <= 30}, -x + y + 2*z]:
linopt::maximize(c)
```

$$\left[\text{UNBOUNDED}, \left\{x = -\text{PHI1}, y = 0, z = \frac{7\,\text{PHI1}}{11} + \frac{30}{11}\right\}, \frac{25\,\text{PHI1}}{11} + \frac{60}{11}\right]$$

But if all variables are constrained to be nonnegative, we get a result. That's also the case if only x and y are constrained to be nonnegative:

```
linopt::maximize(append(c, NonNegative));
linopt::maximize(append(c, {x, y}))
```

$$\left[\text{OPTIMAL}, \left\{x = 0, y = \frac{49}{8}, z = \frac{1}{2}\right\}, \frac{57}{8}\right]$$

$$\left[\text{OPTIMAL}, \left\{x = 0, y = \frac{49}{8}, z = \frac{1}{2}\right\}, \frac{57}{8}\right]$$

```
delete c:
```

Example 3

The following linear program do not have a solution:

```
linopt::maximize([{x <= -1, x >= 0}, x])
```

$$[\text{EMPTY}, \varnothing, -\infty]$$

Example 4

The output of the dual prices can be enforced with the option `DualPrices`:

```
linopt::maximize([{2*c1 <= 1, 2*c2 <= 1},c1 + 2*c2],
                 DualPrices)
```

$$\left[\text{OPTIMAL}, \left\{\text{c1} = \frac{1}{2}, \text{c2} = \frac{1}{2}\right\}, \frac{3}{2}\right], \left\{\left[\text{c1} \le \frac{1}{2}, 1\right], \left[\text{c2} \le \frac{1}{2}, 2\right]\right\}$$

Example 5

We have a look at the knapsack problem with *x1*, *x2*, *x3*, *x4* ∈ {0, 1}:

```
c := {20*x1 + 15*x2 + 20*x3 + 5*x4 <= 25}:
c := c union {x1 <= 1, x2 <= 1, x3 <= 1, x4 <= 1}:
f := 10*x1 + 15*x2 + 16*x3 + x4:
linopt::maximize([c, f, NonNegative, All])
```

$$[\text{OPTIMAL}, \{x1 = 0, x2 = 0, x3 = 1, x4 = 1\}, 17]$$

```
delete c, f:
```

Parameters

`constr`

A set or list of linear constraints

`obj`

A linear expression

`seti`

A set which contains identifiers interpreted as indeterminates

`setn`

A set which contains identifiers interpreted as indeterminates

Options

`All`

All variables are constrained to be integers

`NonNegative`

All variables are constrained to be nonnegative

`DualPrices`

This option is only available in the linear case. It causes the output of the dual-prices in addition to the solution-triple.

Return Values

List or a sequence of a list and a set containing the solution of the linear or mixed-integer program.

References

Papadimitriou, Christos H; Steiglitz, Kenneth: Combinatorial Optimization; Algorithms and Complexity. Prentice-Hall, 1982.

Nemhauser, George L; Wolsey, Laurence A: Integer and Combinatorial Optimization. New York, Wiley, 1988.

Salkin, Harvey M; Mathur, Kamlesh: Foundations of Integer Programming. North-Holland, 1989.

Neumann, Klaus; Morlock, Martin: Operations-Research. Munich, Hanser, 1993.

Duerr, Walter; Kleibohm, Klaus: Operations Research; Lineare Modelle und ihre Anwendungen. Munich, Hanser, 1992.

Suhl, Uwe H: MOPS - Mathematical OPtimization System. European Journal of Operational Research 72(1994)312-322. North-Holland, 1994.

Suhl, Uwe H; Szymanski, Ralf: Supernode Processing of Mixed Integer Models. Boston, Kluwer Academic Publishers, 1994.

See Also

MuPAD Functions

`linopt::corners` | `linopt::minimize` | `linopt::plot_data`

linopt::minimize

Minimize a linear or mixed-integer program

Syntax

```
linopt::minimize([constr, obj], <DualPrices>)

linopt::minimize([constr, obj, <NonNegative>, <seti>])

linopt::minimize([constr, obj, <NonNegative>, <All>])

linopt::minimize([constr, obj, <setn>, <seti>])

linopt::minimize([constr, obj, <setn>, <All>])

linopt::minimize([constr, obj, <NonNegative>], DualPrices)

linopt::minimize([constr, obj, <set>], DualPrices)
```

Description

`linopt::minimize([constr, obj])` returns the solution of the linear or mixed-integer program given by the constraints `constr` and the linear objective function `obj` which should be minimized.

The expression `obj` is the linear objective function to be minimized subject to the linear constraints `constr`. The function `linopt::minimize` returns a triple consisting of the state of the output, `OPTIMAL`, `EMPTY` or `UNBOUNDED`, a set of equations which describes the optimal solution of the specified linear program, which is empty or depends on a free variable Φ subject to the state, and finally the minimal objective function value, which can be either a number, `infinity` or a linear function in Φ.

The states `OPTIMAL`, `EMPTY` or `UNBOUNDED` have the following meanings. `OPTIMAL` means an optimal solution for the linear program was found. If the state is `EMPTY` no optimal solution was found and if it is `UNBOUNDED` then the solution has no upper bound.

If the option `NonNegative` is used all variables are constrained to be nonnegative. If instead of `NonNegative` a set `setn` is given then only the variables from `setn` are constrained to be nonnegative.

If the option `All` is used all variables are constrained to be integers. If instead of `All` a set `seti` is given, then only the variables from `seti` are constrained to be integers.

As a second parameter for `linopt::minimize` the option `DualPrices` is provided for the linear case (the first parameter therefore must not have more than three elements). This option causes the output of the dual-prices in addition to the solution-tripel. In this case the result of `linopt::minimize` is a sequence of a list containing the solution-tripel and a set containing the dual prices. See “Example 4” on page 16-13.

Examples

Example 1

We try to solve the linear program

$$\begin{aligned} c_1 + c_2 &\le 3 \\ c_2 &\le 9 \end{aligned}$$

with the linear objective function - c_1 - c_2:

```
linopt::minimize([{c1 + c2 <= 3, c2 <= 9}, -c1 - c2])
```

$$[\text{OPTIMAL}, \{c1 = 0, c2 = 3\}, -3]$$

Example 2

Now let's have a look at the linear program

$$\begin{aligned} 3x + 4y - 3z &\le 23 \\ 5x - 4y - 3z &\le 10 \\ 7x + 4y + 11z &\le 30 \end{aligned}$$

with the linear objective function - $x + y + 2z$. If we make no restriction to the variables the result is unbounded:

```
c := [{3*x + 4*y - 3*z <= 23, 5*x - 4*y - 3*z <= 10,
       7*x + 4*y + 11*z <= 30}, -x + y + 2*z]:
linopt::minimize(c)
```

$$\left[\text{UNBOUNDED}, \left\{y = -\text{PHI1}, x = -4\,\text{PHI1} - \frac{13}{2}, z = -\frac{16\,\text{PHI1}}{3} - \frac{85}{6}\right\}, -\frac{23\,\text{PHI1}}{3} - \frac{131}{6}\right]$$

But if all variables are constrained to be nonnegative, we get a result. That's also the case if only x and y are constrained to be nonnegative:

```
linopt::minimize(append(c, NonNegative));
linopt::minimize(append(c, {x, y}))
```

$$[\text{OPTIMAL}, \{x = 2, y = 0, z = 0\}, -2]$$

$$\left[\text{OPTIMAL}, \left\{x = 0, y = \frac{13}{8}, z = -\frac{11}{2}\right\}, -\frac{75}{8}\right]$$

```
delete c:
```

Example 3

The following linear program does not have a solution:

```
linopt::minimize([{x <= -1, x >= 0}, x])
```

$$[\text{EMPTY}, \varnothing, \infty]$$

Example 4

The output of the dual prices can be enforced with the option `DualPrices`:

```
linopt::minimize([{c1 + c2 <= 3, c2 <= 9}, -c1 - c2],
                 DualPrices)
```

$$[\text{OPTIMAL}, \{c1 = 0, c2 = 3\}, -3], \{[c1 + c2 \le 3, 1], [c2 \le 9, 0]\}$$

Parameters

`constr`

A set or list of linear constraints

`obj`

A linear expression

`seti`

S set which contains identifiers interpreted as indeterminates

`setn`

A set which contains identifiers interpreted as indeterminates

Options

`All`

All variables are constrained to be integer

`NonNegative`

All variables are constrained to be nonnegative

`DualPrices`

This option is only available in the linear case. It causes the output of the dual-prices in addition to the solution-tripel.

Return Values

List or a sequence of a list and a set containing the solution of the linear or mixed-integer program.

References

Papadimitriou, Christos H; Steiglitz, Kenneth: Combinatorial Optimization; Algorithms and Complexity. Prentice-Hall, 1982.

Nemhauser, George L; Wolsey, Laurence A: Integer and Combinatorial Optimization. New York, Wiley, 1988.

Salkin, Harvey M; Mathur, Kamlesh: Foundations of Integer Programming. North-Holland, 1989.

Neumann, Klaus; Morlock, Martin: Operations-Research. Munich, Hanser, 1993.

Duerr, Walter; Kleibohm, Klaus: Operations Research; Lineare Modelle und ihre Anwendungen. Munich, Hanser, 1992.

Suhl, Uwe H: MOPS - Mathematical OPtimization System. European Journal of Operational Research 72(1994)312-322. North-Holland, 1994.

Suhl, Uwe H; Szymanski, Ralf: Supernode Processing of Mixed Integer Models. Boston, Kluwer Academic Publishers, 1994.

See Also

MuPAD Functions

`linopt::corners` | `linopt::maximize` | `linopt::plot_data`

linopt::plot_data

Plot the feasible region of a linear program

Syntax

```
linopt::plot_data([constr, obj, <NonNegative>, <seti>], vars)
```

```
linopt::plot_data([constr, obj, <NonNegative>, <All>], vars)
```

```
linopt::plot_data([constr, obj, <setn>, <seti>], vars)
```

```
linopt::plot_data([constr, obj, <setn>, <All>], vars)
```

Description

`linopt::plot_data([constr, obj], vars)` returns a graphical description of the feasible region of the linear program [constr, obj], and the line vertical to the objective function vector through the corner with the maximal objective function value found.

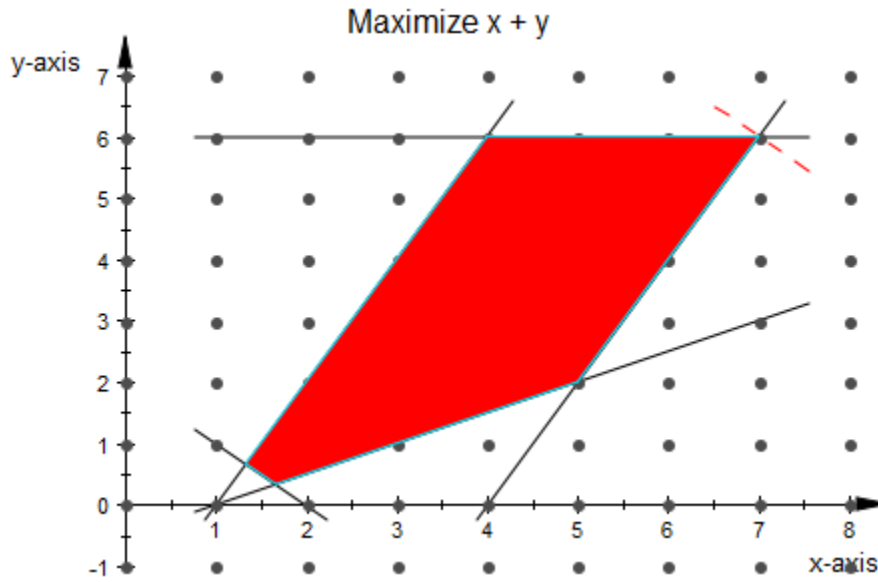
[`constr`, `obj`] is a linear program with exactly two variables. The problem has the same structure like in `linopt::maximize`. The second parameter `vars` specifies which variable belongs to the horizontal and vertical axis.

Examples

Example 1

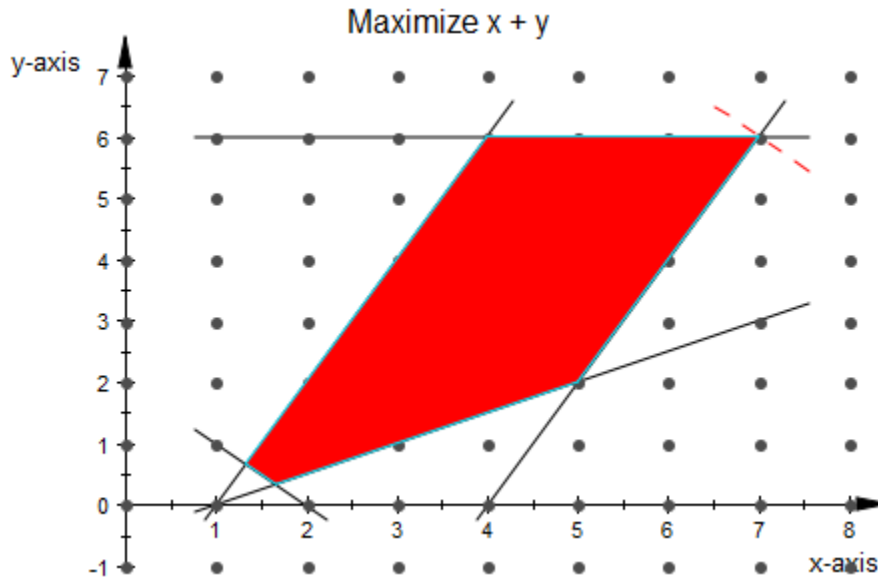
We plot the feasible region of the given linear program. Here the valid corners of the linear program are easy to see:

```
k := [{2*x + 2*y >= 4, -2*x + 4*y >= -2, -2*x + y >= -8,
       -2*x + y <= -2, y <= 6}, x + y, NonNegative]:
g := linopt::plot_data(k, [x, y]):
plot(g):
```



In this example there is no difference if the Option `NonNegative` is given for the linear program or not:

```
k := [{2*x + 2*y >= 4, -2*x + 4*y >= -2, -2*x + y >= -8,
       -2*x + y <= -2, y <= 6}, x + y]:
g := linopt::plot_data(k, [x, y]):
plot(g):
```

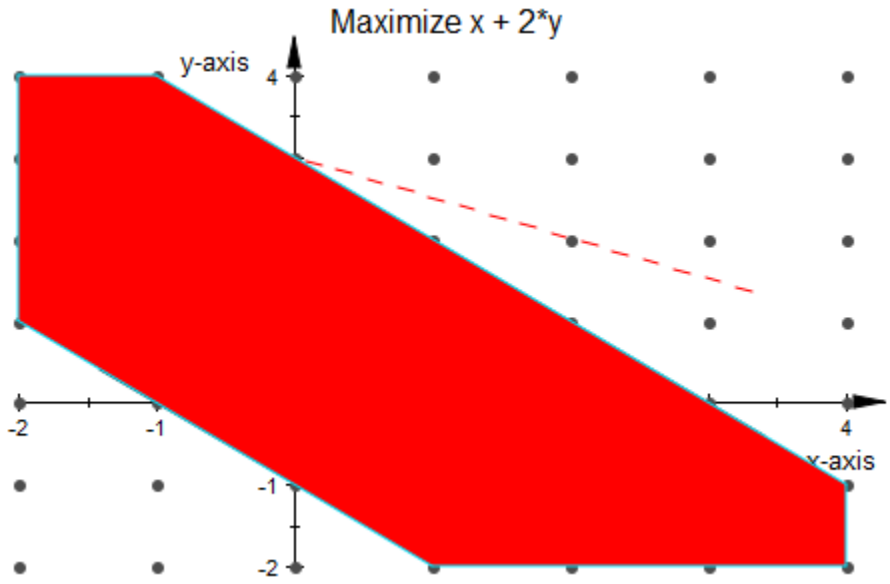



```
delete k, g:
```

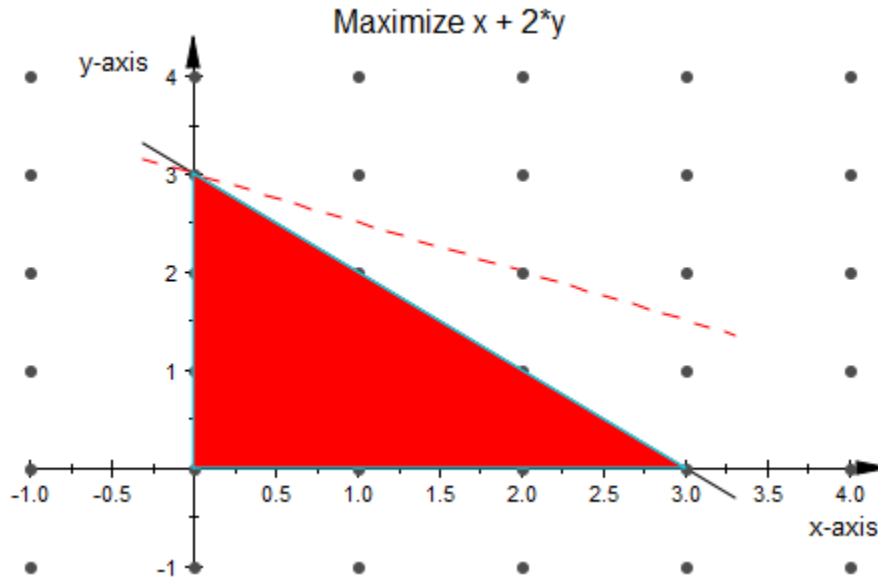
Example 2

Now we give an example where one can see a difference if the variables are constrained to be nonnegative:

```
k := [{x + y >= -1, x + y <= 3}, x + 2*y]:
g := linopt::plot_data(k, [x, y]):
plot(g):
```




```
k := [{x + y >= -1, x + y <= 3}, x + 2*y, NonNegative]:
g := linopt::plot_data(k, [x, y]):
plot(g):
```




```
delete k, g:
```

Parameters

`constr`

A set or list of linear constraints

`obj`

A linear expression

`seti`

A set which contains identifiers interpreted as indeterminates

`setn`

A set which contains identifiers interpreted as indeterminates

vars

A list containing the two variables of the linear program described by `constr` and `obj` and the existing options

Options

All

All variables are constrained to be integer

NonNegative

All variables are constrained to be nonnegative

Return Values

Expression describing a graphical object which can be used by `plot`.

References

Papadimitriou, Christos H; Steiglitz, Kenneth: Combinatorial Optimization; Algorithms and Complexity. Prentice-Hall, 1982.

Nemhauser, George L; Wolsey, Laurence A: Integer and Combinatorial Optimization. New York, Wiley, 1988.

Salkin, Harvey M; Mathur, Kamlesh: Foundations of Integer Programming. North-Holland, 1989.

Neumann, Klaus; Morlock, Martin: Operations-Research. Munich, Hanser, 1993.

Duerr, Walter; Kleibohm, Klaus: Operations Research; Lineare Modelle und ihre Anwendungen. Munich, Hanser, 1992.

Suhl, Uwe H: MOPS - Mathematical OPtimization System. European Journal of Operational Research 72(1994)312-322. North-Holland, 1994.

Suhl, Uwe H; Szymanski, Ralf: Supernode Processing of Mixed Integer Models. Boston, Kluwer Academic Publishers, 1994.

See Also

MuPAD Functions

`linopt::corners` | `linopt::maximize` | `linopt::minimize`

linopt::Transparent

Return the ordinary simplex tableau of a linear program

Syntax

```
linopt::Transparent([constr, obj, <NonNegative>, <seti>])

linopt::Transparent([constr, obj, <NonNegative>, <All>])

linopt::Transparent([constr, obj, <setn>, <seti>])

linopt::Transparent([constr, obj, <setn>, <All>])
```

Description

`linopt::Transparent([constr, obj])` returns the ordinary simplex tableau of the given linear program given by the constraints `constr` and the linear objective function `obj`.

[`constr`, `obj`] is a Linear Optimization Problem of the same structure like in `linopt::maximize`. As the result the ordinary simplex tableau of the given problem is returned; this means that equations will be replaced by two unequations and unbounded variables will be replaced by two new variables.

Internally the tableau returned consists of more information than viewable on the screen. Therefore `linopt::Transparent::convert` is provided to perform the transduction into the structure of the screen-tableau. (This can be necessary if the returned tableau shall serve as an input-parameter for another function — e.g. a user defined procedure for the selection of pivot elements.) If an ordinary simplex with two phases is wished, the next step should be the call of `linopt::Transparent::phaseI_tableau`.

All functions of the linopt library using the tableau returned by `linopt::Transparent` try to minimize the problem! Therefore it can be necessary to multiply the objective function with -1 first.

In the simplex tableau returned a special notation is used. "linopt" stands for the tableau them self, "obj" describes the linear objective function, "restr" stands for the vector of restrictions, slk[1], slk[2], ... are the slack variables and the names of the other variables

stand for themselves. Variables which are given as row labels indicate that they are part of the base.

Examples

Example 1

First a small example, returning the ordinary simplex tableau of the given linear program. One can see that the slack variables are forming the basis:

```
k := [{x + y >= -1, x + y <= 3}, x + 2*y, NonNegative]:
linopt::Transparent(k)
```

"linopt"	"restr"	slk_1	slk_2	y	x
"obj"	0	0	0	2	1
slk_1	1	1	0	−1	−1
slk_2	3	0	1	1	1

It follows a little bit larger example:

```
k := [{3*x + 4*y - 3*z <= 23, 5*x - 4*y - 3*z <= 10,
       7*x + 4*y + 11*z <= 30}, -x + y + 2*z, NonNegative]:
linopt::Transparent(k)
```

"linopt"	"restr"	slk_1	slk_2	slk_3	y	x	z
"obj"	0	0	0	0	1	−1	2
slk_1	23	1	0	0	4	3	−3
slk_2	10	0	1	0	−4	5	−3
slk_3	30	0	0	1	4	7	11

The result of `linopt::Transparent` is of domain type `linopt::Transparent`. So it can be used as input for other `linopt::Transparent::*` function, e.g. for `linopt::Transparent::suggest`:

```
k := [{x + y >= -1, x + y <= 3}, x + 2*y, NonNegative]:
t := linopt::Transparent(k):
domtype(t), linopt::Transparent::suggest(t)
```

linopt::Transparent, OPTIMAL

```
delete k, t:
```

Parameters

`constr`

A set or list of linear constraints

`obj`

A linear expression

`seti`

A set which contains identifiers interpreted as indeterminates

`setn`

A set which contains identifiers interpreted as indeterminates

Options

`All`

All variables are constrained to be integer

`NonNegative`

All variables are constrained to be nonnegative

Return Values

Simplex tableau of domain type `linopt::Transparent`.

References

Papadimitriou, Christos H; Steiglitz, Kenneth: Combinatorial Optimization; Algorithms and Complexity. Prentice-Hall, 1982.

Nemhauser, George L; Wolsey, Laurence A: Integer and Combinatorial Optimization. New York, Wiley, 1988.

Salkin, Harvey M; Mathur, Kamlesh: Foundations of Integer Programming. North-Holland, 1989.

Neumann, Klaus; Morlock, Martin: Operations-Research. Munich, Hanser, 1993.

Duerr, Walter; Kleibohm, Klaus: Operations Research; Lineare Modelle und ihre Anwendungen. Munich, Hanser, 1992.

Suhl, Uwe H: MOPS - Mathematical OPtimization System. European Journal of Operational Research 72(1994)312-322. North-Holland, 1994.

Suhl, Uwe H; Szymanski, Ralf: Supernode Processing of Mixed Integer Models. Boston, Kluwer Academic Publishers, 1994.

See Also

MuPAD Functions

`linopt::Transparent::autostep` | `linopt::Transparent::convert` | `linopt::Transparent::dual_prices` | `linopt::Transparent::phaseI_tableau` | `linopt::Transparent::phaseII_tableau` | `linopt::Transparent::result` | `linopt::Transparent::simplex` | `linopt::Transparent::suggest` | `linopt::Transparent::userstep`

linopt::Transparent::autostep

Perform the next simplex step

Syntax

```
linopt::Transparent::autostep(tableau)
```

Description

`linopt::Transparent::autostep(tableau)` performs the next step of the simplex algorithm. This is the same step that `linopt::Transparent::suggest` would suggest for the given simplex tableau `tableau`.

Normally `linopt::Transparent::autostep` returns the next simplex tableau. If the calculation of the simplex algorithm is finished `linopt::Transparent::autostep` returns a set containing a solution of the given linear program described by `tableau`.

Examples

Example 1

The ordinary simplex tableau of a given linear program is created:

```
k := [{x + y >= 2}, x, NonNegative]:
t := linopt::Transparent(k)
```

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_1 & y & x \\ \text{"obj"} & 0 & 0 & 0 & 1 \\ \text{slk}_1 & -2 & 1 & -1 & -1 \end{pmatrix}$$

The next two steps of the simplex algorithm are executed for the given simplex tableau:

```
linopt::Transparent::autostep(t);
```

```
linopt::Transparent::autostep(%)
```

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_1 & y & x \\ \text{"obj"} & -2 & 1 & -1 & 0 \\ x & 2 & -1 & 1 & 1 \end{pmatrix}$$

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_1 & y & x \\ \text{"obj"} & 0 & 0 & 0 & 1 \\ y & 2 & -1 & 1 & 1 \end{pmatrix}$$

```
delete k, t:
```

Example 2

The ordinary simplex tableau of a given linear program is created:

```
k := [{x + y >= -1, x + y <= 3}, x + 2*y, NonNegative]:
t := linopt::Transparent(k)
```

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_1 & \text{slk}_2 & y & x \\ \text{"obj"} & 0 & 0 & 0 & 2 & 1 \\ \text{slk}_1 & 1 & 1 & 0 & -1 & -1 \\ \text{slk}_2 & 3 & 0 & 1 & 1 & 1 \end{pmatrix}$$

If the end of the simplex algorithm is reached, `linopt::Transparent::autostep` returns a solution of the given linear program:

```
linopt::Transparent::suggest(t),
linopt::Transparent::autostep(t)
```

$$\text{OPTIMAL}, \{x = 0, y = 0\}$$

```
delete k, t:
```

Parameters

`tableau`

A simplex tableau of domain type `linopt::Transparent`

Return Values

Simplex tableau of domain type `linopt::Transparent` or a set which contains the solution of the linear program.

References

Papadimitriou, Christos H; Steiglitz, Kenneth: Combinatorial Optimization; Algorithms and Complexity. Prentice-Hall, 1982.

Nemhauser, George L; Wolsey, Laurence A: Integer and Combinatorial Optimization. New York, Wiley, 1988.

Salkin, Harvey M; Mathur, Kamlesh: Foundations of Integer Programming. North-Holland, 1989.

Neumann, Klaus; Morlock, Martin: Operations-Research. Munich, Hanser, 1993.

Duerr, Walter; Kleibohm, Klaus: Operations Research; Lineare Modelle und ihre Anwendungen. Munich, Hanser, 1992.

Suhl, Uwe H: MOPS - Mathematical OPtimization System. European Journal of Operational Research 72(1994)312-322. North-Holland, 1994.

Suhl, Uwe H; Szymanski, Ralf: Supernode Processing of Mixed Integer Models. Boston, Kluwer Academic Publishers, 1994.

See Also

MuPAD Functions

`linopt::Transparent` | `linopt::Transparent::convert` | `linopt::Transparent::dual_prices` |

```
linopt::Transparent::phaseI_tableau | linopt::Transparent::result
| linopt::Transparent::simplex | linopt::Transparent::suggest |
linopt::Transparent::userstep
```

linopt::Transparent::clean_basis

Delete all slack variables of the first phase from the basis

Syntax

```
linopt::Transparent::clean_basis(tableau)
```

Description

`linopt::Transparent::clean_basis(tableau)` removes the additional slack variables of the phase one of the simplex algorithm from the optimal basic (described by `tableau`) calculated by `linopt::Transparent::phaseI_tableau` and `linopt::Transparent::simplex`.

At the end of the phase one of the 2-phase simplex algorithm, explicitly started by using `linopt::Transparent::phaseI_tableau`, it is necessary to eliminate all artificial variables from the optimal basis before phase two can be started by using `linopt::Transparent::phaseII_tableau`. `linopt::Transparent::clean_basis` performs some pivot steps until all phase one slack variables do not occur in the basis any longer.

Examples

Example 1

In this example we first compute an optimal basis for the first phase of the simplex algorithm:

```
t := linopt::Transparent([{x <= 1,y <= 1,x + y >= 2},
                          0,NonNegative]):
t := linopt::Transparent::phaseI_tableau(t):
t := linopt::Transparent::simplex(t)
```

"linopt"	"restr"	slk_4	slk_5	slk_6	slk_1	slk_2	slk_3	y	x
"obj"	0	2	2	0	1	1	1	0	0
x	1	1	0	0	1	0	0	0	1
y	1	0	1	0	0	1	0	1	0
slk_6	0	−1	−1	1	−1	−1	−1	0	0

As we can see the artificial slack variable slk[6] is an element of the optimal basis. An error message is returned if we apply `linopt::Transparent::phaseII_tableau` or `linopt::Transparent::simplex` to this simplex tableau:

```
linopt::Transparent::phaseII_tableau(t);
```

```
Error: Clean the basis from phase I slack variables first. [linopt::Transparent::phase
```

So we have to use `linopt::Transparent::clean_basis` before continuing with the appropriate function:

```
t := linopt::Transparent::clean_basis(t);
linopt::Transparent::phaseII_tableau(t)
```

"linopt"	"restr"	slk_4	slk_5	slk_6	slk_1	slk_2	slk_3	y	x
"obj"	0	1	1	1	0	0	0	0	0
x	1	0	−1	1	0	−1	−1	0	1
y	1	0	1	0	0	1	0	1	0
slk_1	0	1	1	−1	1	1	1	0	0

"linopt"	"restr"	slk_1	slk_2	slk_3	y	x
"obj"	0	0	0	0	0	0
x	1	0	−1	−1	0	1
y	1	0	1	0	1	0
slk_1	0	1	1	1	0	0

```
delete t:
```

Parameters

`tableau`

A simplex tableau of domain type `linopt::Transparent`

Return Values

Simplex tableau of domain type `linopt::Transparent`.

References

Papadimitriou, Christos H; Steiglitz, Kenneth: Combinatorial Optimization; Algorithms and Complexity. Prentice-Hall, 1982.

Nemhauser, George L; Wolsey, Laurence A: Integer and Combinatorial Optimization. New York, Wiley, 1988.

Salkin, Harvey M; Mathur, Kamlesh: Foundations of Integer Programming. North-Holland, 1989.

Neumann, Klaus; Morlock, Martin: Operations-Research. Munich, Hanser, 1993.

Duerr, Walter; Kleibohm, Klaus: Operations Research; Lineare Modelle und ihre Anwendungen. Munich, Hanser, 1992.

Suhl, Uwe H: MOPS - Mathematical OPtimization System. European Journal of Operational Research 72(1994)312-322. North-Holland, 1994.

Suhl, Uwe H; Szymanski, Ralf: Supernode Processing of Mixed Integer Models. Boston, Kluwer Academic Publishers, 1994.

See Also

MuPAD Functions

`linopt::Transparent` | `linopt::Transparent::autostep` | `linopt::Transparent::convert` | `linopt::Transparent::dual_prices` |

```
linopt::Transparent::phaseII_tableau | linopt::Transparent::result
| linopt::Transparent::simplex | linopt::Transparent::suggest |
linopt::Transparent::userstep
```

linopt::Transparent::convert

Transform the given tableau into a structure printable on screen

Syntax

```
linopt::Transparent::convert(tableau)
```

Description

`linopt::Transparent::convert` converts `tableau` into a two dimensional array which corresponds with the screen-tableau. One can now access the element in the *i*-th row and *j*-th column of the simplex tableau by accessing the corresponding element of the array.

Internally the given `tableau` of domain type `linopt::Transparent` contains a lot of more information than the simplex tableau which is printed by some functions of the `linopt` library, e.g. `linopt::Transparent::simplex`, and which is visible on the screen. Furthermore it is not possible to access the element in the *i*-th row and *j*-th column of `tableau` to get the corresponding element from the simplex tableau which is visible on the screen.

While the internal structure of `tableau` is not known the structure of the two dimensional array is well defined. So it can be easily used in own procedures. See “Example 2” on page 16-36.

Examples

Example 1

We convert a simplex tableau of domain type `linopt::Transparent` into a two dimensional array:

```
k := [{x + y >= 2}, x, NonNegative]:
t := linopt::Transparent(k):
a := linopt::Transparent::convert(t):
t, domtype(t);
```

```
a, domtype(a)
```

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_1 & y & x \\ \text{"obj"} & 0 & 0 & 0 & 1 \\ \text{slk}_1 & -2 & 1 & -1 & -1 \end{pmatrix}, \text{linopt::Transparent}$$

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_1 & y & x \\ \text{"obj"} & 0 & 0 & 0 & 1 \\ \text{slk}_1 & -2 & 1 & -1 & -1 \end{pmatrix}, \text{DOM\_ARRAY}$$

```
delete a, k, t:
```

Example 2

We will write another simplex routine `mysimplex` for solving a linear program. For this we define the function `eigenpivot` for finding the pivot element of a given simplex tableau. `eigenpivot` assumes that the simplex tableau is given as a two dimensional array.

Here is the procedure `eigenpivot`, which is not coded in every detail, e.g., the error checking isn't implemented completely:

```
eigenpivot := proc(T: DOM_ARRAY)
   local i,j,m,n,k,l,mini;
   begin
     m := op(T,[0,2,2]):
     n := op(T,[0,3,2]):
     k := 0:
     l := 0:
     mini := unbesetzt:

      for j from 3 to n do
       if T[2,j] < 0 then
         l := j:
         break
       end_if:
     end_for:
     if l=0 then return(OPTIMAL) end_if:
     for i from 3 to m do
```

```
      if T[i,l] > 0 and (mini=unbesetzt or T[i,2]/T[i,l] < mini) then
        k := i:
        mini := T[k,2]/T[k,l]
      end_if
    end_for:
    if k=0 then return(UNBOUNDED) end_if:
    return(T[k,1],T[1,l]):
end_proc:
```

This is the new simplex algorithm `mysimplex` which uses `eigenpivot` and some function from the `linopt` library:

```
mysimplex := proc(P)
   local T;
   begin
     T := linopt::Transparent(P):
     T := linopt::Transparent::phaseI_tableau(T):
     piv := eigenpivot(linopt::Transparent::convert(T)):
     while piv <> OPTIMAL and piv <> UNBOUNDED do
       T := linopt::Transparent::userstep(T,piv):
       piv := eigenpivot(linopt::Transparent::convert(T))
     end_while:

      if piv = UNBOUNDED then
       error(" Phase I unbounded ?!?")
     end_if:
     if T[2,2] <> 0
        then return(EMPTY)
     end_if:
     T := linopt::Transparent::clean_basis(T):

      T := linopt::Transparent::phaseII_tableau(T):
     piv := eigenpivot(linopt::Transparent::convert(T)):
     while piv <> OPTIMAL and piv <> UNBOUNDED do
       T := linopt::Transparent::userstep(T,piv):
       piv := eigenpivot(linopt::Transparent::convert(T))
     end_while:

      if piv = OPTIMAL
       then return(linopt::Transparent::result(T))
       else return(UNBOUNDED)
     end_if
end_proc:
```

We now apply `mysimplex` to a linear program:

```
k := [{2*x + 2*y >= 4, -2*x + 4*y >= -2, -2*x + y>= -8,
       -2*x + y <= -2, y <= 6}, -x - y]:
k := append(k, NonNegative):
mysimplex(k);
```

$$\{x = 7, y = 6\}$$

```
delete k, eigenpivot, mysimplex:
```

Parameters

`tableau`

A simplex tableau of domain type

Return Values

Two dimensional array, representing the given simplex tableau `tableau`.

References

Papadimitriou, Christos H; Steiglitz, Kenneth: Combinatorial Optimization; Algorithms and Complexity. Prentice-Hall, 1982.

Nemhauser, George L; Wolsey, Laurence A: Integer and Combinatorial Optimization. New York, Wiley, 1988.

Salkin, Harvey M; Mathur, Kamlesh: Foundations of Integer Programming. North-Holland, 1989.

Neumann, Klaus; Morlock, Martin: Operations-Research. Munich, Hanser, 1993.

Duerr, Walter; Kleibohm, Klaus: Operations Research; Lineare Modelle und ihre Anwendungen. Munich, Hanser, 1992.

Suhl, Uwe H: MOPS - Mathematical OPtimization System. European Journal of Operational Research 72(1994)312-322. North-Holland, 1994.

Suhl, Uwe H; Szymanski, Ralf: Supernode Processing of Mixed Integer Models. Boston, Kluwer Academic Publishers, 1994.

See Also

MuPAD Functions

`linopt::Transparent` | `linopt::Transparent::autostep` | `linopt::Transparent::phaseI_tableau` | `linopt::Transparent::phaseII_tableau` | `linopt::Transparent::suggest` | `linopt::Transparent::userstep`

linopt::Transparent::dual_prices

Get the dual solution belonging to the given tableau

Syntax

```
linopt::Transparent::dual_prices(tableau)
```

Description

`linopt::Transparent::dual_prices(tableau)` returns the dual solution of the linear optimization problem given by `tableau`.

This procedure returns the dual solution belonging to the given tableau in form of a set of lists containing two elements, the first one is a restriction and the second one is the value belonging to the slack variable connected to the restriction in the dual solution.

Examples

Example 1

Here it is demonstrated that the dual solution of the final tableau is similar to the second element of the result of `linopt::minimize` using the option `DualPrices`:

First we compute the final tableau of the simplex algorithm:

```
k := [{x <= 2, y <= 2, x + 2*y >= 4}, - x + y, NonNegative]:
t := linopt::Transparent(k):
t := linopt::Transparent::simplex(t)
```

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_1 & \text{slk}_2 & \text{slk}_3 & y & x \\ \text{"obj"} & 1 & \frac{3}{2} & 0 & \frac{1}{2} & 0 & 0 \\ x & 2 & 1 & 0 & 0 & 0 & 1 \\ \text{slk}_2 & 1 & \frac{1}{2} & 1 & \frac{1}{2} & 0 & 0 \\ y & 1 & -\frac{1}{2} & 0 & -\frac{1}{2} & 1 & 0 \end{pmatrix}$$

Now we compute the solutions:

```
linopt::Transparent::dual_prices(t);
linopt::minimize(k, DualPrices)[2]
```

$$\left\{\left[4 \le x+2\,y,\ \frac{1}{2}\right],\ [0 \le x,\ 0],\ \left[x \le 2,\ \frac{3}{2}\right],\ [0 \le y,\ 0],\ [y \le 2,\ 0]\right\}$$

$$\left\{\left[4 \le x+2\,y,\ \frac{1}{2}\right],\ [0 \le x,\ 0],\ \left[x \le 2,\ \frac{3}{2}\right],\ [0 \le y,\ 0],\ [y \le 2,\ 0]\right\}$$

```
delete k, t:
```

Example 2

We compute the dual solution of another linear program:

```
k := [{x <= 2, y <= 2, x + 2*y >= 4}, -x + y, NonNegative]:
t := linopt::Transparent(k);
linopt::Transparent::dual_prices(t)
```

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_1 & \text{slk}_2 & \text{slk}_3 & y & x \\ \text{"obj"} & 0 & 0 & 0 & 0 & 1 & -1 \\ \text{slk}_1 & 2 & 1 & 0 & 0 & 0 & 1 \\ \text{slk}_2 & 2 & 0 & 1 & 0 & 1 & 0 \\ \text{slk}_3 & -4 & 0 & 0 & 1 & -2 & -1 \end{pmatrix}$$

$$\{[4 \le x + 2\,y,\ 0],\ [0 \le x,\ -1],\ [x \le 2,\ 0],\ [0 \le y,\ 1],\ [y \le 2,\ 0]\}$$

```
delete k, t:
```

Parameters

`tableau`

A simplex tableau of domain type `linopt::Transparent`

Return Values

Set of lists, each containing 2 elements.

References

Papadimitriou, Christos H; Steiglitz, Kenneth: Combinatorial Optimization; Algorithms and Complexity. Prentice-Hall, 1982.

Nemhauser, George L; Wolsey, Laurence A: Integer and Combinatorial Optimization. New York, Wiley, 1988.

Salkin, Harvey M; Mathur, Kamlesh: Foundations of Integer Programming. North-Holland, 1989.

Neumann, Klaus; Morlock, Martin: Operations-Research. Munich, Hanser, 1993.

Duerr, Walter; Kleibohm, Klaus: Operations Research; Lineare Modelle und ihre Anwendungen. Munich, Hanser, 1992.

Suhl, Uwe H: MOPS - Mathematical OPtimization System. European Journal of Operational Research 72(1994)312-322. North-Holland, 1994.

Suhl, Uwe H; Szymanski, Ralf: Supernode Processing of Mixed Integer Models. Boston, Kluwer Academic Publishers, 1994.

See Also

MuPAD Functions

`linopt::Transparent` | `linopt::Transparent::result`

linopt::Transparent::phaseI_tableau

Start an ordinary phase one of a 2-phase simplex algorithm

Syntax

```
linopt::Transparent::phaseI_tableau(tableau)
```

Description

`linopt::Transparent::phaseI_tableau` explicitly starts an (ordinary) phase one of the simplex algorithm , i.e. rows associated with infeasible basic variables are multiplied with -1 and another identity matrix with new slack variables is added to the given tableau. As soon as an optimal tableau with relative costs 0 is found the calculation can be continued with `linopt::Transparent::clean_basis` and the second phase of the simplex algorithm (`linopt::Transparent::phaseII_tableau`).

Examples

Example 1

The first simplex tableau is created and the first phase of the simplex algorithm is started:

```
t := linopt::Transparent([{x + y >= 2}, x, NonNegative]);
t := linopt::Transparent::phaseI_tableau(t)
```

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_1 & y & x \\ \text{"obj"} & 0 & 0 & 0 & 1 \\ \text{slk}_1 & -2 & 1 & -1 & -1 \end{pmatrix}$$

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_2 & \text{slk}_1 & y & x \\ \text{"obj"} & -2 & 0 & 1 & -1 & -1 \\ \text{slk}_2 & 2 & 1 & -1 & 1 & 1 \end{pmatrix}$$

We can see that a new slack variable, slk_2, was added to the tableau. And if we now execute `linopt::Transparent::simplex` we can see that we have just finished the first phase of the simplex algorithm:

```
linopt::Transparent::suggest(t);
t := linopt::Transparent::simplex(t):
linopt::Transparent::suggest(t)
```

$$\text{slk}_2,\ y$$

"linopt::Transparent::phaseII_tableau"

We continue the simplex algorithm by executing `linopt::Transparent::clean_basis`, `linopt::Transparent::phaseII_tableau` and `linopt::Transparent::simplex`. Observe in this special case `linopt::Transparent::clean_basis` is not necessary:

```
t := linopt::Transparent::clean_basis(t):
t := linopt::Transparent::phaseII_tableau(t):
t := linopt::Transparent::simplex(t);
linopt::Transparent::suggest(t)
```

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_1 & y & x \\ \text{"obj"} & 0 & 0 & 0 & 1 \\ y & 2 & -1 & 1 & 1 \end{pmatrix}$$

OPTIMAL

```
delete t:
```

Parameters

tableau

A simplex tableau of domain type `linopt::Transparent`

Return Values

Simplex tableau of domain type `linopt::Transparent`.

References

Papadimitriou, Christos H; Steiglitz, Kenneth: Combinatorial Optimization; Algorithms and Complexity. Prentice-Hall, 1982.

Nemhauser, George L; Wolsey, Laurence A: Integer and Combinatorial Optimization. New York, Wiley, 1988.

Salkin, Harvey M; Mathur, Kamlesh: Foundations of Integer Programming. North-Holland, 1989.

Neumann, Klaus; Morlock, Martin: Operations-Research. Munich, Hanser, 1993.

Duerr, Walter; Kleibohm, Klaus: Operations Research; Lineare Modelle und ihre Anwendungen. Munich, Hanser, 1992.

Suhl, Uwe H: MOPS - Mathematical OPtimization System. European Journal of Operational Research 72(1994)312-322. North-Holland, 1994.

Suhl, Uwe H; Szymanski, Ralf: Supernode Processing of Mixed Integer Models. Boston, Kluwer Academic Publishers, 1994.

See Also

MuPAD Functions

`linopt::Transparent` | `linopt::Transparent::autostep` | `linopt::Transparent::clean_basis` | `linopt::Transparent::convert` | `linopt::Transparent::dual_prices` |

linopt::Transparent::phaseII_tableau | linopt::Transparent::result | linopt::Transparent::simplex | linopt::Transparent::suggest | linopt::Transparent::userstep

linopt::Transparent::phaseII_tableau

Start phase two of a 2-phase simplex algorithm

Syntax

```
linopt::Transparent::phaseII_tableau(tableau)
```

Description

`linopt::Transparent::phaseII_tableau(tableau)` starts the second phase of the simplex algorithm on the given simplex tableau `tableau`.

After the explicitly started first phase of the simplex algorithm (see `linopt::Transparent::phaseI_tableau`) terminated with an optimal tableau with associated costs 0 and no phase one slack variables in the basis (see `linopt::Transparent::clean_basis`) this procedure can be used to start phase II. The procedure eliminates all artificial variables of phase I and their associated columns and reenters the old objective function modified for the new basis.

Examples

Example 1

The first simplex tableau is created and the first phase of the simplex algorithm is finished:

```
t := linopt::Transparent([{x + y >= 2}, x, NonNegative]):
t := linopt::Transparent::simplex(
        linopt::Transparent::phaseI_tableau(t))
```

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_2 & \text{slk}_1 & y & x \\ \text{"obj"} & 0 & 1 & 0 & 0 & 0 \\ y & 2 & 1 & -1 & 1 & 1 \end{pmatrix}$$

One sees that the artificial slack variable slk[2] of the first phase is removed by `linopt::Transparent::phaseII_tableau`. In this example it is not necessary to use `linopt::Transparent::clean_basis` for cleaning the basis:

```
linopt::Transparent::phaseII_tableau(t)
```

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_1 & y & x \\ \text{"obj"} & 0 & 0 & 0 & 1 \\ y & 2 & -1 & 1 & 1 \end{pmatrix}$$

```
delete t:
```

Example 2

Again the first simplex tableau is created and the first phase of the simplex algorithm is finished:

```
t := linopt::Transparent([{x <= 1, y <= 1, x + y >= 2},
                          0, NonNegative]):
t := linopt::Transparent::phaseI_tableau(t):
t := linopt::Transparent::simplex(t)
```

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_4 & \text{slk}_5 & \text{slk}_6 & \text{slk}_1 & \text{slk}_2 & \text{slk}_3 & y & x \\ \text{"obj"} & 0 & 2 & 2 & 0 & 1 & 1 & 1 & 0 & 0 \\ x & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ y & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ \text{slk}_6 & 0 & -1 & -1 & 1 & -1 & -1 & -1 & 0 & 0 \end{pmatrix}$$

In this example the artificial slack variable `slk[6]` is an element of the optimal basis. So we have to use `linopt::Transparent::clean_basis` before continuing with `linopt::Transparent::phaseII_tableau`, otherwise we will get an error message:

```
linopt::Transparent::phaseII_tableau(t)
```

```
Error: Clean the basis from phase I slack variables first. [linopt::Transparent::phase
```

```
t := linopt::Transparent::clean_basis(t):
linopt::Transparent::phaseII_tableau(t)
```

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_1 & \text{slk}_2 & \text{slk}_3 & y & x \\ \text{"obj"} & 0 & 0 & 0 & 0 & 0 & 0 \\ x & 1 & 0 & -1 & -1 & 0 & 1 \\ y & 1 & 0 & 1 & 0 & 1 & 0 \\ \text{slk}_1 & 0 & 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

```
delete t:
```

Parameters

`tableau`

A simplex tableau of domain type `linopt::Transparent`

Return Values

Simplex tableau of domain type `linopt::Transparent`.

References

Papadimitriou, Christos H; Steiglitz, Kenneth: Combinatorial Optimization; Algorithms and Complexity. Prentice-Hall, 1982.

Nemhauser, George L; Wolsey, Laurence A: Integer and Combinatorial Optimization. New York, Wiley, 1988.

Salkin, Harvey M; Mathur, Kamlesh: Foundations of Integer Programming. North-Holland, 1989.

Neumann, Klaus; Morlock, Martin: Operations-Research. Munich, Hanser, 1993.

Duerr, Walter; Kleibohm, Klaus: Operations Research; Lineare Modelle und ihre Anwendungen. Munich, Hanser, 1992.

Suhl, Uwe H: MOPS - Mathematical OPtimization System. European Journal of Operational Research 72(1994)312-322. North-Holland, 1994.

Suhl, Uwe H; Szymanski, Ralf: Supernode Processing of Mixed Integer Models. Boston, Kluwer Academic Publishers, 1994.

See Also

MuPAD Functions

`linopt::Transparent` | `linopt::Transparent::autostep` | `linopt::Transparent::clean_basis` | `linopt::Transparent::convert` | `linopt::Transparent::dual_prices` | `linopt::Transparent::phaseI_tableau` | `linopt::Transparent::result` | `linopt::Transparent::simplex` | `linopt::Transparent::suggest` | `linopt::Transparent::userstep`

linopt::Transparent::result

Get the basic feasible solution belonging to the given simplex tableau

Syntax

```
linopt::Transparent::result(tableau)
```

Description

`linopt::Transparent::result(tableau)` returns the basic feasible solution belonging to the given simplex tableau `tableau`.

Only the user defined variables are taken into account - the dual prices can be achieved by use of `linopt::Transparent::dual_prices`.

Examples

Example 1

We first compute an edge for an initial simplex tableau:

```
k := [{x <= 1, y <= 1, x + y >= 2}, 0, NonNegative]:
t := linopt::Transparent(k):
linopt::Transparent::result(t)
```

$$\{x = 0, y = 0\}$$

Now we compute the edge for the final tableau, which is identical to the optimal solution of the linear program given by `k`. We get the final simplex tableau by using `linopt::Transparent::simplex`:

```
t := linopt::Transparent(k):
t := linopt::Transparent::simplex(t):
linopt::Transparent::result(t)
```

$$\{x = 1, y = 1\}$$

```
linopt::minimize(k)
```

$$[\text{OPTIMAL}, \{x = 1, y = 1\}, 0]$$

```
delete k, t:
```

Parameters

`tableau`

A simplex tableau of domain type `linopt::Transparent`

Return Values

Set containing the values of the user defined variables for the feasible solution.

References

Papadimitriou, Christos H; Steiglitz, Kenneth: Combinatorial Optimization; Algorithms and Complexity. Prentice-Hall, 1982.

Nemhauser, George L; Wolsey, Laurence A: Integer and Combinatorial Optimization. New York, Wiley, 1988.

Salkin, Harvey M; Mathur, Kamlesh: Foundations of Integer Programming. North-Holland, 1989.

Neumann, Klaus; Morlock, Martin: Operations-Research. Munich, Hanser, 1993.

Duerr, Walter; Kleibohm, Klaus: Operations Research; Lineare Modelle und ihre Anwendungen. Munich, Hanser, 1992.

Suhl, Uwe H: MOPS - Mathematical OPtimization System. European Journal of Operational Research 72(1994)312-322. North-Holland, 1994.

Suhl, Uwe H; Szymanski, Ralf: Supernode Processing of Mixed Integer Models. Boston, Kluwer Academic Publishers, 1994.

See Also

MuPAD Functions

`linopt::Transparent` | `linopt::Transparent::dual_prices`

linopt::Transparent::simplex

Finish the current phase of the 2-phase simplex algorithm

Syntax

```
linopt::Transparent::simplex(tableau)
```

Description

`linopt::Transparent::simplex` runs the current phase of the 2-phase simplex algorithm until the end, i.e. if phase I was explicitly started (see `linopt::Transparent::phaseI_tableau`) the first phase will lead the optimal tableau. Sometimes it can be necessary to eliminate some slack variables of phase one by using `linopt::Transparent::clean_basis`.

If there was no phase I started by the user, `(linopt::Transparent)::simplex` returns the last optimal tableau or the empty set if there was no feasible solution found.

Examples

Example 1

We apply `linopt::Transparent::simplex` to an ordinary simplex tableau of a linear program and we get the optimal tableau:

```
k := [{x + y >= 2}, x, NonNegative]:
t := linopt::Transparent(k);
t := linopt::Transparent::simplex(t)
```

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_1 & y & x \\ \text{"obj"} & 0 & 0 & 0 & 1 \\ \text{slk}_1 & -2 & 1 & -1 & -1 \end{pmatrix}$$

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_1 & y & x \\ \text{"obj"} & 0 & 0 & 0 & 1 \\ y & 2 & -1 & 1 & 1 \end{pmatrix}$$

Let us proof the obtained result:

```
linopt::Transparent::suggest(t)
```

OPTIMAL

```
delete k, t:
```

Example 2

If the first phase of the simplex algorithm was started explicitly,`linopt::Transparent::simplex` returns only the optimal tableau of the first phase:

```
k := [{x + y >= 2}, y, NonNegative]:
t := linopt::Transparent(k):
t := linopt::Transparent::phaseI_tableau(t);
t := linopt::Transparent::simplex(t)
```

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_2 & \text{slk}_1 & y & x \\ \text{"obj"} & -2 & 0 & 1 & -1 & -1 \\ \text{slk}_2 & 2 & 1 & -1 & 1 & 1 \end{pmatrix}$$

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_2 & \text{slk}_1 & y & x \\ \text{"obj"} & 0 & 1 & 0 & 0 & 0 \\ y & 2 & 1 & -1 & 1 & 1 \end{pmatrix}$$

The next step of the simplex algorithm is computed:

```
linopt::Transparent::suggest(t)
```

"linopt::Transparent::phaseII_tableau"

With `linopt::Transparent::autostep` we execute the first step of the second phase of the simplex algorithm. One can see that the simplex algorithm is not finished yet:

```
t := linopt::Transparent::autostep(t):
linopt::Transparent::suggest(t);
```

$$y, x$$

If we then apply `linopt::Transparent::simplex` again we get the optimal solution. Here we don't had to use `linopt::Transparent::clean_basis`, before using `linopt::Transparent::autostep`, because there are no artificial variables in the basis computed by the first `linopt::Transparent::simplex` call above:

```
t := linopt::Transparent::simplex(t);
linopt::Transparent::suggest(t)
```

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_1 & y & x \\ \text{"obj"} & 0 & 0 & 1 & 0 \\ x & 2 & -1 & 1 & 1 \end{pmatrix}$$

OPTIMAL

```
delete k, t:
```

Parameters

`tableau`

A simplex tableau of domain type `linopt::Transparent`

Return Values

Simplex tableau of domain type `linopt::Transparent` or the empty set if there was no feasible solution found.

References

Papadimitriou, Christos H; Steiglitz, Kenneth: Combinatorial Optimization; Algorithms and Complexity. Prentice-Hall, 1982.

Nemhauser, George L; Wolsey, Laurence A: Integer and Combinatorial Optimization. New York, Wiley, 1988.

Salkin, Harvey M; Mathur, Kamlesh: Foundations of Integer Programming. North-Holland, 1989.

Neumann, Klaus; Morlock, Martin: Operations-Research. Munich, Hanser, 1993.

Duerr, Walter; Kleibohm, Klaus: Operations Research; Lineare Modelle und ihre Anwendungen. Munich, Hanser, 1992.

Suhl, Uwe H: MOPS - Mathematical OPtimization System. European Journal of Operational Research 72(1994)312-322. North-Holland, 1994.

Suhl, Uwe H; Szymanski, Ralf: Supernode Processing of Mixed Integer Models. Boston, Kluwer Academic Publishers, 1994.

See Also

MuPAD Functions

`linopt::Transparent` | `linopt::Transparent::autostep` | `linopt::Transparent::convert` | `linopt::Transparent::dual_prices` | `linopt::Transparent::phaseI_tableau` | `linopt::Transparent::result` | `linopt::Transparent::suggest` | `linopt::Transparent::userstep`

linopt::Transparent::suggest

Suggest the next step in the simplex algorithm

Syntax

```
linopt::Transparent::suggest(tableau)
```

Description

`linopt::Transparent::suggest(tableau)` suggests the next step in the simplex algorithm for the given simplex tableau `tableau`.

Normally this suggestion will be a pivot element, i.e. a sequence of a basic and a non-basic variable. If a phase I of the 2-phase simplex algorithm was started explicitly (see `linopt::Transparent::phaseI_tableau`) and the current tableau belongs to a feasible solution the suggestion will be the string `"linopt::Transparent::phaseII_tableau"`. At the end of the calculation the 'suggestion' is the identifier `OPTIMAL`.

The result of `linopt::Transparent::suggest` can be influenced if the global identifier `OPTIMAL` has a value. For this reason the identifier `OPTIMAL` is protected.

Examples

Example 1

We have a look at a linear program where the ordinary simplex tableau of the given problem is not the last tableau during the computation of the simplex algorithm. Looking at the ordinary simplex tableau we see that the element of the slk[2]-labeled row and the x-labeled column is a pivot element:

```
k := [{3*x + 4*y - 3*z <= 23, 5*x - 4*y - 3*z <= 10,
       7*x + 4*y + 11*z <= 30}, -x + y + 2*z, NonNegative]:
t := linopt::Transparent(k);
```

```
linopt::Transparent::suggest(t)
```

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_1 & \text{slk}_2 & \text{slk}_3 & y & x & z \\ \text{"obj"} & 0 & 0 & 0 & 0 & 1 & -1 & 2 \\ \text{slk}_1 & 23 & 1 & 0 & 0 & 4 & 3 & -3 \\ \text{slk}_2 & 10 & 0 & 1 & 0 & -4 & 5 & -3 \\ \text{slk}_3 & 30 & 0 & 0 & 1 & 4 & 7 & 11 \end{pmatrix}$$

$$\text{slk}_2,\ x$$

```
delete k, t:
```

Example 2

Here the ordinary simplex tableau still contains the solution of the linear program if the linear objective function is to minimize (see `linopt::Transparent` for more information):

```
k := [{x+y>=-1, x+y<=3}, x+2*y, NonNegative]:
t := linopt::Transparent(k);
linopt::Transparent::suggest(t)
```

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_1 & \text{slk}_2 & y & x \\ \text{"obj"} & 0 & 0 & 0 & 2 & 1 \\ \text{slk}_1 & 1 & 1 & 0 & -1 & -1 \\ \text{slk}_2 & 3 & 0 & 1 & 1 & 1 \end{pmatrix}$$

$$\text{OPTIMAL}$$

```
delete k, t:
```

Example 3

Here we explicitly start the first phase of the simplex algorithm. If we want a solution of the original linear program we have to apply the second phase of the simplex algorithm:

```
k := [{3*x + 4*y - 3*z <= 23, 5*x -4*y -3*z <= 10,
       7*x + 4*y + 11*z <= 30}, -x + y + 2*z, NonNegative]:
t := linopt::Transparent(k):
t := linopt::Transparent::phaseI_tableau(t):
t := linopt::Transparent::simplex(t):
linopt::Transparent::suggest(t)
```

```
"linopt::Transparent::phaseII_tableau"
```

```
delete k, t:
```

Parameters

`tableau`

A simplex tableau of domain type `linopt::Transparent`

Return Values

Sequence of 2 identifiers, the identifier `OPTIMAL` or the string `"linopt::Transparent::phaseII_tableau"`.

References

Papadimitriou, Christos H; Steiglitz, Kenneth: Combinatorial Optimization; Algorithms and Complexity. Prentice-Hall, 1982.

Nemhauser, George L; Wolsey, Laurence A: Integer and Combinatorial Optimization. New York, Wiley, 1988.

Salkin, Harvey M; Mathur, Kamlesh: Foundations of Integer Programming. North-Holland, 1989.

Neumann, Klaus; Morlock, Martin: Operations-Research. Munich, Hanser, 1993.

Duerr, Walter; Kleibohm, Klaus: Operations Research; Lineare Modelle und ihre Anwendungen. Munich, Hanser, 1992.

Suhl, Uwe H: MOPS - Mathematical OPtimization System. European Journal of Operational Research 72(1994)312-322. North-Holland, 1994.

Suhl, Uwe H; Szymanski, Ralf: Supernode Processing of Mixed Integer Models. Boston, Kluwer Academic Publishers, 1994.

See Also

MuPAD Functions

`linopt::Transparent` | `linopt::Transparent::autostep` | `linopt::Transparent::convert` | `linopt::Transparent::dual_prices` | `linopt::Transparent::phaseI_tableau` | `linopt::Transparent::result` | `linopt::Transparent::simplex` | `linopt::Transparent::userstep`

linopt::Transparent::userstep

Perform a user defined simplex step

Syntax

```
linopt::Transparent::userstep(tableau, basvar, nonbasbar)
```

Description

`linopt::Transparent::userstep(tableau, basvar, nonbasbar)` performs a user defined simplex step in the `tableau` with the pivot element defined by `basvar` and `nonbasvar`.

Examples

Example 1

We execute the simplex step given by the pivot element `(slk[1],x)`:

```
k := [{x + y >= 2}, x, NonNegative]:
t:= linopt::Transparent(k);
linopt::Transparent::userstep(t, slk[1], x)
```

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_1 & y & x \\ \text{"obj"} & 0 & 0 & 0 & 1 \\ \text{slk}_1 & -2 & 1 & -1 & -1 \end{pmatrix}$$

$$\begin{pmatrix} \text{"linopt"} & \text{"restr"} & \text{slk}_1 & y & x \\ \text{"obj"} & -2 & 1 & -1 & 0 \\ x & 2 & -1 & 1 & 1 \end{pmatrix}$$

Example 2

If we specify a wrong pivot element, we will get an error message:

```
k := [{3*x + 4*y - 3*z <= 23, 5*x - 4*y - 3*z <= 10,
       7*x + 4*y + 11*z <= 30}, -x + y + 2*z, NonNegative]:
t:= linopt::Transparent(k);
linopt::Transparent::userstep(t, x, y)
```

"linopt"	"restr"	slk_1	slk_2	slk_3	y	x	z
"obj"	0	0	0	0	1	−1	2
slk_1	23	1	0	0	4	3	−3
slk_2	10	0	1	0	−4	5	−3
slk_3	30	0	0	1	4	7	11

```
Error: The pivot element is not specified or specified incorrectly. [linopt::Transparer

delete k, t:
```

Parameters

`tableau`

A simplex tableau of domain type `linopt::Transparent`

`basvar`

A basic variable represented by an identifier that has to leave the basis

`nonbasvar`

A non-basic variable represented by an identifier that has to enter the basis

Return Values

Simplex tableau of domain type `linopt::Transparent`.

References

Papadimitriou, Christos H; Steiglitz, Kenneth: Combinatorial Optimization; Algorithms and Complexity. Prentice-Hall, 1982.

Nemhauser, George L; Wolsey, Laurence A: Integer and Combinatorial Optimization. New York, Wiley, 1988.

Salkin, Harvey M; Mathur, Kamlesh: Foundations of Integer Programming. North-Holland, 1989.

Neumann, Klaus; Morlock, Martin: Operations-Research. Munich, Hanser, 1993.

Duerr, Walter; Kleibohm, Klaus: Operations Research; Lineare Modelle und ihre Anwendungen. Munich, Hanser, 1992.

Suhl, Uwe H: MOPS - Mathematical OPtimization System. European Journal of Operational Research 72(1994)312-322. North-Holland, 1994.

Suhl, Uwe H; Szymanski, Ralf: Supernode Processing of Mixed Integer Models. Boston, Kluwer Academic Publishers, 1994.

See Also

MuPAD Functions

`linopt::Transparent` | `linopt::Transparent::autostep` | `linopt::Transparent::convert` | `linopt::Transparent::dual_prices` | `linopt::Transparent::phaseI_tableau` | `linopt::Transparent::result` | `linopt::Transparent::simplex` | `linopt::Transparent::suggest`

17

listlib – Manipulating Lists

listlib::insert
listlib::insertAt
listlib::merge
listlib::removeDuplicates
listlib::removeDupSorted
listlib::setDifference
listlib::singleMerge
listlib::sublist

listlib::insert

Insert an element into a list

Syntax

```
listlib::insert(list, element, <function>)
```

Description

`listlib::insert(list, element)` inserts `element` into `list`.

With the function `listlib::insert` any element can be inserted into any list.

With the third optional argument a function can be given that compare the elements of the list with the element to insert and therewith determines the position, the element is inserted. The given function is called with two elements and have to return `TRUE`, if the two elements are in the right order, otherwise `FALSE` (see next paragraph).

The given function is called step by step with an element of the list as first argument and the given element as second argument, until it returns `FALSE`. Then the given element is inserted into the list in *front* of the last proved element (see “Example 2” on page 17-3).

Note: The list must be ordered with regard to the order function, otherwise the element could be inserted at the wrong place.

If no third argument is given the function `_less` is used. If no order of the elements with regard to `_less` is defined, a function must be given, otherwise an error appears. The system function `sysorder` always can be used.

Examples

Example 1

Insert 3 into the given ordered list:

```
listlib::insert([1, 2, 4, 5, 6], 3)
```

$$[1, 2, 3, 4, 5, 6]$$

Insert 3 into the given descending ordered list. The insert function represents and preserves the order of the list:

```
listlib::insert([6, 5, 4, 2, 1], 3, _not@_less)
```

$$[6, 5, 4, 3, 2, 1]$$

Because identifiers cannot be ordered by `_less`, another function must be given, e.g., the function that represents the systems internal order:

```
listlib::insert([a, b, d, e, f], c, sysorder)
```

$$[a, b, c, d, e, f]$$

Example 2

Because no function is given as third argument, the function `_less` is used. `_less` is called: `_less(1, 3)`, `_less(2, 3)`, `_less(4, 3)` and then 3 is inserted in front of 4:

```
listlib::insert([1, 2, 4], 3)
```

$$[1, 2, 3, 4]$$

If the list is not ordered right, then the insert position could be wrong:

```
listlib::insert([4, 1, 2], 3)
```

$$[3, 4, 1, 2]$$

Example 3

The following example shows, how expressions can be ordered by a user defined priority. This order is given by the function named `priority`, which returns a smaller number, when the expression has a type with higher priority:

```
priority := X -> contains(["_power", "_mult", "_plus"], type(X)):
priority(x^2), priority(x + 2)
```

$$1, 3$$

The function `sortfunc` returns `TRUE`, if the both given arguments are in the right order, i.e., the first argument has a higher (or equal) priority than the second argument:

```
sortfunc := (X, Y) -> bool(priority(Y) > priority(X)):
sortfunc(x^2, x + 2), sortfunc(x + 2, x*2)
```

$$\text{TRUE}, \text{FALSE}$$

Now the expression `x*2` is inserted at the "right" place in the list:

```
listlib::insert([x^y, x^2, x*y, -y, x + y], x*2, sortfunc)
```

$$[x^y, x^2, 2\,x, x\,y, -y, x+y]$$

Parameters

`list`

MuPAD list

`element`

MuPAD expression to insert

`function`

Function that determines the insert position

Return Values

Given list enlarged with the inserted element

See Also

MuPAD Functions

`_concat` | `append` | `listlib::insertAt`

listlib::insertAt

Insert an element into a list at a given position

Syntax

```
listlib::insertAt(list, element, <pos>)
```

Description

`listlib::insertAt(list, element, pos)` inserts `element` into `list` at position `pos`.

With the function `listlib::insertAt` any element can be inserted into any list at a specified place.

The third argument (the "insert index") determines the place to insert the element into the given list.

If the insert index is less than `1` the element is inserted in front of the list. If the insertion index is greater than `nops(list)` the element is appended to the list. To append an element to a list the kernel function `append` is faster.

If no third argument is given, the given element is inserted in front of the list.

If the argument `element` is a list too, the elements of this list will be inserted (or appended) instead of the whole list by preserving the order.

Examples

Example 1

Insertion `2` at the third place of the given list:

```
listlib::insertAt([1, 1, 1], 2, 3)
```

```
[1, 1, 2, 1]
```

Insertion of an element in front of a list. The third argument is optional in this case:

```
listlib::insertAt([1, 1, 3, 1], 2, 0), listlib::insertAt([1, 1, 3, 1], 2)
```

$$[2, 1, 1, 3, 1], [2, 1, 1, 3, 1]$$

Appending of an element. This could be done faster with `append`:

```
listlib::insertAt([1, 2, 3], 4, 1000), append([1, 2, 3], 4)
```

$$[1, 2, 3, 4], [1, 2, 3, 4]$$

Parameters

`list`

A list

`element`

Any MuPAD object

`pos`

Any integer

Return Values

Given list enlarged with the inserted element

See Also

MuPAD Functions

`_concat` | `append` | `listlib::insert`

listlib::merge

Merging two ordered lists

Syntax

```
listlib::merge(list1, list2, <function>)
```

Description

`listlib::merge(list1, list2)` merges both lists into one list.

With the third optional argument a function can be given that compare the elements of the lists and therewith determines the order of the elements. The given function is called with two elements and have to return `TRUE`, if the two elements are in the right order, otherwise `FALSE` (see next paragraph).

The given function is called step by step with an element of the first list as first argument and an element of the second list as second argument, until it returns `FALSE`. Then the element of the second list is inserted into the first list in *front* of the last proved element (see “Example 2” on page 17-9).

Note: The lists must be ordered with regard to the order function, otherwise the elements could be inserted at the wrong place.

If no third argument is given the function `_less` is used. If no order of the elements with regard to `_less` is defined, a function must be given, otherwise an error appears. The system function `sysorder` always can be used.

Examples

Example 1

Merging two ascending ordered lists:

```
listlib::merge([1, 3, 5, 7], [2, 4, 6, 8])
```

$$[1, 2, 3, 4, 5, 6, 7, 8]$$

Merging two descending ordered lists:

```
listlib::merge([7, 5, 3, 1], [8, 6, 4, 2], _not@_less)
```

$$[8, 7, 6, 5, 4, 3, 2, 1]$$

Example 2

The following example shows, how expressions can be ordered by a user defined priority. This order is given by the function named `priority`, which returns a smaller number, when the expression has a type with higher priority:

```
priority := X -> contains(["_power", "_mult", "_plus"], type(X)):
priority(x^2), priority(x + 2)
```

$$1, 3$$

The function `sortfunc` returns `TRUE`, if the both given arguments are in the right order, i.e., the first argument has a higher (or equal) priority than the second argument:

```
sortfunc := (X, Y) -> bool(priority(Y) > priority(X)):
sortfunc(x^2, x + 2), sortfunc(x + 2, x*2)
```

$$\text{TRUE}, \text{FALSE}$$

Now the both lists are merged with regard to the given priority:

```
listlib::merge([x^y, x*2, -y], [x^2, x*y, x + y], sortfunc)
```

$$[x^y, x^2, 2\,x, -y, x\,y, x+y]$$

```
delete priority, sortfunc:
```

Parameters

`list1, list2`

A MuPAD list

`function`

A function that determines the merging order

Return Values

Ordered list that contains the elements of both lists

See Also

MuPAD Functions

`_concat` | `listlib::insert` | `listlib::singleMerge` | `zip`

listlib::removeDuplicates

Removes duplicate entries

Syntax

```
listlib::removeDuplicates(list, <KeepOrder>)
```

Description

`listlib::removeDuplicates(list)` removes all duplicates of each entry of the list `list`. The new list is build up from right to left with the order of the *last* occurrence of each entry in `list`. Cf. “Example 1” on page 17-11.

A faster possibibliy to remove duplicate entries is to convert the list into a `set` and back into a list. You will loose the order of the list entries in this case. Cf. “Example 3” on page 17-12.

Examples

Example 1

Per default `listlib::removeDuplicates` removes duplicate entries in reverse order:

```
list:= [1, 1, 1, 3, 1, 5, 5, 1, 3, 3, 1, 7]:
listlib::removeDuplicates(list)
```

```
[5, 3, 1, 7]
```

Example 2

With option `KeepOrder` entries are selected in the order of their occurrence:

```
list:= [1, 1, 1, 3, 1, 5, 5, 1, 3, 3, 1, 7]:
listlib::removeDuplicates(list, KeepOrder)
```

```
[1, 3, 5, 7]
```

Example 3

If you don't need the order of list entries any more, you may convert the list into a set and back into a list, this is much faster:

```
list:= [1, 1, 1, 3, 1, 5, 5, 1, 3, 3, 1, 7]:
[op({op(list)})]
```

```
[1, 3, 5, 7]
```

Parameters

`list`

A MuPAD list

Options

`KeepOrder`

`listlib::removeDuplicates(list, KeepOrder)` returns a list of the entries of `list` in the order of their *first* occurrence. The list is build up from left to right. See “Example 2” on page 17-11.

Return Values

List that contains each entry only once

See Also

MuPAD Domains
`DOM_LIST`

MuPAD Functions
`listlib::removeDupSorted`

listlib::removeDupSorted

Remove duplicates of any element from ordered lists

Syntax

```
listlib::removeDupSorted(list)
```

Description

`listlib::removeDupSorted(list)` removes all duplicates of any element of the ordered list `list`.

`listlib::removeDupSorted` does the same as `listlib::removeDuplicates`, but it assumes that the list is sorted and, therefore, it is faster. A notable gain will only occur, if there are only few duplicates in a long list.

Examples

Example 1

`listlib::removeDupSorted` removes all duplicates from the given list:

```
listlib::removeDupSorted([1, 1, 1, 3, 5, 5, 5, 5, 5, 5, 5, 7, 7, 7])
```

```
[1, 3, 5, 7]
```

If the list is not ordered, `listlib::removeDupSorted` fails:

```
listlib::removeDupSorted([1, 3, 5, 7, 1, 3, 5, 7, 1, 3, 5, 7])
```

```
[1, 3, 5, 7, 1, 3, 5, 7, 1, 3, 5, 7]
```

Parameters

`list`

An ordered MuPAD list

Return Values

List that contains every element only once

See Also

MuPAD Functions

`listlib::removeDuplicates`

listlib::setDifference

Remove elements from a list

Syntax

```
listlib::setDifference(list1, list2)
```

Description

`listlib::setDifference(list1, list2)` removes all elements from `list1`, that are given by `list2`.

Note: The order of the list is not preserved.

Examples

Example 1

`listlib::setDifference` removes `2`, `4`, `6` and `8` from the given list:

```
listlib::setDifference([1, 2, 3, 4, 5, 6, 7, 8], [2, 4, 6, 8])
```

```
[1, 3, 5, 7]
```

Parameters

`list1`, `list2`

A MuPAD list

Return Values

First list without all elements of the second list

See Also

MuPAD Functions
`minus`

listlib::singleMerge

Merging of two ordered lists without duplicates

Syntax

```
listlib::singleMerge(list1, list2, <function>)
```

Description

`listlib::singleMerge(list1, list2)` merges the both lists into one list. It is assumed that the lists are "disjunct", no element appears in both lists. Otherwise such elements are inserted only once in the result list.

With the third optional argument a function can be given that compare the elements of the lists and therewith determines the order of the elements. The given function is called with two elements and have to return `TRUE`, if the two elements are in the right order, otherwise `FALSE` (see next paragraph).

The given function is called step by step with an element of the first list as first argument and an element of the second list as second argument, until it returns `FALSE`. Then the element of the second list is inserted into the first list in *front* of the last proved element (see "Example 3" on page 17-19).

Note: The lists must be ordered with regard to the order function, otherwise the elements could be inserted at the wrong place.

If no third argument is given the function `_less` is used. If no order of the elements with regard to `_less` is defined, a function must be given, otherwise an error appears. The system function `sysorder` always can be used.

Examples

Example 1

Merging two ascending ordered lists:

```
listlib::singleMerge([1, 3, 5, 7], [2, 4, 6, 8])
```

$$[1, 2, 3, 4, 5, 6, 7, 8]$$

Merging two descending ordered lists:

```
listlib::singleMerge([7, 5, 3, 1], [8, 6, 4, 2], _not@_less)
```

$$[8, 7, 6, 5, 4, 3, 2, 1]$$

Example 2

Merging two ascending ordered lists with duplicates:

```
listlib::singleMerge([1, 2, 5, 7], [2, 5, 6, 8])
```

$$[1, 2, 5, 6, 7, 8]$$

But the following lists does not contain mutual equal elements:

```
listlib::singleMerge([1, 1, 3, 3], [2, 2, 4, 4])
```

$$[1, 1, 2, 2, 3, 3, 4, 4]$$

Example 3

The following example shows, how expressions can be ordered by a user defined priority. This order is given by the function named `priority`, which returns a smaller number when the expression has a type with higher priority:

```
priority := X -> contains(["_power", "_mult", "_plus"], type(X)):
priority(x^2), priority(x + 2)
```

$$1, 3$$

The function `sortfunc` returns `TRUE`, if the both given arguments are in the right order, i.e., the first argument has a higher (or equal) priority than the second argument:

```
sortfunc := (X, Y) -> bool(priority(Y) > priority(X)):
sortfunc(x^2, x + 2), sortfunc(x + 2, x*2)
```

$$\text{TRUE}, \text{FALSE}$$

Now the both lists are merged with regard to the given priority:

```
listlib::singleMerge([x^y, x*2, -y], [x^2, x*y, x + y], sortfunc)
```

$$[x^y, x^2, 2\,x, -y, x\,y, x+y]$$

```
delete priority, sortfunc:
```

Parameters

`list1, list2`

A MuPAD list

`function`

A function that determines the merging order

Return Values

Ordered list that contains the elements of both lists

See Also

MuPAD Functions

`_concat` | `listlib::insert` | `listlib::merge` | `zip`

listlib::sublist

Search sublists

Syntax

```
listlib::sublist(list1, list2, <index>, <Consecutive>)
```

Description

`listlib::sublist(list1, list2)` determines, whether the list `list1` contains another list `list2`.

With `listlib::sublist` the position of the first appearance of a list in another list can be determined.

The position that was found is returned as integer. If the given list does not contain the given `sublist`, the number `0` is returned.

If an index is given, the search starts at this position. There with multiple occurrence of a sublist can be determined.

With the option `Consecutive`, the list must contain the sublist in one piece without elements in between.

Examples

Example 1

The sublist is a part of the list, but not in one piece:

```
listlib::sublist([1, 2, 3, 4, 5, 6, 7, 8], [2, 3, 5, 6])
```

```
2
```

```
listlib::sublist([1, 2, 3, 4, 5, 6, 7, 8], [2, 3, 5, 6], Consecutive)
```

```
    0
```

The list contains the sublist, coherent and incoherent:

```
listlib::sublist([1, 2, 3, 4, 5, 1, 3, 5], [1, 3, 5])
```

```
    1
```

```
listlib::sublist([1, 2, 3, 4, 5, 1, 3, 5], [1, 3, 5], Consecutive)
```

```
    6
```

Example 2

Find the last occurrence of the sublist inside of the list:

```
POS:= 0:
while listlib::sublist([1, 2, 3, 1, 3, 1, 2, 3], [1, 2, 3], POS + 1) > 0 do
  POS:= listlib::sublist([1, 2, 3, 1, 3, 1, 2, 3], [1, 2, 3], POS + 1)
end_while:
POS
```

```
    6
```

```
delete POS:
```

Parameters

`list1`, `list2`

MuPAD list

`index`

Integer that determines the first search position

Options

`Consecutive`

Determines that the sublist `list2` is containing coherent in `list1`

Return Values

Position of the first element of the containing sublist or zero

See Also

MuPAD Functions
`contains` | `op`

misc – Miscellanea

misc::breakmap
misc::genassop
misc::maprec
misc::pslq

misc::breakmap

Stops the mapping currently done by maprec

Syntax

```
misc::breakmap()
```

Description

`misc::breakmap`() stops the recursive application of a function to all subexpressions of an expression that `misc::maprec` is just working on.

`misc::breakmap` is useful as a command inside the procedure mapped by `misc::maprec` in case we know that we are finished with our work and the remaining recursive mapping is not necessary.

Examples

Example 1

We want to know whether a given expression contains a particular type `t`. As soon as we have found the first occurrence of `t`, we can terminate our search.

```
myfound := FALSE:
misc::maprec(hold(((23+5.0)/3+4*I)*PI),
            {DOM_COMPLEX}=proc()
                          begin
                            myfound := misc::breakmap();
                            args()
                          end_proc):
myfound
```

```
    TRUE
```

What did we do? We told `misc::maprec` just to go down the expression tree and look for subexpressions of type `DOM_COMPLEX`; and, whenever such subexpression should be

found, to apply a certain procedure to it. That procedure stops the recursive mapping, remembers that we have found the type we had searched for, and returns exactly its argument such that the result returned by `misc::maprec` equals the input. In the example below, we test whether our given expression contains the type `DOM_POLY`.

```
myfound := FALSE:
misc::maprec(hold(((23+5.0)/3+4*I)*PI),
             {DOM_POLY}=proc()
                        begin
                          myfound := misc::breakmap();
                          args()
                        end_proc):
myfound
```

```
FALSE
```

Note that you do not need to use this method when searching for subexpressions of a given type; calling `hastype` is certainly more convenient.

Return Values

`misc::breakmap` always returns `TRUE`.

See Also

MuPAD Functions

`misc::maprec`

misc::genassop

Generates an n-ary associative operator from a binary one

Syntax

```
misc::genassop(binaryop, zeroelement)
```

Description

`misc::genassop(binaryop, zeroelement)` generates an n-ary associative operator from the binary operator `binaryop`, where `zeroelement` is a neutral element for `binaryop`.

`binaryop` must be a function taking two arguments (no matter of what kind) and returning a valid argument to itself. It must satisfy the associative law `binaryop`(`binaryop`(a, b), c) = `binaryop`(a, `binaryop`(b, c)).

`zeroelement` is an object such that `binaryop(a, zeroelement) = a` holds for every `a`.

`misc::genassop` returns a procedure which returns `zeroelement` if it is called without arguments and the argument if it is called with one argument.

Note: `misc::genassop` doesn't check whether `binaryop` is really associative and whether `zeroelement` is really a neutral element for `binaryop`.

Examples

Example 1

We know that `_plus` is an n-ary operator anyway, but let us assume that `_plus` was only a binary operator. We can create an own n-ary addition as follows:

```
myplus := misc::genassop(_plus, 0)
```

```
proc genericAssop() ... end
```

Now we make `myplus` add some values.

```
myplus(3, 4, 8), myplus(-5), myplus()
```

```
15, -5, 0
```

As mentioned in the "Details" section, `myplus` returns the argument if is called with exactly one argument, and it returns the `zeroelement` `0` if it is called without arguments.

Parameters

`binaryop`

A function

`zeroelement`

An object

Return Values

`misc::genassop` returns a procedure `f`. That procedure accepts an arbitrary number of arguments of the same kind `binaryop` does; it returns `zeroelement` if it is called without argument, and its only argument if it is called with one argument; its value on *n* arguments is inductively defined by `f(x1, ..., xn)=f(binaryop(x1,x2), x3,...,xn)`.

See Also

MuPAD Functions

`operator`

misc::maprec

Map a function to subexpressions of an expression

Syntax

```
misc::maprec(ex, selector = funci, …, <PreMap | PostMap>, <NoOperators>, <Unsimplified>
```

Description

`misc::maprec(ex, selector=funci)` maps the function `funci` to all subexpressions of the expression `ex` that satisfy a given criterion (defined by `selector`) and replaces each selected subexpression `s` by `funci(s)`.

Several different functions may be mapped to subexpressions satisfying different selection criteria.

`misc::maprec(ex, selector`$_1$ `= funci`$_1$`, …, selector`$_n$ `= funci`$_n$`)` does two steps: it tests whether `ex` meets a selection criterion defined by some selector `selector_k` (and, if yes, replaces `ex` by `funci_k(ex)`); and it applies itself recursively to all operands of `ex`. The order of these steps is determined by the options `PreMap` and `PostMap`.

Selectors are applied from left to right; if the expression meets some selection criterion, no further selectors are tried.

`selector` can have two forms. It can be a set $\{t_1, \ldots, t_n\}$. Here a subexpression `s` of `ex` is selected if type(s1) is one of the types $t_1, \ldots, t_n$. If it is not a set, a subexpression `s` of `ex` is selected if `p`(s) returns `TRUE`. As every MuPAD object may be applied as a function to `s`, `p` may be of any type in the latter case.

In order not to select a subexpression, the selector need not return `FALSE`; it suffices that it does not return `TRUE`.

If neither the option `PreMap` nor the option `PostMap` is given, then `PreMap` is used.

Use a `misc::breakmap` command inside `funci` in order to stop the recursive mapping. See the help page of `misc::breakmap` for an example.

Note: Only subexpressions of domain type `DOM_ARRAY`, `DOM_EXPR`, `DOM_LIST`, `DOM_SET`, and `DOM_TABLE` are mapped recursively, as well as domain elements of a domain `T` for which a slot `T::enableMaprec` exists and equals `TRUE`; a slot `T::map` working properly must then exist, too. To subexpressions of other types, `selector` is applied, but `misc::maprec` is not mapped to their operands. (This is to avoid unwanted substitutions.) If you want to recurse on them, either add an `enableMaprec`-slot, or use a `selector` that selects such subexpressions, and make `funci` initiate another recursive mapping.

`misc::maprec` is overloadable. If the domain of a subexpression has a method `"maprec"`, then this method is called with the subexpression and the other arguments of the call.

Note: The subexpression is replaced by the result, but `misc::maprec` is not mapped to its operands; such recursive mapping must be done by the domain method if desired.

Note: The operators of expressions (`op(expression, 0)`) are also mapped recursively like all the other operands. Use `NoOperators` to switch this off.

Examples

Example 1

In the following example every integer of the given expression `a+3+4` is substituted by the value 10. Since `10(n)` returns `10` for every integer `n`, it suffices to write `10` instead of `n -> 10` here.

```
misc::maprec(hold(a+3+4), {DOM_INT} = 10)
```

$$a + 20$$

In the example above, we used `hold` to suppress the evaluation of the expression because otherwise `a+3+4` is evaluated to `a+7` and we get the result:

```
misc::maprec(a+3+4, {DOM_INT} = 10)
```

$$a + 10$$

The simplification of the resulting `10 + 10` to `20` can be avoided by using the option `Unsimplified`:

```
misc::maprec(hold(a+3+4), {DOM_INT} = 10, Unsimplified)
```

$$a + 10 + 10$$

Example 2

Now we give an example where the `selector` is a function. We want to eleminate all the prime numbers from an expression.

```
misc::maprec(hold(_plus)(i $ i=1..20), isprime= null(), PostMap)
```

133

Here `isprime` returns `TRUE` for every prime number between 1 and 20. Every prime number between 1 and 20 is replaced by null() (since `null()(p)` gives `null()`) which means the above call computes the sum of all non-prime numbers between 1 and 20.

Example 3

Normally, `misc::maprec` recurses also into the operators of subexpressions. This may be unwanted in many cases:

```
misc::maprec(a+b, {DOM_IDENT}= (x -> x.1))
```

plus1(a1, b1)

We just wanted to replace the summands, but not the operator. Using the option `NoOperators` helps:

```
misc::maprec(a+b, {DOM_IDENT}= (x -> x.1), NoOperators)
```

$$a1 + b1$$

Parameters

`ex`

Any MuPAD object

`selector`

Any MuPAD object

`funci`

Any MuPAD object

Options

`PreMap`

For each subexpressions `s` of `ex`, the selector is applied to it *after* visiting all of its subexpressions; `s` may have changed at that time due to substitutions in the subexpressions.

`PostMap`

For each subexpressions `s` of `ex`, the selector is applied to it *before* visiting its subexpressions. If `s` is selected by `selector`, it is replaced by `funci(s)`, and `misc::maprec` is *not* recursively applied to the operands of `funci(s)`; otherwise, `misc::maprec` is recursively applied to the operands of `s`.

`NoOperators`

The selector is not applied to the operator of `ex`.

`Unsimplified`

The resulting expressions are not further simplified.

Return Values

`misc::maprec` may return any MuPAD object.

Overloaded By

ex

See Also

MuPAD Functions
`map` | `mapcoeffs` | `misc::breakmap`

misc::pslq

Heuristic detection of relations between real numbers

Syntax

```
misc::pslq(numberlist, precision)
```

Description

`misc::pslq(numberlist, precision)` returns a list of integers `[k1, ..., kn]` such that — denoting the elements of `numberlist` by `a1, ..., an` — the absolute value of $\sum_{i=0}^{n} a_i k_i$ is less than $\frac{1}{10^{\text{precision}}}$ times the Euclidean norm of `numberlist`, or `FAIL` if such integers could not be detected.

This method can be used to get an idea about linear dependencies, before proving them.

Environment Interactions

`misc::pslq` is *not* affected by the current value of `DIGITS`. Numerical computations are carried out with more significant digits such that the output meets the specification given above.

Examples

Example 1

Does π satisfy a polynomial equation of degree at most 2 ?

```
misc::pslq([1, PI, PI^2], 20)
```

```
FAIL
```

Example 2

Having forgotten the relation between sine and cosine, we can try the heuristic way.

```
misc::pslq([1, sin(0.32), sin(0.32)^2, cos(0.32), cos(0.32)^2], 10)
```

```
(1 0 -1 0 -1)
```

Parameters

`numberlist`

List of real numbers or objects that can be converted to real numbers by the function `float`.

`precision`

Positive integer

Return Values

List of integers, or `FAIL`

Algorithms

This function has been written by Raymond Manzoni.

The algorithm has been taken from Bailey and Plouffe, *Recognizing numerical constants*. See also Helaman R.P. Ferguson and David Bailey, *A Polynomial Time, Numerically Stable Integer Relation Algorithm*, RNR Technical Report RNR-92-032.

numeric – Numerical Algorithms

numeric::butcher
numeric::complexRound
numeric::cubicSpline
numeric::cubicSpline2d
numeric::det
numeric::eigenvalues
numeric::eigenvectors
numeric::expMatrix
numeric::factorCholesky
numeric::factorLU
numeric::factorQR
numeric::fft
numeric::invfft
numeric::fMatrix
numeric::fsolve
numeric::gaussAGM
numeric::gldata
numeric::gtdata
numeric::indets
numeric::int
numeric::inverse
numeric::leastSquares
numeric::linsolve
numeric::matlinsolve
numeric::ncdata
numeric::odesolve
numeric::odesolve2
numeric::odesolveGeometric
numeric::ode2vectorfield
numeric::odeToVectorField
numeric::polyrootbound
numeric::polyroots

numeric::polysysroots
numeric::product
numeric::quadrature
numeric::rank
numeric::rationalize
numeric::realroot
numeric::realroots
numeric::rotationMatrix
numeric::singularvalues
numeric::singularvectors
numeric::svd
numeric::solve
numeric::sort
numeric::spectralradius
numeric::spectralRadius
numeric::sum

numeric::butcher

Butcher parameters of Runge-Kutta schemes

Syntax

```
numeric::butcher(EULER1 | RKF43 | xRKF43 | RK4 | RKF34 | xRKF34 | RKF54a | xRKF54a | RK
```

Description

`numeric::butcher(method)` returns the Butcher parameters of the Runge-Kutta scheme named `method`.

An s-stage Runge-Kutta method for the numerical integration of a dynamical system $\frac{dy}{dt} = f(t, y)$ with step size h is a map

$$(t, y) \to \left(t + h,\ y + h\, b_1\, k_1 + \cdots + h\, b_s\, k_s\right).$$

The "intermediate stages" $k_1, \ldots, k_s$ are defined as the solutions of the algebraic equations

$$k_i = f\left(t + c_i\, h,\ y + h\, a_{i1}\, k_1 + \cdots + h\, a_{is}\, k_s\right),\ 1 \le i \le s.$$

If the $s \times s$ "Butcher matrix" a_{ij} is strictly lower triangular, the method is called "explicit". In this case, the intermediate stages are computed recursively as:

$$\begin{aligned} k_1 &= f(t, y) \\ k_2 &= f\left(t + c_2\, h,\ y + h\, a_{21}\, k_1\right) \\ &\vdots \\ k_s &= f\left(t + c_s\, h,\ y + h\, a_{s1}\, k_1 + \ldots + h\, a_{s,s-1}\, k_{s-1}\right). \end{aligned}$$

Various numerical schemes arise from different choices of the Butcher parameters: the $s \times s$-matrix a_{ij}, the weights $b = [b_1, \ldots, b_s]$ and the abscissae $c = [c_1, c_2, \ldots, c_s]$.

Embedded pairs of Runge-Kutta methods consist of two methods that share the matrix a_{ij} and the abscissae c_i, but use different weights b_i.

The returned list `[s, c, a, b1, b2, order1, order2]` contains the Butcher data of the method: `s` is the number of stages, `c` is the list of abscissae, `a` is the Butcher matrix, b_1 and b_2 are lists of weights. The integers $order_1$ and $order_2$ are the orders of the scheme when using the weights b_1 or b_2, respectively, in conjunction with the matrix `a` and the abscissae `c`.

The methods `EULER1` (order 1), `RK4` (order 4) and `BUTCHER6` (order 6) are single methods with $b_1 = b_2$ and $order_1 = order_2$. All other methods are embedded pairs of Runge-Kutta-Fehlberg (`RKFxx`), Dormand-Prince (`DOPRIxx`) or Cash-Karp (`CKxx`) type. The names indicate the orders of the subprocesses, e.g., `CK45` is the Cash-Karp pair of orders 4 and 5. `CK54` is the same pair with reversed ordering of the subprocesses. The second subprocess is used to produce a time step of the Runge-Kutta method, the first subprocess is used for estimating the error of the time step.

The methods `GAUSS(s)` or, equivalently, `GAUSS = s` are the implicit Gauss methods with `s` stages of order $2s$.

The data of all explicit methods are returned as exact rational numbers. The data of the Gauss methods are returned as floating-point numbers.

The Butcher data are called by the routines `numeric::odesolve`, `numeric::odesolve2`, and `numeric::odesolveGeometric`.

Environment Interactions

When computing the data for `GAUSS(s)`, the function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

The Butcher data of the classical 4 stage, 4th order Runge-Kutta scheme are:

```
numeric::butcher(RK4)
```

$$\left[4,\ \begin{pmatrix}0 & \frac{1}{2} & \frac{1}{2} & 1\end{pmatrix},\ \begin{pmatrix}0 & 0 & 0 & 0\\ \frac{1}{2} & 0 & 0 & 0\\ 0 & \frac{1}{2} & 0 & 0\\ 0 & 0 & 1 & 0\end{pmatrix},\ \begin{pmatrix}\frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6}\end{pmatrix},\ \begin{pmatrix}\frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6}\end{pmatrix},\ 4,\ 4\right]$$

Note that the weights b_1 and b_2 coincide: this classical method does not provide an embedded pair.

The Butcher data of the (implicit) 3 stage Gauss method:

```
DIGITS := 5:
numeric::butcher(GAUSS(3));
delete DIGITS:
```

$$\left[3,\ \begin{pmatrix}0.1127 & 0.5 & 0.8873\end{pmatrix},\ \begin{pmatrix}0.13889 & -0.035977 & 0.0097894\\ 0.30026 & 0.22222 & -0.022485\\ 0.26799 & 0.48042 & 0.13889\end{pmatrix},\ \begin{pmatrix}0.27778 & 0.44444 & 0.27778\end{pmatrix},\ \begin{pmatrix}0.27778 & 0.44444 & 0.27778\end{pmatrix},\ 6,\ 6\right]$$

Example 2

The Butcher data of the embedded Runge-Kutta-Fehlberg pair `RKF34` of orders 3 and 4 are:

```
[s, c, a, b1, b2, order1, order2] := numeric::butcher(RKF34):
```

The number of stages `s` of the 4th order subprocess is 5, the abscissae `c` and the matrix `a` are given by:

```
s, c, a
```

$$5, \left(0\ \frac{1}{4}\ \frac{4}{9}\ \frac{6}{7}\ 1\right), \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{4} & 0 & 0 & 0 & 0 \\ \frac{4}{81} & \frac{32}{81} & 0 & 0 & 0 \\ \frac{57}{98} & -\frac{432}{343} & \frac{1053}{686} & 0 & 0 \\ \frac{1}{6} & 0 & \frac{27}{52} & \frac{49}{156} & 0 \end{pmatrix}$$

Using these parameters with the weights

```
b1, b2
```

$$\left(\frac{1}{6}\ 0\ \frac{27}{52}\ \frac{49}{156}\ 0\right), \left(\frac{43}{288}\ 0\ \frac{243}{416}\ \frac{343}{1872}\ \frac{1}{12}\right)$$

yields a numerical scheme of order 3 or 4, respectively:

```
order1, order2
```

$$3, 4$$

```
delete s, c, a, b1, b2, order1, order2:
```

Example 3

We plot the stability regions of the two sub-methods of `DOPRI78`. The stability function of a Runge-Kutta scheme with Butcher parameters (c, a, b) is given by

$$p(z) = 1 + z\left(\left\langle b, \frac{e}{(1 - z\,a)}\right\rangle\right),$$

where e is the column vector $(1, 1, \ldots, 1)^T$. For an explicit s-stage scheme (the matrix a is strictly lower triangular), this stability function reduces to the polynomial

$$p(z) = 1 + \left(\sum_{i=1}^{s} z^i \left(\left\langle b,\, a^{i-1}\, e \right\rangle \right) \right).$$

We compute the coefficients of the stability polynomials associated with the Butcher matrix `a` and the weights b_1 and b_2 of the sub-methods of `DOPRI78`:

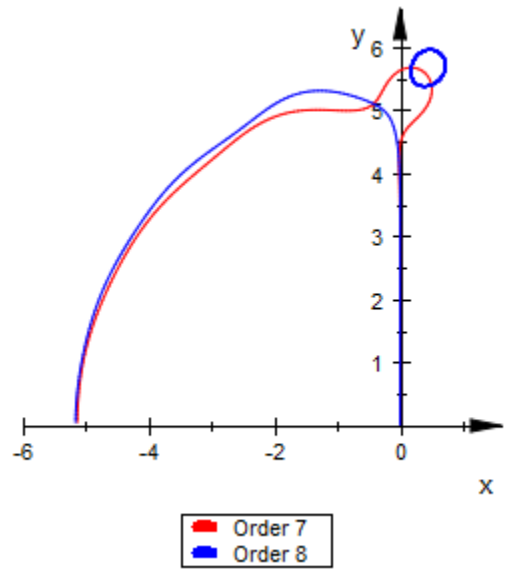
```
[s, c, a, b1, b2, order1, order2] := numeric::butcher(DOPRI78):
e := matrix([1 $ s]):
a := float(matrix(a)):
b1 := linalg::transpose(float(matrix(b1))):
b2 := linalg::transpose(float(matrix(b2))):
for i from 1 to s do
    c1[i] := (b1*a^(i-1)*e)[1, 1];
    c2[i] := (b2*a^(i-1)*e)[1, 1];
end_for:
```

We define the stability polynomials:

```
z := x + I*y:
p1 := 1 + _plus(c1[i]*z^i $ i = 1..s):
p2 := 1 + _plus(c2[i]*z^i $ i = 1..s):
```

The boundary of the stability region $\{z \in \mathbb{C} \mid |p(z)| < 1\}$ is the curve defined by $|p(z)| = 1$. We plot these implicit curves associated with the stability polynomials $p_1(z)$ and $p_2(z)$ defined above:

```
plot(plot::Implicit2d(abs(p1) = 1, x = -6..1, y = 0..6,
                      Color = RGB::Red, Legend = "Order 7"),
     plot::Implicit2d(abs(p2) = 1, x = -6..1, y = 0..6,
                      Color = RGB::Blue, Legend = "Order 8"),
     Scaling = Constrained):
```




```
delete s, c, a, b1, b2, order1, order2, e, c1, c2, z, p1, p2:
```

Parameters

s

The number of stages of the Gauss method: a positive integer

digits

The number of significant digits with which the Butcher data of the methods `GAUSS(s)` are computed. The default value for `digits` is the current value of the environment variable `DIGITS`. This argument is only relevant for the Gauss methods.

Return Values

A list `[s, c, a, b1, b2, order1, order2]` is returned.

Algorithms

The Butcher parameters provided in this original paper consist of rational *approximations* of solutions of the order equations of Runge-Kutta systems. The parameters provided by `numeric::butcher` are *exact* rational solutions of the order equations. The approximations given by Prince and Dormand coincide with the MuPAD exact values through 16 decimal digits.

References

J.C. Butcher: *The Numerical Analysis of Ordinary Differential Equations*, Wiley, Chichester (1987).

E. Hairer, S.P. Norsett and G. Wanner: *Solving Ordinary Differential Equations I*, Springer, Berlin (1993).

The methods `DOPRI87` and `DOPRI78` correspond to the method `RK8(7)13M` published in:

P.J. Prince and J.R.Dormand: *High order embedded Runge-Kutta formulae*, Journal of Computational and Applied Mathematics 7(1), 1981.

See Also

MuPAD Functions

`numeric::odesolve` | `numeric::odesolve2` | `numeric::odesolveGeometric`

numeric::complexRound

Round a complex number towards the real or imaginary axis

Syntax

```
numeric::complexRound(z, <eps>)
```

Description

`numeric::complexRound(z)` discards small real or imaginary parts of complex floating-point numbers `z`.

If the real part of z satisfies $\Re(z) < eps\ |z|$, then it is replaced by zero and $\Im(z)$ i is returned.

If the imaginary part of z satisfies $\Im(z) < eps\ |z|$, then it is replaced by zero and $\Re(z)$ is returned.

With the default of `eps=10^-DIGITS`, this rounding changes a complex floating-point number by less than the relative standard precision.

Only precisions `eps >= 10^-DIGITS` are accepted.

Numerical expressions such as `eps` = $\pi\ \sqrt{2}\ 10^{-10}$ etc. are accepted as `eps`.

This function removes small real or imaginary parts of complex floating-points numbers generated by numerical roundoff. It is used to simplify the floating-point output of `numeric::fsolve`, `numeric::polyroots`, `numeric::polysysroots` and `numeric::sum`.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`.

Examples

Example 1

Exact numbers are not changed:

```
numeric::complexRound(2 + I/10^20)
```

$$2 + \frac{i}{100000000000000000000}$$

Also the following number has an exact imaginary part and is not rounded:

```
numeric::complexRound(2.0 + sqrt(2)*I/10^20)
```

$$2.0 + \frac{\sqrt{2}\,i}{100000000000000000000}$$

Rounding occurs for complex floats, if this does not change its value significantly:

```
numeric::complexRound(1.0 + 2.0*I/10^10),
numeric::complexRound(1.0 + 2.0*I/10^11)
```

$$1.0 + 0.0000000002\,i,\ 1.0$$

Note that rounding is based on relative precision, i.e., only the ratio of real and imaginary parts is relevant:

```
numeric::complexRound((1.0 + 2.0*I)/10^100)
```

$$1.0\ 10^{-100} + 2.0\ 10^{-100}\,i$$

```
numeric::complexRound((1.0 + 1.0/10^11*I)/10^100)
```

$$1.0\ 10^{-100}$$

The relative precision for rounding may be reduced by the optional parameter `eps`:

```
numeric::complexRound(2.0/10^10 + I),
```

```
numeric::complexRound(2.0/10^10 + I, PI/10^5)
```

$$0.0000000002 + 1.0\,i,\ 1.0\,i$$

Parameters

z

An arbitrary MuPAD object

eps

A real number no less than $\frac{1}{10^{DIGITS}}$

Return Values

If `z` is a complex floating-point number, then a real or complex floating-point number is returned. For all other types, `z` is returned unchanged.

See Also

MuPAD Functions

`ceil` | `floor` | `frac` | `round` | `trunc`

numeric::cubicSpline

Interpolation by cubic splines

Syntax

```
numeric::cubicSpline([x₀, y₀], [x₁, y₁], …, <BoundaryCondition>, <Symbolic>, <NoWarning>
numeric::cubicSpline([x₀, x₁, …], [y₀, y₁, …], <BoundaryCondition>, <Symbolic>, <NoWarni
numeric::cubicSpline([[x₀, x₁, …], [y₀, y₁, …]], <BoundaryCondition>, <Symbolic>, <NoWar
```

Description

`numeric::cubicSpline([x₀, y₀], [x₁, y₁], …)` returns the cubic spline function interpolating a sequence of coordinate pairs $[x_i, y_i]$.

The call `S := numeric::cubicSpline([x₀, y₀], …, [xₙ, yₙ])` yields the cubic spline function S interpolating the data $[x_0, y_0]$, …, $[x_n, y_n]$, i.e., $S(x_i) = y_i$ for $i = 0, \ldots, n$. The spline function is a piecewise polynomial of degree ≤ 3 on the intervals $(-\infty, x_1]$, $[x_1, x_2]$, …, $[x_{n-1}, \infty)$. S and its first two derivatives S', S'' are continuous at the points $x_1, \ldots, x_{n-1}$. Note that S extends the polynomial representation on $[x_0, x_1]$, $[x_{n-1}, x_n]$ to $(-\infty, x_0]$ and $[x_{n-1}, \infty)$, respectively.

By default, `NotAKnot` boundary conditions are assumed, i.e., the third derivative S''' is continuous at the points x_1 and x_{n-1}. With this boundary condition, S is a cubic polynomial on the intervals $(-\infty, x_2]$ and $[x_{n-2}, \infty)$.

By default, all input data are converted to floating-point numbers. This conversion may be suppressed by the option `Symbolic`.

Without the option `Symbolic`, the abscissae x_i must be numerical real values in ascending order. If these data are not ordered, then `numeric::cubicSpline` reorders the abscissae internally, issuing a warning. The warning may be switched off by the option `NoWarning`.

The function S returned by `numeric::cubicSpline` may be called with one, two or three arguments:

- The call `S(z)` returns an explicit expression or a number, if `z` is a real number. Otherwise, the symbolic call `S(z)` is returned.
- The call `S(z, [k])` with a nonnegative integer `k` returns the k-th derivative of S. Cf. "Example 4" on page 19-17. For $k > 3$, zero is returned for any z.
- The call `S(z, i)` is meant for symbolic arguments z. The argument i must be an integer. Internally, z is assumed to satisfy $x_i \le z < x_{i+1}$, and `S(z, i)` returns the polynomial expression in z representing the spline function on this interval.
- The call `S(z, i, [k])` with an integer i and a nonnegative integer k returns the polynomial representation of the k-th derivative of the spline function on the interval $x_i \le z < x_{i+1}$.

If `S` is generated with symbolic abscissae x_i (necessarily using the option `Symbolic`), then the call `S(z)` with numerical `z` leads to an error. The call `S(z, i)` must be used for symbolic abscissae!

Note: Note that the interpolation of 2 points (x_0, y_0), (x_1, y_1) must be specified by `numeric::cubicSpline( [x0, y0] , [x1, y1] )`, not by `numeric::cubicSpline( [x0, x1] , [y0, y1] )`!

Examples

Example 1

We demonstrate some calls with numerical input data:

```
data := [i, sin(i*PI/20)] $ i= 0..40:
S1 := numeric::cubicSpline(data):
S2 := numeric::cubicSpline(data, Natural):
S3 := numeric::cubicSpline(data, Periodic):
S4 := numeric::cubicSpline(data, Complete = [3, PI]):
```

At the abscissae, the corresponding input data are reproduced:

```
float(data[6][2]), S1(5), S2(5), S3(5), S4(5)
```

0.7071067812, 0.7071067812, 0.7071067812, 0.7071067812, 0.7071067812

0.7071067812, 0.7071067812, 0.7071067812, 0.7071067812, 0.7071067812

Interpolation between the abscissae depends on the boundary condition:

```
S1(4.5), S2(4.5), S3(4.5), S4(4.5)
```

0.6494470263, 0.6494470123, 0.6494470123, 0.6517696766

These are the cubic polynomials in z defining the spline on the interval $x_0 = 0 \le z < x_1 = 1$:

```
expand(S1(z, 0)); expand(S2(z, 0));
expand(S3(z, 0)); expand(S4(z, 0))
```

$$-0.000632116114\,z^3 - 0.00002961951081\,z^2 + 0.1570962007\,z$$

$$0.1570790998\,z - 0.0006446347923\,z^3$$

$$-0.0006446347923\,z^3 + 1.270549421\,10^{-21}\,z^2 + 0.1570790998\,z$$

$$2.080517906\,z^3 - 4.924083441\,z^2 + 3.0\,z$$

$$-0.000632116114\,z^3 - 0.00002961951081\,z^2 + 0.1570962007\,z$$

$$0.1570790998\,z - 0.0006446347923\,z^3$$

$$-0.0006446347923\,z^3 + 1.270549421\,10^{-21}\,z^2 + 0.1570790998\,z$$

$$2.080517906\,z^3 - 4.924083441\,z^2 + 3.0\,z$$

```
delete data, S1, S2, S3, S4:
```

Example 2

We demonstrate some calls with symbolic data:

```
S := numeric::cubicSpline([i, y.i] $ i = 0..3):
S(1/2)
```

$$0.3125\ y0 + 0.9375\ y1 - 0.3125\ y2 + 0.0625\ y3$$

This is the cubic polynomial in z defining the spline on the interval $x_0 = 0 \le z < x_1 = 1$:

```
S(z, 0)
```

$$y0 - z\,(1.833333333\ y0 - 3.0\ y1 + 1.5\ y2 - 0.3333333333\ y3 + z\,(2.5\ y1 - 1.0\ y0 - 2.0\ y2 + 0.5\ y3 + z\,(0.1666666667\ y0 - 0.5\ y1 + 0.5\ y2 - 0.1666666667\ y3)))$$

With the option `Symbolic`, exact arithmetic is used:

```
S := numeric::cubicSpline([i, y.i] $ i = 0..3, Symbolic):
S(1/2)
```

$$\frac{5\ y0}{16} + \frac{15\ y1}{16} - \frac{5\ y2}{16} + \frac{y3}{16}$$

Also symbolic boundary data are accepted:

```
S := numeric::cubicSpline([i, exp(i)] $ i = 0..10,
                          Complete = [a, b]):
S(0.1)
```

$$0.08341154273\ a + 0.00000005947817812\ b + 1.020064753$$

```
S := numeric::cubicSpline([0, y0], [1, y1], [2, y2],
                          Symbolic, Complete = [a, 5]):
collect(S(z, 0), z)
```

$$\left(\frac{3\ a}{4} + \frac{5\ y0}{4} - 2\ y1 + \frac{3\ y2}{4} - \frac{5}{4}\right) z^3 + \left(3\ y1 - \frac{9\ y0}{4} - \frac{7\ a}{4} - \frac{3\ y2}{4} + \frac{5}{4}\right) z^2 + a\ z + y0$$

$$\left(\frac{3\ a}{4} + \frac{5\ y0}{4} - 2\ y1 + \frac{3\ y2}{4} - \frac{5}{4}\right) z^3 + \left(3\ y1 - \frac{9\ y0}{4} - \frac{7\ a}{4} - \frac{3\ y2}{4} + \frac{5}{4}\right) z^2 + a\ z + y0$$

```
delete S:
```

Example 3

We demonstrate the use of symbolic abscissae. Here the option `Symbolic` is mandatory.

```
S := numeric::cubicSpline([x.i, y.i] $ i = 0..2, Symbolic):
```

The spline function `S` can only be called with 2 arguments. This is the cubic polynomial in `z` defining the spline on the interval $x_0 \le z < x_1$:

```
S(z, 0)
```

$$\mathrm{y0} + (\mathrm{x0} - z)\left((\mathrm{x0} - z)\left(\frac{\mathrm{y0} - \mathrm{y1}}{(\mathrm{x0} - \mathrm{x1})\,(\mathrm{x0} - \mathrm{x2})} - \frac{\mathrm{y1} - \mathrm{y2}}{(\mathrm{x0} - \mathrm{x2})\,(\mathrm{x1} - \mathrm{x2})}\right) + \frac{(\mathrm{y0} - \mathrm{y1})\,(\mathrm{x1} - 2\,\mathrm{x0} + \mathrm{x2})}{(\mathrm{x0} - \mathrm{x1})\,(\mathrm{x0} - \mathrm{x2})} + \frac{(\mathrm{x0} - \mathrm{x1})\,(\mathrm{y1} - \mathrm{y2})}{(\mathrm{x0} - \mathrm{x2})\,(\mathrm{x1} - \mathrm{x2})}\right)$$

$$\mathrm{y0} + (\mathrm{x0} - z)\left((\mathrm{x0} - z)\left(\frac{\mathrm{y0} - \mathrm{y1}}{(\mathrm{x0} - \mathrm{x1})\,(\mathrm{x0} - \mathrm{x2})} - \frac{\mathrm{y1} - \mathrm{y2}}{(\mathrm{x0} - \mathrm{x2})\,(\mathrm{x1} - \mathrm{x2})}\right) + \frac{(\mathrm{y0} - \mathrm{y1})\,(\mathrm{x1} - 2\,\mathrm{x0} + \mathrm{x2})}{(\mathrm{x0} - \mathrm{x1})\,(\mathrm{x0} - \mathrm{x2})} + \frac{(\mathrm{x0} - \mathrm{x1})\,(\mathrm{y1} - \mathrm{y2})}{(\mathrm{x0} - \mathrm{x2})\,(\mathrm{x1} - \mathrm{x2})}\right)$$

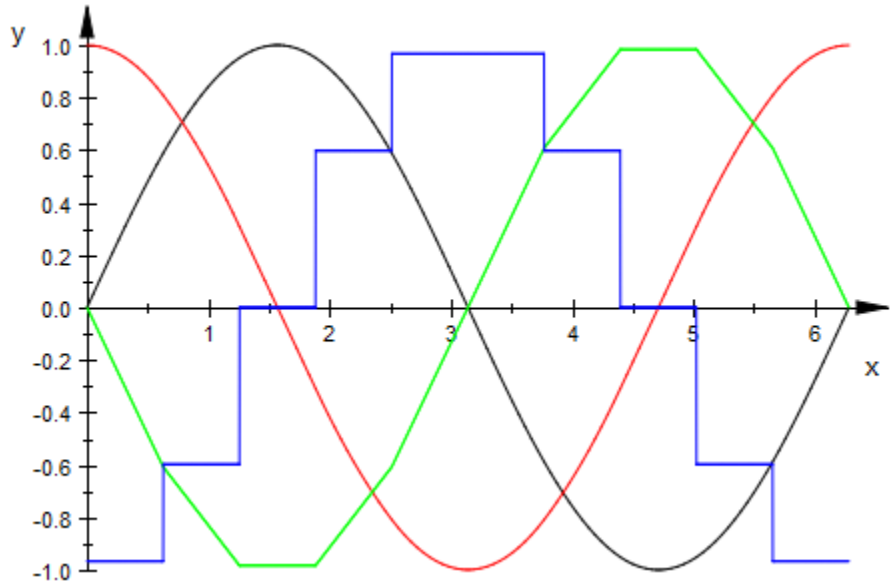
```
delete S:
```

Example 4

We plot a spline function together with its first three derivatives. The spline approximates the function $sin(x)$:

```
n := 10:
x := array(0..n, [i/n*2*PI $ i = 0..n]):
S := numeric::cubicSpline([x[i], sin(x[i])] $ i = 0..n, Natural):
delete x:
plot(
   plot::Function2d(S(x), x = 0..2*PI, Color = RGB::Black),
   plot::Function2d(S(x, [1]), x = 0..2*PI, Color = RGB::Red),
```

```
    plot::Function2d(S(x, [2]), x = 0..2*PI, Color = RGB::Green,
                     Mesh = 1000),
    plot::Function2d(S(x, [3]), x = 0..2*PI, Color = RGB::Blue,
                     Mesh = 1000)
)
```




```
delete n, S:
```

Example 5

We demonstrate how to generate a phase plot of the differential equation $x''(t) + x(t)^3 = \sin(t)$, with initial conditions $x(0) = x'(0) = 0$. First, we use `numeric::odesolve` to compute a numerical mesh of solution points $[x_i, y_i] = [x(t_i), x'(t_i)]$ with $n + 1$ equidistant time nodes $t_0, \ldots, t_n$ in the interval [0, 20]:

```
DIGITS := 4: n := 100:
for i from 0 to n do t[i] := 20/n*i: end_for:
f := (t, x) -> [x[2], sin(t) - x[1]^3]:
x[0] := 0: y[0] := 0:
for i from 1 to n do
  [x[i], y[i]] :=
```

```
  numeric::odesolve(t[i-1]..t[i], f, [x[i-1], y[i-1]]):
end_for:
```

The mesh of the $(x(t), x'(t))$ phase plot consists of the following points:

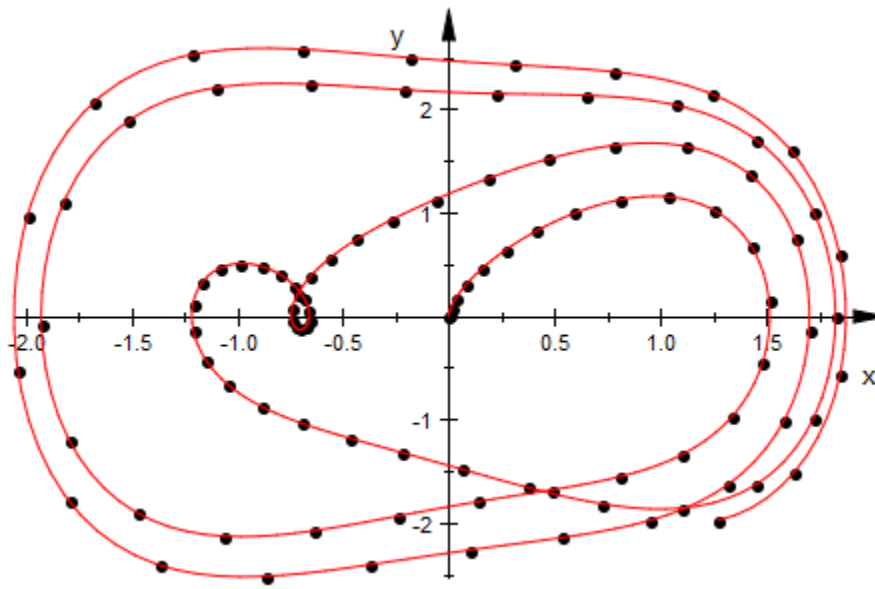
```
Plotpoints := [[x[i], y[i]] $ i = 0..n]:
```

We wish to connect these points by a spline curve. We define a spline interpoland $Sx(t)$ approximating the solution $x(t)$ by interpolating the data $[t_0, x_0]$, ..., $[t_n, x_n]$. A spline interpoland $Sy(t)$ approximating $x'(t)$ is obtained by interpolating the data $[t_0, y_0]$, ..., $[t_n, y_n]$:

```
Sx := numeric::cubicSpline([t[i], x[i]] $ i = 0..n):
Sy := numeric::cubicSpline([t[i], y[i]] $ i = 0..n):
```

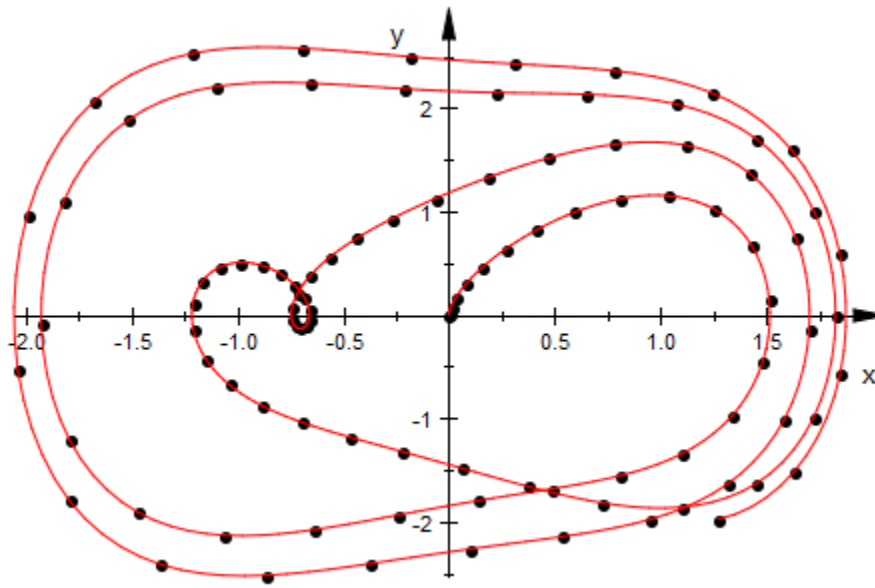
Finally, we plot the mesh points together with the interpolating spline curve:

```
plot(
     plot::PointList2d(Plotpoints, PointColor = RGB::Black),
     plot::Curve2d([Sx(z), Sy(z)], z = 0..20, Mesh = 5*(n - 1) + 1,
                   LineColor = RGB::Red)
    )
```



The functions `plot::Ode2d` and `plot::Ode3d` serve for displaying numerical solutions of ODEs. In fact, they are implemented as indicated by the previous commands. The following call produces the same plot:

```
plot(plot::Ode2d(
 [t[i] $ i = 0..n], f, [x[0], y[0]],
 [(t, x) -> [x[1], x[2]], Style = Points, Color = RGB::Black],
 [(t, x) -> [x[1], x[2]], Style = Splines, Color = RGB::Red])):
```




```
delete DIGITS, n, i, t, f, x, y, Plotpoints, Sx, Sy:
```

Parameters

x_0, x_1, …

Numerical real values in ascending order

y_0, y_1, …

Arbitrary expressions

BoundaryCondition

The type of the boundary condition: either `NotAKnot`, `Natural`, `Periodic`, or `Complete = [a, b]` with arbitrary arithmetical expressions `a, b`.

Options

Symbolic

With this option, no conversion of the input data to floating point numbers occurs.

Symbolic abscissae x_i are accepted.

The ordering $x_0 < x_1 < \ldots < x_n$ is assumed by `numeric::cubicSpline`. This ordering is not checked, even if the abscissae are numerical!

NoWarning

The x-values of the interpolation points must be in ascending order. If the input data violate this condition, the routine issues a warning and reorders the data automatically. With this option, the warning is switched off.

NotAKnot

With the default boundary condition `NotAKnot`, the third derivative S''' of the spline function is continuous at the points x_1 and x_{n-1}. With this boundary condition, S is a polynomial on the intervals $(-\infty, x_2]$ and $[x_{n-2}, \infty)$.

Natural

The boundary condition `Natural` produces a spline function S satisfying $S''(x_0) = S''(x_n) = 0$.

Periodic

The boundary condition `Periodic` produces a spline function S satisfying $S(x_0) = S(x_n)$, $S'(x_0) = S'(x_n)$, $S''(x_0) = S''(x_n)$. With this option, the input data y_0, y_n must coincide, otherwise an error is raised.

`Complete`

Option, specified as `Complete = [a, b]`

The boundary condition `Complete = [a, b]` produces a spline function S satisfying $S'(x_0) = a$, $S'(x_n) = b$. Symbolic data `a, b` are accepted.

Return Values

Spline interpoland: a MuPAD procedure.

See Also

MuPAD Functions

`interpolate` | `numeric::cubicSpline2d`

numeric::cubicSpline2d

Interpolation by 2-dimensional bi-cubic splines

Syntax

```
numeric::cubicSpline2d([x0, x1, …, xn], [y0, y1, …, ym], z, <[xBC, yBC]>, <Symbolic>)
```

Description

`numeric::cubicSpline2d([ x0, x1, ...], [ y0, y1, ...], z)` returns the bi-cubic spline function interpolating data $z_{i,j}$ over a rectangular mesh (x_i, y_j).

The call `S := numeric::cubicSpline2d([x0, …, xn], [y0, …, yn], z, Option)` yields the cubic spline function S interpolating the data $(x_i, y_j, z_{i,j})$, i.e, $S(x_i, y_j) = z_{i,j}$ for $i = 0, \ldots, n$, $j = 0, \ldots, m$. The spline function is a piecewise bi-cubic polynomial: on the 'patch'

$$p_{i,j} = \left\{ (x, y) \;\middle|\; x_i \le x < x_{i+1}, y_j \le y < y_{j+1} \right\},$$

it has the representation

$$S(x, y) = \sum_{u=0}^{3} \left(\sum_{v=0}^{3} a_{i,j}^{(u,v)} (x - x_i)^u \left(y - y_j\right)^v \right)$$

with suitable coefficients $a_{i,j}^{(u,v)}$ depending on the patch. The spline S and its partial derivatives S_x, S_y, S_{xx}, S_{xy}, S_{yy}, S_{xxy}, S_{xyy}, S_{xxyy} are continous functions over the entire x, y plane. In the x-direction, `S` extends the polynomial representation on the boundary patches $[x_0, x_1]$ and $[x_{n-1}, x_n]$ to $(-\infty, x_1]$ and $[x_{n-1}, \infty)$, respectively. The same holds with respect to the y-direction.

By default, `NotAKnot` boundary conditions are assumed, i.e., the partial derivatives S_{xxx}, S_{yyy}, …, S_{xxxyyy}, are continuous at the points with x-coordinates x_1 and x_{n-1} or y-coordinates y_1 and y_{m-1}.

By default, all input data are converted to floating-point numbers. This conversion may be suppressed by the option `Symbolic`.

Without the option `Symbolic`, the abscissae x_i, y_j must be numerical real values in ascending order. If these data are not ordered, then `numeric::cubicSpline2d` reorders the abscissae internally, issuing a warning.

The function `S` returned by `numeric::cubicSpline2d` may be called with two, three, four, or five arguments, respectively:

- The call `S(x, y)` returns an arithmetical expression if x and y are numerical expressions. A float is returned if either x or y is a float and all parameters involved can be converted to floats.

  If either x or y contains symbolic objects, the symbolic call `S(x, y)` is returned.

- The call `S(x, y, [u, v])` with nonnegative integers `u`, `v` returns the partial derivative $\frac{\partial^v}{\partial y^v}\frac{\partial^u}{\partial x^u}S$ of the spline. If either x or y contain symbolic objects, the symbolic call `S(x, y, [u, v])` is returned. The result is 0 if either $u > 3$ or $v > 3$. The calls `S(x, y, [0, 0])` and `S(x, y)` are equivalent.

- The call `S(x, y, i, j)` with nonnegative integers `i`, `j` returns the polynomial representation of the spline on the patch $p_{i,j}$. Here, x and y may be arbitrary numerical or symbolic arithmetical expressions. Internally, (x, y) are assumed to lie in the patch $p_{i,j}$.

- The call `S(x, y, i, j, [u, v])` with nonnegative integers `i`, `j`, `u`, `v` returns the polynomial representation of the partial derivatives of the spline function. In this call, x and y may be arbitrary numerical or symbolic arithmetical expressions which are assumed to lie in the patch $p_{i,j}$. The result is 0 if either $u > 3$ or $v > 3$. The calls `S(x, y, i, j, [0, 0])` and `S(x, y, i, j)` are equivalent.

If `S` is generated with symbolic abscissae x_i, y_j (necessarily using the option `Symbolic`), the call `S(x, y, [ u , v ])` is returned symbolically. The call `S(x, y, i, j, [ u , v ])` must be used for symbolic abscissae!

Examples

Example 1

We demonstrate some calls with numerical input data. The function $f(x, y) = sin(2\,\pi\,(x + y))$ with $0 \le x \le 1$, $0 \le y \le 1$ is to be interpolated by $n + 1 = 11$ equidistant points in the x-direction and $m + 1 = 13$ equidistant points in the y-direction:

```
f := (x, y) -> sin((x + y)*2*PI):
n := 10: x := [i/n $ i = 0..n]:
m := 12: y := [j/m $ j = 0..m]:
z := array(0..n, 0..m, [[f(i/n, j/m) $ j = 0..m] $ i = 0..n]):
S1 := numeric::cubicSpline2d(x, y, z, [NotAKnot, NotAKnot]):
S2 := numeric::cubicSpline2d(x, y, z, [Natural, Natural]):
S3 := numeric::cubicSpline2d(x, y, z, [NotAKnot, Periodic]):
```

We consider `Complete` boundary conditions in the y-direction. They consist of the values $f_y(x_i, y_0) = f_y(x_i, 0) = 2\,\pi\,cos(2\,\pi\,x_i)$ and $f_y(x_i, y_m) = f_y(x_i, 1) = 2\,\pi\,cos(2\,\pi\,x_i)$:

```
ybc:= [[2*PI*cos(2*PI*i/n) $ i = 0..n],
       [2*PI*cos(2*PI*i/n) $ i = 0..n]]:
S4 := numeric::cubicSpline2d(x, y, z, [Periodic, Complete = ybc]):
```

At the mesh points (x_i, y_j), the input data $z_{i,j}$ are reproduced:

```
x := 4/n: y := 8/m:
float(f(x, y)), S1(x, y), S2(x, y), S3(x, y), S4(x, y)
```

$$0.4067366431,\ 0.4067366431,\ 0.4067366431,\ 0.4067366431,\ 0.4067366431$$

Interpolation between the mesh points depends on the boundary condition:

```
x := 0.92: y:= 0.55: S1(x, y), S2(x, y), S3(x, y), S4(x, y)
```

$$0.1879484554,\ 0.1897776244,\ 0.1879457306,\ 0.1873726747$$

The approximation of the function value $f(0.92, 0.55)$ is good for the `NotAKnot`, `Periodic`, and `Complete` boundary conditions. The `Natural` boundary conditions are

less appropriate because the second partial derivatives of the function *f* do not vanish at the boundaries. Consequently, the approximation error of `S2` is larger than the other approximation errors:

```
z := float(f(x, y)):
S1(x, y) - z, S2(x, y) - z, S3(x, y) - z, S4(x, y) - z
```

$$0.0005671407743,\ 0.002396309797,\ 0.0005644160155,\ -0.00000863986878$$

This is the bi-cubic polynomial in `X`, `Y` defining the spline `S1` on the patch $x_0 = 0 \le X < x_1 = \frac{1}{n}, y_3 = \frac{3}{m} \le Y \le y_4 = \frac{4}{m}$:

```
expand(S1(X, Y, 0, 3))
```

$$-804.8629918\,X^3\,Y^3 + 6.210308852\,X^3\,Y^2 + 323.2124519\,X^3\,Y - 39.06086335\,X^3$$
$$-373.407714\,X^2\,Y^3 + 770.1743348\,X^2\,Y^2 - 297.137889\,X^2\,Y + 7.665899226\,X^2$$
$$+260.7571691\,X\,Y^3 - 199.7700052\,X\,Y^2 + 10.51686235\,X\,Y + 6.008422078\,X$$
$$+10.65805697\,Y^3 - 28.15535765\,Y^2 + 12.07773495\,Y - 0.426256024$$

```
delete f, n, m, ybc, x, y, z, S1, S2, S3, S4:
```

Example 2

We demonstrate some calls with symbolic data. With the option `Symbolic`, exact arithmetic is used:

```
S := numeric::cubicSpline2d(
       [i $ i = 0..3],
       [j $ j = 0..4],
       array(0..3, 0..4, [[z.i.j $ j = 0..4] $ i = 0..3]),
       Symbolic
    ):

S(1/2, 3/2)
```

$$\frac{5\,z01}{32} - \frac{15\,z00}{1024} + \frac{105\,z02}{512} - \frac{5\,z03}{128} + \frac{5\,z04}{1024} - \frac{45\,z10}{1024} + \frac{15\,z11}{32} + \frac{315\,z12}{512} - \frac{15\,z13}{128} + \frac{15\,z14}{1024}$$

$$+ \frac{15\,z20}{1024} - \frac{5\,z21}{32} - \frac{105\,z22}{512} + \frac{5\,z23}{128} - \frac{5\,z24}{1024} - \frac{3\,z30}{1024} + \frac{z31}{32} + \frac{21\,z32}{512} - \frac{z33}{128} + \frac{z34}{1024}$$

This is the bi-cubic polynomial in X, Y defining the spline with $x_0 = 0 \le X \le x_1 = 1$, $y_1 = 1 \le Y \le y_2 = 2$:

```
expand(S(X, Y, 0, 1))
```

$$z00 + X^2\,z00 - \frac{X^3\,z00}{6} - \frac{5\,X^2\,z10}{2} + \cdots - \frac{X\,Y\,z34}{36}$$

```
z00 - (11*X*z00)/6 + 3*X*z10 - (3*X*z20)/2 + (X*z30)/3 - (23*Y*z00)/12 + (10*Y*z01)/3
- 2*Y*z02 + (2*Y*z03)/3 - (Y*z04)/12 + X^2*z00 - (X^3*z00)/6 - (5*X^2*z10)/2 +
(X^3*z10)/2 + 2*X^2*z20 - (X^3*z20)/2 - (X^2*z30)/2 + (X^3*z30)/6 + (9*Y^2*z00)/8 -
3*Y^2*z01 - (5*Y^3*z00)/24 + (11*Y^2*z02)/4 + (2*Y^3*z01)/3 - Y^2*z03 - (3*Y^3*z02)/4
+ (Y^2*z04)/8 + (Y^3*z03)/3 - (Y^3*z04)/24 + (9*X^2*Y^2*z00)/8 - 3*X^2*Y^2*z01 -
(5*X^2*Y^3*z00)/24 - (3*X^3*Y^2*z00)/16 + (11*X^2*Y^2*z02)/4 + (2*X^2*Y^3*z01)/3
+ (X^3*Y^2*z01)/2 + (5*X^3*Y^3*z00)/144 - X^2*Y^2*z03 - (3*X^2*Y^3*z02)/4
- (11*X^3*Y^2*z02)/24 - (X^3*Y^3*z01)/9 + (X^2*Y^2*z04)/8 + (X^2*Y^3*z03)/3
+ (X^3*Y^2*z03)/6 + (X^3*Y^3*z02)/8 - (X^2*Y^3*z04)/24 - (X^3*Y^2*z04)/48 -
(X^3*Y^3*z03)/18 + (X^3*Y^3*z04)/144 - (45*X^2*Y^2*z10)/16 + (15*X^2*Y^2*z11)/2 +
(25*X^2*Y^3*z10)/48 + (9*X^3*Y^2*z10)/16 - (55*X^2*Y^2*z12)/8 - (5*X^2*Y^3*z11)/3
- (3*X^3*Y^2*z11)/2 - (5*X^3*Y^3*z10)/48 + (5*X^2*Y^2*z13)/2 + (15*X^2*Y^3*z12)/8
+ (11*X^3*Y^2*z12)/8 + (X^3*Y^3*z11)/3 - (5*X^2*Y^2*z14)/16 - (5*X^2*Y^3*z13)/6
- (X^3*Y^2*z13)/2 - (3*X^3*Y^3*z12)/8 + (5*X^2*Y^3*z14)/48 + (X^3*Y^2*z14)/16
+ (X^3*Y^3*z13)/6 - (X^3*Y^3*z14)/48 + (9*X^2*Y^2*z20)/4 - 6*X^2*Y^2*z21 -
(5*X^2*Y^3*z20)/12 - (9*X^3*Y^2*z20)/16 + (11*X^2*Y^2*z22)/2 + (4*X^2*Y^3*z21)/3
+ (3*X^3*Y^2*z21)/2 + (5*X^3*Y^3*z20)/48 - 2*X^2*Y^2*z23 - (3*X^2*Y^3*z22)/2
- (11*X^3*Y^2*z22)/8 - (X^3*Y^3*z21)/3 + (X^2*Y^2*z24)/4 + (2*X^2*Y^3*z23)/3
+ (X^3*Y^2*z23)/2 + (3*X^3*Y^3*z22)/8 - (X^2*Y^3*z24)/12 - (X^3*Y^2*z24)/16 -
(X^3*Y^3*z23)/6 + (X^3*Y^3*z24)/48 - (9*X^2*Y^2*z30)/16 + (3*X^2*Y^2*z31)/2 +
(5*X^2*Y^3*z30)/48 + (3*X^3*Y^2*z30)/16 - (11*X^2*Y^2*z32)/8 - (X^2*Y^3*z31)/3
- (X^3*Y^2*z31)/2 - (5*X^3*Y^3*z30)/144 + (X^2*Y^2*z33)/2 + (3*X^2*Y^3*z32)/8
+ (11*X^3*Y^2*z32)/24 + (X^3*Y^3*z31)/9 - (X^2*Y^2*z34)/16 - (X^2*Y^3*z33)/6
- (X^3*Y^2*z33)/6 - (X^3*Y^3*z32)/8 + (X^2*Y^3*z34)/48 + (X^3*Y^2*z34)/48 +
(X^3*Y^3*z33)/18 - (X^3*Y^3*z34)/144 + (253*X*Y*z00)/72 - (55*X*Y*z01)/9 +
```

```
(11*X*Y*z02)/3 - (11*X*Y*z03)/9 + (11*X*Y*z04)/72 - (23*X*Y*z10)/4 + 10*X*Y*z11
- 6*X*Y*z12 + 2*X*Y*z13 - (X*Y*z14)/4 + (23*X*Y*z20)/8 - 5*X*Y*z21 + 3*X*Y*z22
- X*Y*z23 + (X*Y*z24)/8 - (23*X*Y*z30)/36 + (10*X*Y*z31)/9 - (2*X*Y*z32)/3
+ (2*X*Y*z33)/9 - (X*Y*z34)/36 - (33*X*Y^2*z00)/16 - (23*X^2*Y*z00)/12 +
(11*X*Y^2*z01)/2 + (55*X*Y^3*z00)/144 + (10*X^2*Y*z01)/3 + (23*X^3*Y*z00)/72
- (121*X*Y^2*z02)/24 - (11*X*Y^3*z01)/9 - 2*X^2*Y*z02 - (5*X^3*Y*z01)/9 +
(11*X*Y^2*z03)/6 + (11*X*Y^3*z02)/8 + (2*X^2*Y*z03)/3 + (X^3*Y*z02)/3 -
(11*X*Y^2*z04)/48 - (11*X*Y^3*z03)/18 - (X^2*Y*z04)/12 - (X^3*Y*z03)/9 +
(11*X*Y^3*z04)/144 + (X^3*Y*z04)/72 + (27*X*Y^2*z10)/8 + (115*X^2*Y*z10)/24
- 9*X*Y^2*z11 - (5*X*Y^3*z10)/8 - (25*X^2*Y*z11)/3 - (23*X^3*Y*z10)/24 +
(33*X*Y^2*z12)/4 + 2*X*Y^3*z11 + 5*X^2*Y*z12 + (5*X^3*Y*z11)/3 - 3*X*Y^2*z13
- (9*X*Y^3*z12)/4 - (5*X^2*Y*z13)/3 - X^3*Y*z12 + (3*X*Y^2*z14)/8 + X*Y^3*z13
+ (5*X^2*Y*z14)/24 + (X^3*Y*z13)/3 - (X*Y^3*z14)/8 - (X^3*Y*z14)/24 -
(27*X*Y^2*z20)/16 - (23*X^2*Y*z20)/6 + (9*X*Y^2*z21)/2 + (5*X*Y^3*z20)/16 +
(20*X^2*Y*z21)/3 + (23*X^3*Y*z20)/24 - (33*X*Y^2*z22)/8 - X*Y^3*z21 - 4*X^2*Y*z22 -
(5*X^3*Y*z21)/3 + (3*X*Y^2*z23)/2 + (9*X*Y^3*z22)/8 + (4*X^2*Y*z23)/3 + X^3*Y*z22
- (3*X*Y^2*z24)/16 - (X*Y^3*z23)/2 - (X^2*Y*z24)/6 - (X^3*Y*z23)/3 + (X*Y^3*z24)/16 +
(X^3*Y*z24)/24 + (3*X*Y^2*z30)/8 + (23*X^2*Y*z30)/24 - X*Y^2*z31 - (5*X*Y^3*z30)/72
- (5*X^2*Y*z31)/3 - (23*X^3*Y*z30)/72 + (11*X*Y^2*z32)/12 + (2*X*Y^3*z31)/9 +
X^2*Y*z32 + (5*X^3*Y*z31)/9 - (X*Y^2*z33)/3 - (X*Y^3*z32)/4 - (X^2*Y*z33)/3 -
(X^3*Y*z32)/3 + (X*Y^2*z34)/24 + (X*Y^3*z33)/9 + (X^2*Y*z34)/24 + (X^3*Y*z33)/9 -
(X*Y^3*z34)/72 - (X^3*Y*z34)/72
```

```
delete S:
```

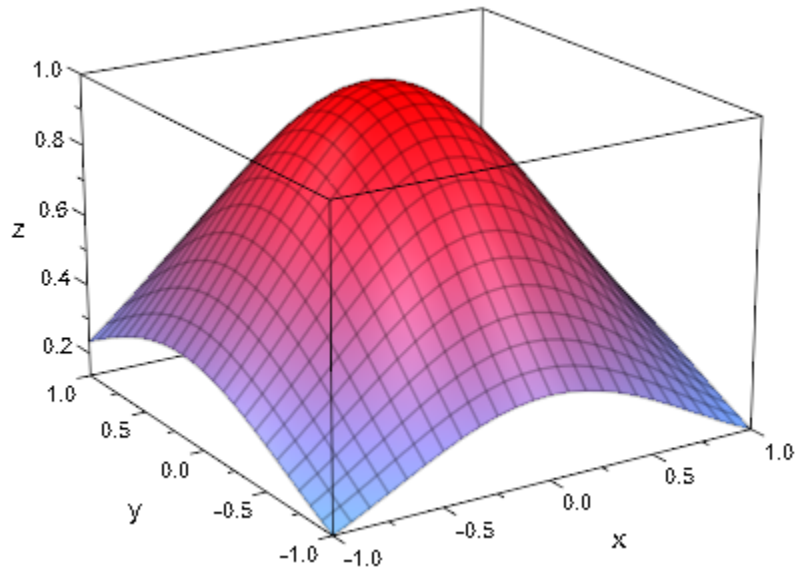
Example 3

We consider a spline interpolation of the function $f(x, y) = \frac{1}{e^{x^2+y^2}}$ with $-1 \le x \le 1$, $-1 \le y \le 1$:

```
n := 10: xmesh := [-1 + 2*i/n $ i = 0..n]:
m := 12: ymesh := [-1 + 2*j/n $ j = 0..m]:
f := (x, y) -> exp(-x^2 - y^2):
z := array(0..n, 0..m,
     [[f(-1 + 2*i/n, -1 + 2*j/m) $ j=0..m] $ i = 0..n]):
S := numeric::cubicSpline2d(xmesh, ymesh, z):
```

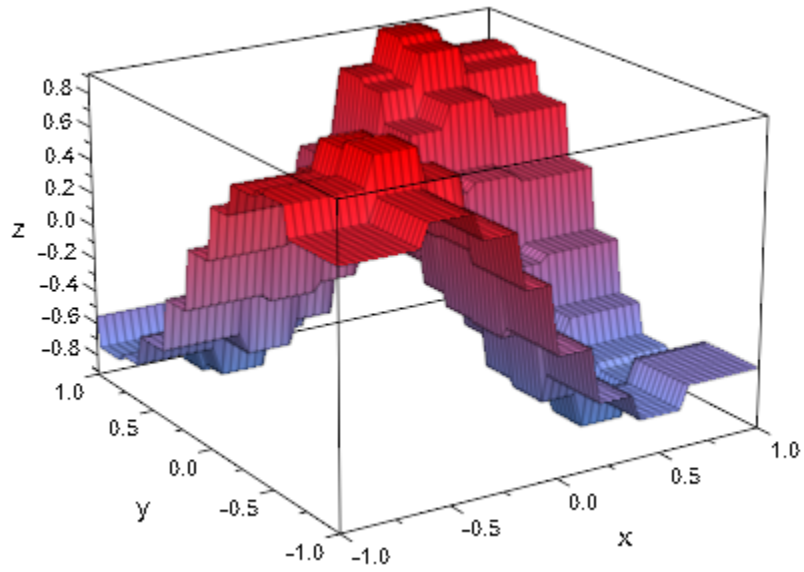
We plot the spline function $S(x, y)$:

```
plotfunc3d(S(x, y), x = -1 .. 1, y = -1 .. 1):
```



We plot the partial derivative $S_{xxxyyy}(x, y)$. It is constant on each patch with jumps at the boundaries of the patches. The renderer uses $[5\,n + 1, 5\,m + 1]$ mesh points: in each direction, 4 extra points between adjacent mesh points of the spline are used for the graphical representation:

```
plotfunc3d(S(x, y, [3, 3])/10, x = -1 .. 1, y = -1 .. 1,
           Mesh = [5*n + 1, 5*m+ 1])
```




```
delete n, xmesh, m, ymesh, f, z, S:
```

Example 4

We demonstrate the spline interpretation of a surface. We consider a sphere parametrized by spherical coordinates u, v with $0 \le u \le 2\,\pi$, $0 \le v \le \pi$:

$$x = \cos(u)\,\sin(v),\ y = \sin(u)\,\sin(v),\ z = \cos(v)$$

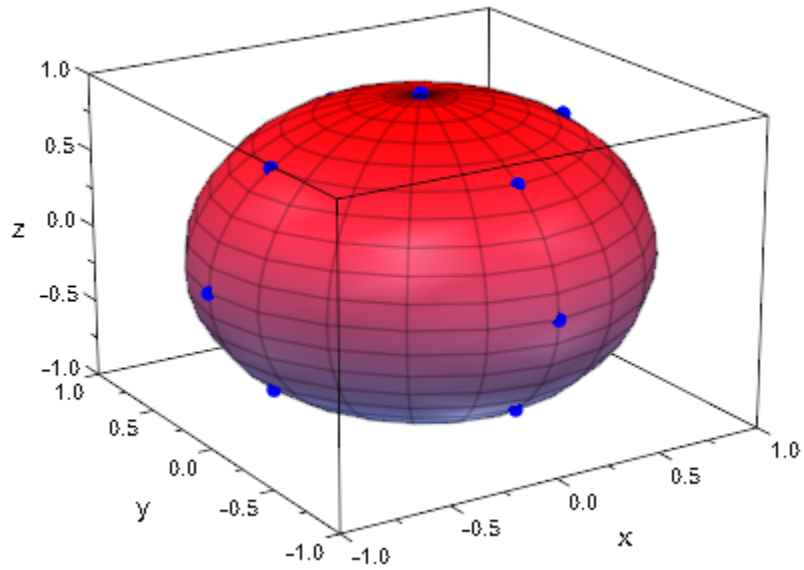
We interpolate the functions x, y, z over a rectangular mesh in the u-v-plane. Since x, y and (trivally) z are $2\,\pi$-periodic in u, we choose `Periodic` boundary conditions for u. For v, we choose `Complete` boundary conditions with boundary values of the first partial v-derivative fitting the parametrization:

```
x:= (u, v) -> cos(u)*sin(v): x_v := diff(x(u, v), v):
y:= (u, v) -> sin(u)*sin(v): y_v := diff(y(u, v), v):
z:= (u, v) -> cos(v): z_v := diff(z(u, v), v):
n := 4: umesh := [i*2*PI/n $ i = 0..n]:
m := 4: vmesh := [j*PI/m $ j = 0..m]:
```

```
vBC := Complete = [
    [subs(x_v, u = umesh[i], v = vmesh[1]) $ i = 1 .. n+1],
    [subs(x_v, u = umesh[i], v = vmesh[n + 1]) $ i = 1 .. n+1]]:
X := numeric::cubicSpline2d(umesh, vmesh,
  array(0..n, 0..m, [[x(i*2*PI/n, j*PI/m) $ j=0..m] $ i=0..n]),
  [Periodic, vBC]):

vBC := Complete = [
    [subs(y_v, u = umesh[i], v = vmesh[1]) $ i = 1 .. n+1],
    [subs(y_v, u = umesh[i], v = vmesh[n + 1]) $ i = 1 .. n+1]]:
Y := numeric::cubicSpline2d(umesh, vmesh,
   array(0..n, 0..m, [[y(i*2*PI/n, j*PI/m) $ j=0..m] $ i=0..n]),
   [Periodic, vBC]):

vBC := Complete = [
    [subs(z_v, u = umesh[i], v = vmesh[1]) $ i = 1 .. n+1],
    [subs(z_v, u = umesh[i], v = vmesh[n + 1]) $ i = 1 .. n+1]]:
Z := numeric::cubicSpline2d(umesh, vmesh,
   array(0..n, 0..m, [[z(i*2*PI/n, j*PI/m) $ j=0..m] $ i=0..n]),
   [Periodic, vBC]):
```

With only $(n + 1)\times(m + 1) = 5\times5$ mesh points, the spline surface yields a respectable approximation of a sphere. The interpolation nodes are added to the plot as blue points:

```
plot(
  plot::Surface([X(u, v), Y(u, v), Z(u, v)],
                u = 0..2*PI, v = 0..PI,
                Mesh = [5*n + 1, 5*m + 1],
                Color = RGB::Red),
  plot::Point3d(x(umesh[i], vmesh[j]),
               y(umesh[i], vmesh[j]),
               z(umesh[i], vmesh[j]),
               PointSize = 2*unit::mm,
               Color = RGB::Blue
              ) $ i = 1..n+1 $ j = 1..m+1
):
```




```
delete x, x_v, y, y_v, z, z_v, n, m,
       umesh, vmesh, vBC, X, Y, Z:
```

Parameters

x_0, x_1, …, x_n

The *x*-coordinates of the nodes: distinct numerical real values in ascending order

y_0, y_1, …, y_m

The *y*-coordinates of the nodes: distinct numerical real values in ascending order

z

The function values: an array of the form `array(0..n, 0..m, [...])` with numerical or symbolic arithmetical expressions.

xBC, yBC

The type of the boundary condition: the boundary condition in the *x*- or *y*-direction may be one of the flags `NotAKnot`, `Natural`, `Periodic` or `Complete = [...]`.

`Complete` boundary conditions consist of prescribed values for the derivatives S_x or S_y, respectively, along the mesh boundaries in the x- or y-direction, respectively. In the x-direction, these value may be passed in the form `Complete = [[a0, …, am], [b0, …, bm]]` with arbitrary numerical or symbolic arithmetical expressions `a0, …, bm`.

In the y-direction, these value may be passed in the form `Complete = [[a0, …, an], [b0, …, bn]]` with arbitrary numerical or symbolic arithmetical expressions `a0, …, bn`.

Options

Symbolic

With this option, no conversion of the input data to floating point numbers occurs.

Symbolic abscissae x_i, y_j are accepted.

The ordering $x_0 < x_1 < \ldots < x_n$, $y_0 < y_1 < \ldots < y_m$ is assumed. This ordering is not checked even if the node coordinates are numerical!

NotAKnot

With the default boundary condition `xBC = yBC = NotAKnot`, all partial derivatives of the spline function are continuous at the nodes with x-coordinates x_1 and x_{n-1} or y-coordinates y_1 and y_{m-1}, respectively. With this boundary condition, `S` is a polynomial on the union of the patches $p_{0,j}$, $p_{1,j}$ and $p_{n-2,j}$, $p_{n-1,j}$ or $p_{i,0}$, $p_{i,1}$ and $p_{i,m-2}$, $p_{i,m-1}$, respectively.

This boundary condition is recommended if no information on the behaviour of the data near the mesh boundaries is available.

Natural

The boundary condition `Natural` produces a spline function `S` with vanishing second partial derivatives at the boundary of the mesh.

This boundary condition is recommended if it is known that the data correspond to a surface with vanishing curvature near the mesh boundaries.

Periodic

The boundary condition `Periodic` produces a spline function `S` satisfying

$$S(x_0, y) = S(x_n, y),\ S'(x_0, y) = S'(x_n, y),\ S''(x_0, y) = S''(x_n, y)$$

Or

$$S(x, y_0) = S(x, y_m),\ S'(x, y_0) = S'(x, y_m),\ S''(x, y_0) = S''(x, y_m)\ ,$$

Respectively. With this option, the input data $z_{0,j}$, $z_{n,j}$, respectively $z_{i,0}$, $z_{i,m}$, must coincide. Otherwise, an error is raised.

This boundary condition is recommended if the interpolation is to represent a periodic function.

`Complete`

Option, specified as `Complete = [...]`

The `xBC` boundary condition `Complete = [[a0, …, am], [b0, …, bm]]` produces a spline function S satisfying $S_x(x_0, y_j) = a_j$, $S_x(x_n, y_j) = b_j$, $j = 0, \ldots, m$.

The `yBC` boundary condition `Complete = [[a0, …, an], [b0, …, bn]]` produces a spline function S satisfying $S_y(x_i, y_0) = a_i$, $S_y(x_i, y_m) = b_i$, $i = 0, \ldots, n$.

Symbolic data a_k, b_k are accepted.

This boundary condition is recommended if the data $z_{i,j}$ correpond to a function with known values of the first partial derivatives at the mesh boundaries.

Return Values

Spline function: a MuPAD procedure.

See Also

MuPAD Functions

`interpolate` | `numeric::cubicSpline`

numeric::det

Determinant of a matrix

Syntax

```
numeric::det(A, <mode>, <MinorExpansion>, <NoWarning>)
```

Description

`numeric::det(A)` returns the determinant of the matrix *A*.

Without the option `Symbolic`, all entries of `A` must be numerical. Numerical expressions such as e^{π}, $\sqrt{2}$ etc. are accepted and converted to floats. If symbolic entries are found in the matrix, `numeric::det` automatically switches to `Symbolic` issuing a warning.

The option `Symbolic` should be used if the matrix contains symbolic objects that cannot be converted to floating point numbers.

Note: Matrices `A` of a matrix domain such as `Dom::Matrix(...)` or `Dom::SquareMatrix(...)` are internally converted to arrays over expressions via `expr(A)`. Note that `det` must be used, when the determinant is to be computed over the component domain. See “Example 2” on page 19-37. Note that the option `Symbolic` should be used if the entries cannot be converted to numerical expressions.

Environment Interactions

Without the option `Symbolic`, the function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

Numerical matrices can be processed with or without the option `Symbolic`:

```
A := array(1..3, 1..3,[[1, 1, I], [1, exp(1), I], [1, 2, 2]]):
numeric::det(A), numeric::det(A, Symbolic)
```

$$3.436563657 - 1.718281828\,i,\ e\,(2 - i) - 2 + i$$

The option `Symbolic` must be used when the matrix has non-numerical entries:

```
A := array(1..2, 1..2, [[1/(x + 1),  1], [1/(x + 2), PI]]):
numeric::det(A, Symbolic)
```

$$\frac{2\,\pi - x + \pi\,x - 1}{(x+1)\,(x+2)}$$

If the option `MinorExpansion` is used, symbolic entries are accepted, even if the option `Symbolic` is not specified:

```
detN := numeric::det(A, MinorExpansion);
detS := numeric::det(A, Symbolic, MinorExpansion)
```

$$\frac{0.0000000008\,(2676990817.0\,x + 6603981634.0)}{(x+1.0)\,(x+2.0)}$$

$$\frac{2\,\pi - x + \pi\,x - 1}{(x+1)\,(x+2)}$$

Simplify these results using `Simplify`:

```
Simplify(detN), Simplify(detS)
```

$$\frac{3.141592654}{x+1.0} - \frac{1.0}{x+2.0},\ \frac{\pi}{x+1} - \frac{1}{x+2}$$

```
delete A:
```

Example 2

The following matrix has domain components:

```
A := Dom::Matrix(Dom::IntegerMod(7))([[6, -1], [1, 6]])
```

$$\begin{pmatrix} 6 \bmod 7 & 6 \bmod 7 \\ 1 \bmod 7 & 6 \bmod 7 \end{pmatrix}$$

Note that `numeric::det` computes the determinant of the following matrix:

```
expr(A), numeric::det(A)
```

$$\begin{pmatrix} 6 & 6 \\ 1 & 6 \end{pmatrix}, 30.0$$

`det` must be used, if the determinant is to be computed over the component domain `Dom::IntegerMod(7)`:

```
det(A)
```

$$2 \bmod 7$$

```
delete A:
```

Example 3

We demonstrate the use of hardware floats. Hilbert matrices are notoriously ill-conditioned: the computation of the determinant is subject to severe cancellation effects. The following results, both with `HardwareFloats` as well as with `SoftwareFloats`, are marred by numerical roundoff:

```
A := linalg::hilbert(15):
float(numeric::det(A, Symbolic)),
numeric::det(A, HardwareFloats),
numeric::det(A, SoftwareFloats)
```

$$1.058542743\ 10^{-124},\ -3.822215463\ 10^{-121},\ 3.277553006\ 10^{-123}$$

```
delete A:
```

Parameters

A

A square matrix of domain type DOM_ARRAY, or DOM_HFARRAY, or of category Cat::Matrix

mode

One of the flags Hard, HardwareFloats, Soft, SoftwareFloats, or Symbolic

Options

Hard, HardwareFloats, Soft, SoftwareFloats

With Hard (or HardwareFloats), computations are done using fast hardware float arithmetic from within a MuPAD session. Hard and HardwareFloats are equivalent. With this option, the input data are converted to hardware floats and processed by compiled C code. The result is reconverted to MuPAD floats and returned to the MuPAD session.

With Soft (or SoftwareFloats) computations are dome using software float arithmetic provided by the MuPAD kernel. Soft and SoftwareFloats are equivalent. SoftwareFloats is used by default if the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY.

Compared to the SoftwareFloats used by the MuPAD kernel, the computation with HardwareFloats may be many times faster. Note, however, that the precision of hardware arithmetic is limited to about 15 digits. Further, the size of floating-point numbers may not be larger than approximately 10^{308} and not smaller than approximately 10^{-308}.

If no HardwareFloats or SoftwareFloats are requested explicitly, the following strategy is used: If the current value of DIGITS is smaller than 16 or if the matrix A is a hardware float array of domain type DOM_HFARRAY, then hardware arithmetic is tried. If this is successful, the result is returned.

If the result cannot be computed with hardware floats, software arithmetic by the MuPAD kernel is tried.

If the current value of `DIGITS` is larger than 15 and the input matrix `A` is not of domain type `DOM_HFARRAY`, or if one of the options `Soft`, `SoftwareFloats` or `Symbolic` is specified, MuPAD computes the result with its software arithmetic without trying to use hardware floats first.

There may be several reasons for hardware arithmetic to fail:

- The current value of `DIGITS` is larger than 15.
- The data contains symbolic objects.
- The data contains numbers larger than 10^{308} or smaller than 10^{-308} that cannot be represented by hardware floats.

If neither `HardwareFloats` nor `SoftwareFloats` is specified, the user is not informed whether hardware floats or software floats are used.

If `HardwareFloats` are specified but fail due to one of the reasons above, a warning is issued that the (much slower) software floating-point arithmetic of the MuPAD kernel is used.

Note that `HardwareFloats` can only be used if all input data can be converted to floating-point numbers.

The trailing digits in floating-point results computed with `HardwareFloats` and `SoftwareFloats` may differ.

Note: For ill conditioned matrices the results returned with `HardwareFloats` and `SoftwareFloats` may differ significantly! See “Example 3” on page 19-37.

`Symbolic`

This option prevents conversion of the input data to floats. With this option, symbolic entries are accepted. It overrides the option `HardwareFloats`.

Note: This option should not be used for floating-point matrices! The `Symbolic` algorithm does not implement safeguards against numerical instabilities in floating-point operations.

`MinorExpansion`

With this option, recursive minor expansion along the columns is used. This option may be useful for small matrices with symbolic entries.

This option implies `SoftwareFloats`.

With this option, symbolic entries are accepted even if the option `Symbolic` is not used.

`NoWarning`

Suppresses warnings

Return Values

By default, the determinant is returned as a floating-point number. With the option `Symbolic`, an expression is returned.

Algorithms

Without the option `Symbolic`, *QR*-factorization of *A* via Householder transformations is used.

With `Symbolic`, *LU*-factorization of *A* is used.

See Also

MuPAD Functions
`det`

numeric::eigenvalues

Numerical eigenvalues of a matrix

Syntax

```
numeric::eigenvalues(A, options)
```

Description

`numeric::eigenvalues(A)` returns numerical eigenvalues of the matrix `A`.

All entries of `A` must be numerical. Numerical expressions such as e^{π}, $\sqrt{2}$ etc. are accepted and converted to floats. Non-numerical symbolic entries lead to an error.

Note: Matrices `A` of a matrix domain such as `Dom::Matrix(...)` or `Dom::SquareMatrix(..)` are internally converted to arrays over expressions via `expr(A)`. Note that `linalg::eigenvalues` must be used, when the eigenvalues are to be computed over the component domain. Cf. “Example 2” on page 19-43.

The eigenvalues are sorted by `numeric::sort`.

Note: Eigenvalues are approximated with an *absolute* precision of $\frac{r}{10^{\mathrm{DIGITS}}}$, where r is the spectral radius of `A` (i.e., r is the maximum of the absolute values of the eigenvalues). Consequently, large eigenvalues should be computed correctly to `DIGITS` decimal places. The numerical approximations of the small eigenvalues are less accurate.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We compute the eigenvalues of the 3×3 Hilbert matrix:

```
numeric::eigenvalues(linalg::hilbert(3))
```

$$[1.408318927, 0.1223270659, 0.002687340356]$$

The following matrix is ill-conditioned. It has very large as well as very small eigenvalues:

```
A := array(1..3, 1..3,
           [[   I     ,     PI    ,  exp(1)  ],
            [   2     ,  10^100   ,    1     ],
            [10^(-100), 10^(-100), 10^(-100)]
           ]):
```

Precision goal and working precision are set by `DIGITS`. With the standard setting of *DIGITS* = 10, the following result is computed with `HardwareFloats`:

```
numeric::eigenvalues(A)
```

$$\left[1.0\,10^{100},\ 5.0\,10^{-101},\ 1.0\,i\right]$$

$$\left[1.0\,10^{100},\ 5.0\,10^{-101},\ 1.0\,i\right]$$

Note that small eigenvalues may be influenced by roundoff. We increase the working precision by increasing `DIGITS`. The smallest of the eigenvalues is computed more accurately:

```
DIGITS := 200:
eigenvals := numeric::eigenvalues(A):
DIGITS := 5:
eigenvals;
```

$$\left[1.0\,10^{100} + 6.2832\,10^{-200}\,i,\ 1.0\,10^{-100} + 2.7183\,10^{-100}\,i,\ -6.2832\,10^{-100} + 1.0\,i\right]$$

```
delete A, eigenvals, DIGITS:
```

Example 2

The following matrix has domain components:

```
A := Dom::Matrix(Dom::IntegerMod(7))(
    [[6, -1, 4], [0,  3, 3], [0,  0, 3]])
```

$$\begin{pmatrix} 6 \bmod 7 & 6 \bmod 7 & 4 \bmod 7 \\ 0 \bmod 7 & 3 \bmod 7 & 3 \bmod 7 \\ 0 \bmod 7 & 0 \bmod 7 & 3 \bmod 7 \end{pmatrix}$$

Note that `numeric::eigenvalues` computes the eigenvalues of the following matrix:

```
expr(A), numeric::eigenvalues(A)
```

$$\begin{pmatrix} 6 & 6 & 4 \\ 0 & 3 & 3 \\ 0 & 0 & 3 \end{pmatrix}, [6.0, 3.0, 3.0]$$

If the eigenvalues are to be computed over the component domain `Dom::IntegerMod(7)`, `linalg::eigenvalues` should be used:

```
linalg::eigenvalues(A, Multiple)
```

$$[[3 \bmod 7, 2], [6 \bmod 7, 1]]$$

```
delete A:
```

Example 3

We demonstrate the use of hardware floats. Hilbert matrices are notoriously ill-conditioned: the computation of small eigenvalues is subject to severe roundoff effects. In the following results, both with `HardwareFloats` as well as with `SoftwareFloats`, the small eigenvalues are marred by numerical roundoff:

```
A := linalg::hilbert(15):
```

```
numeric::eigenvalues(A, HardwareFloats),
numeric::eigenvalues(A, SoftwareFloats)
```

$$\left[1.845927746,\ \ldots,\ 8.619013323\ 10^{-17},\ 2.268836709\ 10^{-17}\right],$$

$$\left[1.845927746,\ \ldots,\ 2.095930088\ 10^{-18},\ 6.646989777\ 10^{-21}\right]$$

```
delete A:
```

Parameters

A

A numerical matrix of domain type `DOM_ARRAY`, `DOM_HFARRAY`, or of category `Cat::Matrix`.

Options

`Hard`, `HardwareFloats`, `Soft`, `SoftwareFloats`

With `Hard` (or `HardwareFloats`), computations are done using fast hardware float arithmetic from within a MuPAD session. `Hard` and `HardwareFloats` are equivalent. With this option, the input data are converted to hardware floats and processed by compiled C code. The result is reconverted to MuPAD floats and returned to the MuPAD session.

With `Soft` (or `SoftwareFloats`) computations are dome using software float arithmetic provided by the MuPAD kernel. `Soft` and `SoftwareFloats` are equivalent. `SoftwareFloats` is used by default if the current value of `DIGITS` is larger than 15 and the input matrix `A` is not of domain type `DOM_HFARRAY`.

Compared to the `SoftwareFloats` used by the MuPAD kernel, the computation with `HardwareFloats` may be many times faster. Note, however, that the precision of hardware arithmetic is limited to about 15 digits. Further, the size of floating-point numbers may not be larger than approximately 10^{308} and not smaller than approximately 10^{-308}.

If no HardwareFloats or SoftwareFloats are requested explicitly, the following strategy is used: If the current value of DIGITS is smaller than 16 or if the matrix A is a hardware float array of domain type DOM_HFARRAY, then hardware arithmetic is tried. If this is successful, the result is returned.

If the result cannot be computed with hardware floats, software arithmetic by the MuPAD kernel is tried.

If the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY, or if one of the options Soft, SoftwareFloats or Symbolic is specified, MuPAD computes the result with its software arithmetic without trying to use hardware floats first.

There may be several reasons for hardware arithmetic to fail:

- The current value of DIGITS is larger than 15.
- The data contains symbolic objects.
- The data contains numbers larger than 10^{308} or smaller than 10^{-308} that cannot be represented by hardware floats.

If neither HardwareFloats nor SoftwareFloats is specified, the user is not informed whether hardware floats or software floats are used.

If HardwareFloats are specified but fail due to one of the reasons above, a warning is issued that the (much slower) software floating-point arithmetic of the MuPAD kernel is used.

Note that HardwareFloats can only be used if all input data can be converted to floating-point numbers.

The trailing digits in floating-point results computed with HardwareFloats and SoftwareFloats may differ.

Note: For ill conditioned matrices the results returned with HardwareFloats and SoftwareFloats may differ significantly! See “Example 3” on page 19-43.

NoWarning

Suppresses warnings

Return Values

Ordered list of numerical eigenvalues

Algorithms

The function implements standard numerical algorithms from the Handbook of Automatic Computation by Wilkinson and Reinsch.

See Also

MuPAD Functions

`linalg::eigenvalues` | `linalg::eigenvectors` | `numeric::eigenvectors` | `numeric::singularvalues` | `numeric::singularvectors` | `numeric::spectralradius`

numeric::eigenvectors

Numerical eigenvalues and eigenvectors of a matrix

Syntax

```
numeric::eigenvectors(A, options)
```

Description

`numeric::eigenvectors(A)` returns numerical eigenvalues and eigenvectors of the matrix `A`.

All entries of the matrix must be numerical. Numerical expressions such as e^{π}, $\sqrt{2}$ etc. are accepted and converted to floats. Non-numerical symbolic entries lead to an error.

The eigenvalues are sorted by `numeric::sort`.

The matrix `X` provides the eigenvectors: the *i*-th column of `X` is a numerical eigenvector corresponding to the eigenvalue `d[i]`. Each column is either zero or normalized to the Euclidean length 1.0.

For matrices with multiple eigenvalues and an insufficient number of eigenvectors, some of the eigenvectors may coincide or may be zero, i.e., `X` is not necessarily invertible.

The list of residues *res* = [res_1, res_2, …] provides some control over the quality of the numerical spectral data. The residues are given by

$$\mathrm{res}_i = \left\| A\, x_i - d_i\, x_i \right\|_2 ,$$

where x_i is the normalized eigenvector (the *i*-th column of `X`) associated with the numerical eigenvalue d_i. For Hermitian matrices, res_i provides an upper bound for the absolute error of d_i.

With the option `NoResidues`, the computation of the residues is suppressed, the returned value is `NIL`.

If no return type is specified via the option `ReturnType = t`, the domain type of the eigenvector matrix `X` depends on the type of the input matrix `A`:

- The eigenvectors of an array are returned as an array.
- The eigenvectors of an hfarray are returned as an hfarray.
- The eigenvectors of a dense matrix of type `Dom::DenseMatrix()` are returned as a dense matrix of type `Dom::DenseMatrix()` over the ring of expressions.
- For all other matrices of category `Cat::Matrix`, the eigenvectors are returned as matrices of type `Dom::Matrix()` over the ring of MuPAD expressions. This includes input matrices `A` of type `Dom::Matrix(...)`, `Dom::SquareMatrix(...)`, `Dom::MatrixGroup(...)` etc.

Note: Matrices `A` of a matrix domain such as `Dom::Matrix(...)` or `Dom::SquareMatrix(...)` are internally converted to arrays over expressions via `expr(A)`. Note that `linalg::eigenvectors` must be used, when the eigenvalues/vectors are to be computed over the component domain. Cf. “Example 3” on page 19-50.

Note: Eigenvalues are approximated with an *absolute* precision of $\frac{r}{10^{\mathrm{DIGITS}}}$, where r is the spectral radius of `A` (i.e., r is the maximal singular value of `A`). Consequently, large eigenvalues should be computed correctly to `DIGITS` decimal places. The numerical approximations of the small eigenvalues are less accurate.

Note: For a numerical algorithm, it is not possible to distinguish between badly separated distinct eigenvalues and multiple eigenvalues. For this reason, `numeric::eigenvectors` and `linalg::eigenvectors` use different return formats: the latter can provide information on the multiplicity of eigenvalues due to its internal exact arithmetic.

Use `numeric::eigenvalues` if only eigenvalues are to be computed.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We compute the spectral data of the 2×2 Hilbert matrix:

```
A := linalg::hilbert(2)
```

$$\begin{pmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{3} \end{pmatrix}$$

```
[d, X, res] := numeric::eigenvectors(A):
```

The eigenvalues:

```
d
```

$$[1.267591879,\ 0.06574145409]$$

The eigenvectors:

```
X
```

$$\begin{pmatrix} 0.8816745988 & -0.4718579255 \\ 0.4718579255 & 0.8816745988 \end{pmatrix}$$

Hilbert matrices are Hermitian, i.e., computing the spectral data is a numerically stable process. This is confirmed by the small residues:

```
res
```

$$[2.355138688\ 10^{-16},\ 1.388645872\ 10^{-16}]$$

The routine `linalg::hilbert` provides the input as a matrix of type `Dom::Matrix()`. Consequently, the eigenvectors also consist of such a matrix. For further processing, we convert the list of eigenvalues to a diagonal matrix:

```
d := matrix(2, 2, d, Diagonal):
```

We reconstruct the matrix from its spectral data:

```
X*d*X^(-1)
```

$$\begin{pmatrix} 1.0 & 0.5 \\ 0.5 & 0.3333333333 \end{pmatrix}$$

We extract an eigenvector from the matrix `X` and doublecheck its numerical quality:

```
eigenvector1 := X::dom::col(X, 1);
norm(A*eigenvector1 - d[1, 1]*eigenvector1)
```

$$\begin{pmatrix} 0.8816745988 \\ 0.4718579255 \end{pmatrix}$$

$$1.769417945\ 10^{-16}$$

```
delete A, d, X, res, eigenvector1:
```

Example 2

We demonstrate a numerically ill-conditioned case. The following matrix has only one eigenvector and cannot be diagonalized. Numerically, the zero vector is returned as the second column of the eigenvector matrix:

```
A := array(1..2, 1..2, [[5, -1], [4, 1]]):
DIGITS := 6:
numeric::eigenvectors(A)
```

$$\left[[3.0, 3.0], \begin{pmatrix} 0.447214 & 0 \\ 0.894427 & 0 \end{pmatrix}, \left[1.66533\ 10^{-16}, 0.0\right]\right]$$

```
delete A, DIGITS:
```

Example 3

The following matrix has domain components:

```
A := Dom::Matrix(Dom::IntegerMod(7))([[6, -1], [0,  3]])
```

$$\begin{pmatrix} 6 \bmod 7 & 6 \bmod 7 \\ 0 \bmod 7 & 3 \bmod 7 \end{pmatrix}$$

Note that `numeric::eigenvectors` computes the spectral data of the following matrix:

```
expr(A)
```

$$\begin{pmatrix} 6 & 6 \\ 0 & 3 \end{pmatrix}$$

```
numeric::eigenvectors(A, NoResidues)
```

$$\left[[6.0,\ 3.0],\ \begin{pmatrix} 1.0 & 0.894427191 \\ 0 & -0.4472135955 \end{pmatrix},\ \text{NIL}\right]$$

The routine `linalg::eigenvectors` should be used if the spectral data are to be computed over the component domain `Dom::IntegerMod(7)`:

```
linalg::eigenvectors(A)
```

$$\left[\left[3 \bmod 7,\ 1,\ \left[\begin{pmatrix} 5 \bmod 7 \\ 1 \bmod 7 \end{pmatrix}\right]\right],\ \left[6 \bmod 7,\ 1,\ \left[\begin{pmatrix} 1 \bmod 7 \\ 0 \bmod 7 \end{pmatrix}\right]\right]\right]$$

```
delete A:
```

Example 4

We demonstrate the use of hardware floats. The following matrix is degenerate: it has rank 1. For the double eigenvalue 0, different base vectors of the corresponding eigenspace are returned with `HardwareFloats` and `SoftwareFloats`, respectively:

```
A := array(1..3, 1..3, [[1, 2, 3], [2, 4, 6],
                        [3*10^12, 6*10^12, 9*10^12]]):
[d1, X1, res1] := numeric::eigenvectors(A, HardwareFloats):
```

```
d1, X1
```

$$\left[9.0\ 10^{12},\ 0.0,\ -4.543838814\ 10^{-16}\right],$$

$$\begin{pmatrix} 3.333333333\ 10^{-13} & 0.9486832981 & 0.9621023987 \\ 6.666666667\ 10^{-13} & -2.108185107\ 10^{-13} & -0.1012739367 \\ 1.0 & -0.316227766 & -0.2531848418 \end{pmatrix}$$

```
[d2, X2, res2] := numeric::eigenvectors(A, SoftwareFloats):
d2, X2
```

$$\left[9.0\ 10^{12},\ 5.421010862\ 10^{-20},\ 0.0\right],\ \begin{pmatrix} 0 & 0.9592641938 & 0.9486832981 \\ 0 & -0.06851887098 & 0 \\ 1.0 & -0.2740754839 & -0.316227766 \end{pmatrix}$$

```
delete A, d1, X1, res1, d2, X2, res2:
```

Parameters

A

A numerical matrix of domain type `DOM_ARRAY`, or `DOM_HFARRAY`, or of category `Cat::Matrix`.

Options

Hard, HardwareFloats, Soft, SoftwareFloats

With `Hard` (or `HardwareFloats`), computations are done using fast hardware float arithmetic from within a MuPAD session. `Hard` and `HardwareFloats` are equivalent. With this option, the input data are converted to hardware floats and processed by compiled C code. The result is reconverted to MuPAD floats and returned to the MuPAD session.

With `Soft` (or `SoftwareFloats`) computations are dome using software float arithmetic provided by the MuPAD kernel. `Soft` and `SoftwareFloats` are equivalent. `SoftwareFloats` is used by default if the current value of `DIGITS` is larger than 15 and the input matrix `A` is not of domain type `DOM_HFARRAY`.

Compared to the `SoftwareFloats` used by the MuPAD kernel, the computation with `HardwareFloats` may be many times faster. Note, however, that the precision of hardware arithmetic is limited to about 15 digits. Further, the size of floating-point numbers may not be larger than approximately 10^{308} and not smaller than approximately 10^{-308}.

If no `HardwareFloats` or `SoftwareFloats` are requested explicitly, the following strategy is used: If the current value of `DIGITS` is smaller than 16 or if the matrix `A` is a hardware float array of domain type `DOM_HFARRAY`, then hardware arithmetic is tried. If this is successful, the result is returned.

If the result cannot be computed with hardware floats, software arithmetic by the MuPAD kernel is tried.

If the current value of `DIGITS` is larger than 15 and the input matrix `A` is not of domain type `DOM_HFARRAY`, or if one of the options `Soft`, `SoftwareFloats` or `Symbolic` is specified, MuPAD computes the result with its software arithmetic without trying to use hardware floats first.

There may be several reasons for hardware arithmetic to fail:

- The current value of `DIGITS` is larger than 15.
- The data contains symbolic objects.
- The data contains numbers larger than 10^{308} or smaller than 10^{-308} that cannot be represented by hardware floats.

If neither `HardwareFloats` nor `SoftwareFloats` is specified, the user is not informed whether hardware floats or software floats are used.

If `HardwareFloats` are specified but fail due to one of the reasons above, a warning is issued that the (much slower) software floating-point arithmetic of the MuPAD kernel is used.

Note that `HardwareFloats` can only be used if all input data can be converted to floating-point numbers.

The trailing digits in floating-point results computed with `HardwareFloats` and `SoftwareFloats` may differ.

Note: For ill conditioned matrices the results returned with `HardwareFloats` and `SoftwareFloats` may differ significantly! See "Example 4" on page 19-51.

`NoResidues`

Suppresses the computation of error estimates

If no error estimates are required, this option may be used to suppress the computation of the residues `res`. The return values for these data are `NIL`.

The alternative option name `NoErrors` used in previous MuPAD versions is still available.

`ReturnType`

Option, specified as `ReturnType = t`

Return the eigenvectors as a matrix of domain type `t`. The following return types t are available: `DOM_ARRAY`, `DOM_HFARRAY`, `Dom::Matrix()`, or `Dom::DenseMatrix()`.

`NoWarning`

Suppresses warnings

Return Values

List `[d, X, res]`. The list `d = [d`$_1$`, d`$_2$`, …]` contains the numerical eigenvalue. The i-th column of the matrix `X` is the eigenvector associated with the eigenvalue `d`$_i$. The list of residues `res = [res`$_1$`, res`$_2$`, …]` provides error estimates for the numerical eigenvalues.

Algorithms

The routine implements standard numerical algorithms from the Handbook of Automatic Computation by Wilkinson and Reinsch.

See Also

MuPAD Functions

`linalg::eigenvalues` | `linalg::eigenvectors` | `numeric::eigenvalues` | `numeric::singularvalues` | `numeric::singularvectors` | `numeric::spectralradius`

numeric::expMatrix

Exponential of a matrix

Syntax

```
numeric::expMatrix(A, <mode>, <method>, options)
numeric::expMatrix(A, x, <mode>, <method>, options)
numeric::expMatrix(A, X, <mode>, <method>, options)
```

Description

`numeric::expMatrix(A)` returns the exponential e^A of a square matrix A.

`numeric::expMatrix(A, x)` with a vector x returns the vector $e^A x$.

`numeric::expMatrix(A, X)` with a matrix X returns the matrix $e^A X$.

If no return type is specified via the option `ReturnType = d`, the domain type of the result depends on the type of the input matrix `A`:

- For an array A, the result is returned as an array.
- For an hfarray A, the result is returned as an hfarray.
- For a dense matrix$A$ of type `Dom::DenseMatrix(Ring)`, the result is again a matrix of type `Dom::DenseMatrix()` over the ring of expressions.
- For all other matrices A of category `Cat::Matrix`, the result is returned as a `matrix` of type `Dom::Matrix()` over the ring of expressions. This includes input matrices `A` of type `Dom::Matrix(Ring)`, `Dom::SquareMatrix(Ring)`, `Dom::MatrixGroup(Ring)` etc.

The components of A must not contain symbolic objects which cannot be converted to numerical values via `float`. Numerical symbolic expressions such as π, $\sqrt{2}$, e^{-1} etc. are accepted. They are converted to floats.

The specification of a method such as `TaylorExpansion` etc. implies `SoftwareFloats`, i.e., the result is computed via the software arithmetic of the MuPAD kernel.

The methods `Diagonalization` and `Interpolation` do not work for all matrices (see below).

With `SoftwareFloats`, special algorithms are implemented for traceless 2×2 matrices and skew symmetric 3×3 matrices. Specification of a particular method does not have any effect for such matrices.

If $e^A x$ or $e^A X$ is required, one should not compute e^A first and then multiply the resulting matrix with the vector/matrix x/X. In general, the call `numeric::expMatrix(A, x)` or `numeric::expMatrix(A, X)`, respectively, is faster.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We consider a lower triangular matrix given by an array:

```
A := array(1..2, 1..2, [[1, 0] , [1, PI]]):
expA := numeric::expMatrix(A)
```

$$\begin{pmatrix} 2.718281828 & 0 \\ 9.536085572 & 23.14069263 \end{pmatrix}$$

We consider a vector given by a list `x1` and by an equivalent 1-dimensional array `x2`, respectively:

```
x1 := [1, 1]:
x2 := array(1..2, [1, 1]):
```

Further, an equivalent input vector X of type `Dom::Matrix()` is used:

```
X := matrix(x1):
```

The following three calls all yield a vector represented by an 2×1 array corresponding to the type of the input matrix A:

```
numeric::expMatrix(A, x1),
numeric::expMatrix(A, x2, Krylov),
numeric::expMatrix(A, X, Diagonalization)
```

$$\begin{pmatrix} 2.718281828 \\ 32.6767782 \end{pmatrix}, \begin{pmatrix} 2.718281828 \\ 32.6767782 \end{pmatrix}, \begin{pmatrix} 2.718281828 \\ 32.6767782 \end{pmatrix}$$

For further processing, the array `expA` is converted to an element of the matrix domain `Dom::Matrix()`:

```
expA := matrix(expA):
```

Now, the overloaded arithmetical operators `+`, `*`, `^` etc. can be used for further computations:

```
expA*X
```

$$\begin{pmatrix} 2.718281828 \\ 32.6767782 \end{pmatrix}$$

```
delete A, expA, x1, x2, X:
```

Example 2

We demonstrate the different precision goals of the methods. Note that software arithmetic is used when a method is specified:

```
A := array(1..3, 1..3, [[ 1000,    1,    0 ],
                        [   0,     1,    1 ],
                        [1/10^100, 0, -1000]]):
```

The default method `TaylorExpansion` computes *each component* of e^{-A} correctly:

```
numeric::expMatrix(A, SoftwareFloats)
```

$$\begin{pmatrix} 1.970071114\ 10^{434} & 1.972043157\ 10^{431} & 9.860215786\ 10^{427} \\ 9.860215786\ 10^{327} & 9.870085871\ 10^{324} & 4.935042936\ 10^{321} \\ 9.85035557\ 10^{330} & 9.860215786\ 10^{327} & 4.930107893\ 10^{324} \end{pmatrix}$$

The method `Diagonalization` produces a result, which is accurate in the sense that $\|\text{error}\|_\infty \le \|e^{-A}\|_\infty \frac{1}{10^{\text{DIGITS}}}$ holds. Indeed, the largest components of e^{-A} are correct.

However, `Diagonalization` does not even get the right order of magnitude of the smaller components:

```
numeric::expMatrix(A, Diagonalization)
```

$$\begin{pmatrix} 1.970071114\,10^{434} & 1.972043157\,10^{431} & 0 \\ 0 & 2.718281828 & 0 \\ 0 & 0 & 5.075958898\,10^{-435} \end{pmatrix}$$

Note that e^{A} is very sensitive to small changes in A. After elimination of the small lower triangular element, both methods yield the same result with correct digits for all entries:

```
B := array(1..3, 1..3, [[ 1000, 1,    0 ],
                        [   0 , 1,    1 ],
                        [   0 , 0, -1000]]):
numeric::expMatrix(B, SoftwareFloats)
```

$$\begin{pmatrix} 1.970071114\,10^{434} & 1.972043157\,10^{431} & 9.860215786\,10^{427} \\ 0 & 2.718281828 & 0.002715566262 \\ 0 & 0 & 5.075958897\,10^{-435} \end{pmatrix}$$

```
numeric::expMatrix(B, Diagonalization)
```

$$\begin{pmatrix} 1.970071114\,10^{434} & 1.972043157\,10^{431} & 9.860215786\,10^{427} \\ 0 & 2.718281828 & 0.002715566262 \\ 0 & 0 & 5.075958898\,10^{-435} \end{pmatrix}$$

```
delete A, B:
```

Example 3

Hilbert matrices $H_{ij} = \frac{1}{i+j-1}$ have real positive eigenvalues. For large dimension, most of these eigenvalues are small and may be regarded as a single cluster. Consequently, the option `Krylov` is useful:

```
numeric::expMatrix(linalg::hilbert(100), [1 $ 100], Krylov)
```

```
[25.47080919, 18.59337041, ..., 2.863083064, 2.848538965]
```

Parameters

A

A square $n \times n$ matrix of domain type DOM_ARRAY, DOM_HFARRAY, or of category `Cat::Matrix`

x

A vector represented by a list `[`x_1`, …, `x_n`]` or a 1-dimensional array `array(1..n, [`x_1`, …, `x_n`] )`, or a 1-dimensional hfarray `hfarray(1..n, [`x_1`, …, `x_n`] )`

X

An $n \times m$ matrix of domain type DOM_ARRAY, DOM_HFARRAY, `Dom::Matrix(Ring)` or `Dom::DenseMatrix(Ring)` with a suitable coefficient ring `Ring`

mode

One of the flags `Hard`, `HardwareFloats`, `Soft`, or `SoftwareFloats`

method

One of the flags `Diagonalization`, `Interpolation`, `Krylov`, or `TaylorExpansion`

Options

Hard, HardwareFloats, Soft, SoftwareFloats

With `Hard` (or `HardwareFloats`), computations are done using fast hardware float arithmetic from within a MuPAD session. `Hard` and `HardwareFloats` are equivalent. With this option, the input data are converted to hardware floats and processed by

compiled C code. The result is reconverted to MuPAD floats and returned to the MuPAD session.

With `Soft` (or `SoftwareFloats`) computations are dome using software float arithmetic provided by the MuPAD kernel. `Soft` and `SoftwareFloats` are equivalent. `SoftwareFloats` is used by default if the current value of `DIGITS` is larger than 15 and the input matrix `A` is not of domain type `DOM_HFARRAY`.

Compared to the `SoftwareFloats` used by the MuPAD kernel, the computation with `HardwareFloats` may be many times faster. Note, however, that the precision of hardware arithmetic is limited to about 15 digits. Further, the size of floating-point numbers may not be larger than approximately 10^{308} and not smaller than approximately 10^{-308}.

If no `HardwareFloats` or `SoftwareFloats` are requested explicitly, the following strategy is used: If the current value of `DIGITS` is smaller than 16 or if the matrix `A` is a hardware float array of domain type `DOM_HFARRAY`, then hardware arithmetic is tried. If this is successful, the result is returned.

If the result cannot be computed with hardware floats, software arithmetic by the MuPAD kernel is tried.

If the current value of `DIGITS` is larger than 15 and the input matrix `A` is not of domain type `DOM_HFARRAY`, or if one of the options `Soft`, `SoftwareFloats` or `Symbolic` is specified, MuPAD computes the result with its software arithmetic without trying to use hardware floats first.

There may be several reasons for hardware arithmetic to fail:

- The current value of `DIGITS` is larger than 15.
- The data contains symbolic objects.
- The data contains numbers larger than 10^{308} or smaller than 10^{-308} that cannot be represented by hardware floats.

If neither `HardwareFloats` nor `SoftwareFloats` is specified, the user is not informed whether hardware floats or software floats are used.

If `HardwareFloats` are specified but fail due to one of the reasons above, a warning is issued that the (much slower) software floating-point arithmetic of the MuPAD kernel is used.

Note that `HardwareFloats` can only be used if all input data can be converted to floating-point numbers.

The trailing digits in floating-point results computed with `HardwareFloats` and `SoftwareFloats` may differ.

`Diagonalization`, `Interpolation`, `Krylov`, `TaylorExpansion`

The specification of a method implies `SoftwareFloats`, i.e., the result is always computed via the software arithmetic of the MuPAD kernel.

The method `TaylorExpansion` is the default algorithm. It produces fast results for matrices with small norms.

The default method `TaylorExpansion` computes *each individual component* of e^A, $e^A x$, $e^A X$ to a relative precision of about `10^(-DIGITS)`, unless numerical roundoff prevents reaching this precision goal. Roughly speaking: all digits of all components of the result are reliable up to roundoff effects.

Note: The methods `Diagonalization`, `Interpolation`, and `Krylov` compute the result to a relative precision w.r.t. the norm: $||\text{error}||_\infty \le \frac{||\text{result}||_\infty}{10^{\text{DIGITS}}}$. Consequently, if the result has components of different orders of magnitude, then the smaller components have larger relative errors than the large components. Not all digits of the small components are reliable! Cf. “Example 2” on page 19-58.

Note: The method `Diagonalization` only works for diagonalizable matrices. For matrices without a basis of eigenvectors, `numeric::expMatrix` may either produce an error or the returned result is dominated by roundoff effects. For symmetric/Hermitian or skew/skew-Hermitian matrices, this method produces reliable results.

Note: The method `Interpolation` may become numerically unstable for certain matrices. The algorithm tries to detect such instabilities and stops with an error message.

The method `Krylov` is only available for computing $e^A x$ with a vector x. Also vectors represented by $n \times 1$ matrices are accepted.

This method is fast when x is spanned by few eigenvectors of A. Further, if A has only few clusters of similar eigenvalues, then this method can be much faster than the other methods. Cf. “Example 3” on page 19-59.

`NoWarning`

Suppresses warnings

`ReturnType`

Option, specified as `ReturnType = d`

Return the result matrix or vector as a matrix of domain type `d`. The following return types are available: `DOM_ARRAY`, `DOM_HFARRAY`, `Dom::Matrix()`, or `Dom::DenseMatrix()`.

Return Values

All results are float matrices/vectors. For an $n \times n$ matrix A:

- `numeric::expMatrix(A, method)` returns e^A as an $n \times n$ matrix,
- `numeric::expMatrix(A, x, method)` returns $e^A x$ as an $n \times 1$ matrix,
- `numeric::expMatrix(A, X, method)` returns $e^A X$ as an $n \times m$ matrix.

The domain type of the result depends on the domain type of the input matrix A unless a return type is requested explicitly via `ReturnType = d`.

Algorithms

The method `TaylorExpansion` sums the usual Taylor series

$$e^A = 1 + A + \frac{A^2}{2} + \cdots$$

in a suitable numerically stable way.

The method `Diagonalization` computes $e^A = T \operatorname{diag}\left(e^{\lambda_1}, e^{\lambda_2}, \ldots\right) T^{-1}$ by a diagonalization $A = T\,diag(\lambda_1, \lambda_2, \ldots)\,T^{-1}$.

The method `Interpolation` computes a polynomial P interpolating the function *exp* at the eigenvalues of A. Evaluation of the matrix polynomial yields $e^A = P(A)$.

The method `Krylov` reduces A to a Hessenberg matrix H and computes an approximation of $e^A x$ from e^H. Depending on A and x, the dimension of H may be smaller than the dimension of A.

`numeric::expMatrix` uses polynomial arithmetic to multiply matrices and vectors. Thus, sparse matrices are handled efficiently based on the MuPAD internal sparse representation of polynomials.

References

Y. Saad, “Analysis of some Krylov Subspace Approximations to the Matrix Exponential Operator”, SIAM Journal of Numerical Analysis *29* (1992).

See Also

MuPAD Functions

`exp` | `funm` | `linalg::sqrtMatrix` | `numeric::fMatrix`

numeric::factorCholesky

Cholesky factorization of a matrix

Syntax

```
numeric::factorCholesky(A, options)
```

Description

`numeric::factorCholesky(A)` returns the factor L of the Cholesky factorization $A = L\,L^H$ of a positive definite Hermitian matrix A.

`numeric::factorCholesky(A, Symmetric)` returns the factor L of a Cholesky type factorization $A = L\,L^T$ of a symmetric matrix A.

The Cholesky factorization of a square Hermitian matrix is $A = L\,L^H$, where L is a regular complex lower triangular matrix and L^H is the Hermitian transpose of L (i.e., the complex conjugate of the transpose of L). Such a factorization only exists if A is positive definite.

By default, a numerical factorization is computed. If the option `Symbolic` is not used, all components of the matrix are converted to floating-point numbers. In this case, the matrix must not contain symbolic objects that cannot be converted to floats. Numerical symbolic expressions such as π, $\sqrt{2}$, e^{-1} etc. are accepted.

If no return type is specified via the option `ReturnType = d`, the domain type of the Cholesky factor L depends on the type of the input matrix `A`:

- The factor of an array is returned as an array.
- The factor of an hfarray is returned as an hfarray.
- The factor of a dense matrix of type `Dom::DenseMatrix()` is a dense matrix of type `Dom::DenseMatrix()` over the ring of MuPAD expressions.
- For all other matrices of category `Cat::Matrix`, the factor L is returned as a `matrix` of type `Dom::Matrix()` over the ring of MuPAD expressions. This includes

input matrices A of type `Dom::Matrix(...)`, `Dom::SquareMatrix(...)`, `Dom::MatrixGroup(...)` etc.

The Cholesky factor returned by `numeric::factorCholesky` is normalized such that its diagonal elements are real and positive.

Environment Interactions

Without the option `Symbolic`, the function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We consider the matrix

```
A := array(1..2, 1..2, [[1, I] , [-I, PI]]):
```

We compute a numerical factorization

```
numeric::factorCholesky(A)
```

$$\begin{pmatrix} 1.0 & 0 \\ -1.0\,i & 1.46341814 \end{pmatrix}$$

and a symbolic factorization:

```
L := numeric::factorCholesky(A, Symbolic)
```

$$\begin{pmatrix} 1 & 0 \\ -i & \sqrt{\pi - 1} \end{pmatrix}$$

For further processing, the Cholesky factor (of domain type `DOM_ARRAY`) is converted to an element of the matrix domain `Dom::Matrix()`:

```
L := matrix(L):
```

Now, the overloaded arithmetical operators +, *, ^ etc. can be used for further computations:

```
L*linalg::transpose(conjugate(L))
```

$$\begin{pmatrix} 1 & i \\ -i & \pi \end{pmatrix}$$

```
delete A, L:
```

Example 2

The following matrix is not positive definite:

```
A := matrix([[-2, sqrt(2)], [sqrt(2), 1]]):
numeric::factorCholesky(A)
```

```
Error: The matrix is not positive definite within working precision. [stdlib::hfa::fact
  Evaluating: numeric::factorCholesky
```

However, a symmetric factorization with a complex Cholesky factor does exist:

```
numeric::factorCholesky(A, Symmetric)
```

$$\begin{pmatrix} 1.414213562\,i & 0 \\ -1.0\,i & 1.414213562 \end{pmatrix}$$

```
delete A:
```

Example 3

The option `NoCheck` should be used when the matrix contains symbolic objects:

```
assume(x > 0): assume(z > 0):
A := array(1..2, 1..2, [[x, conjugate(y)], [y, z]]):
```

```
numeric::factorCholesky(A, Symbolic, NoCheck)
```

$$\begin{pmatrix} \sqrt{x} & 0 \\ \frac{y}{\sqrt{x}} & \sqrt{z - \frac{|y|^2}{x}} \end{pmatrix}$$

Note that with `NoCheck`, it is assumed that the matrix is Hermitian and positive definite! All upper triangular entries are ignored. The following result implicitly assumes `u = conjugate(y)`:

```
A := array(1..2, 1..2, [[x, u], [y, z]]):
numeric::factorCholesky(A, Symbolic, NoCheck)
```

$$\begin{pmatrix} \sqrt{x} & 0 \\ \frac{y}{\sqrt{x}} & \sqrt{z - \frac{|y|^2}{x}} \end{pmatrix}$$

```
delete A:
```

Example 4

We demonstrate the use of hardware floats. Hilbert matrices are notoriously ill-conditioned and difficult to factor with low values of `DIGITS`. The following results, both with `HardwareFloats` as well as with `SoftwareFloats`, are marred by numerical roundoff. Consequently, the factorization with and without hardware floats, respectively, differ significantly:

```
A := linalg::hilbert(13):
L1 := numeric::factorCholesky(A, HardwareFloats):
L2 := numeric::factorCholesky(A, SoftwareFloats):
L1[13, 13] <> L2[13, 13]
```

$$0.0000001052365221 \neq 0.00000007585706698$$

All Hilbert matrices are positive definite. However, in the following call, numerical roundoff makes the hardware floating-point tool think that the matrix is not definite:

```
numeric::factorCholesky(linalg::hilbert(14), HardwareFloats):
```

```
Error: The matrix is not positive definite within working precision. [stdlib::hfa::fact
  Evaluating: numeric::factorCholesky
```

A factorization is computed successfully with `SoftwareFloats`:

```
L := numeric::factorCholesky(linalg::hilbert(14), SoftwareFloats):
norm(linalg::hilbert(14) - L*linalg::transpose(L))
```

$$8.67361738\ 10^{-19}$$

```
delete A, L1, L2, L:
```

Parameters

A

A square matrix of domain type `DOM_ARRAY`, `DOM_HFARRAY`, or of category `Cat::Matrix`

Options

`Hard`, `HardwareFloats`, `Soft`, `SoftwareFloats`

With `Hard` (or `HardwareFloats`), computations are done using fast hardware float arithmetic from within a MuPAD session. `Hard` and `HardwareFloats` are equivalent. With this option, the input data are converted to hardware floats and processed by compiled C code. The result is reconverted to MuPAD floats and returned to the MuPAD session.

With `Soft` (or `SoftwareFloats`) computations are dome using software float arithmetic provided by the MuPAD kernel. `Soft` and `SoftwareFloats` are equivalent. `SoftwareFloats` is used by default if the current value of `DIGITS` is larger than 15 and the input matrix `A` is not of domain type `DOM_HFARRAY`.

Compared to the `SoftwareFloats` used by the MuPAD kernel, the computation with `HardwareFloats` may be many times faster. Note, however, that the precision of

hardware arithmetic is limited to about 15 digits. Further, the size of floating-point numbers may not be larger than approximately 10^{308} and not smaller than approximately 10^{-308}.

If no `HardwareFloats` or `SoftwareFloats` are requested explicitly, the following strategy is used: If the current value of `DIGITS` is smaller than 16 or if the matrix `A` is a hardware float array of domain type `DOM_HFARRAY`, then hardware arithmetic is tried. If this is successful, the result is returned.

If the result cannot be computed with hardware floats, software arithmetic by the MuPAD kernel is tried.

If the current value of `DIGITS` is larger than 15 and the input matrix `A` is not of domain type `DOM_HFARRAY`, or if one of the options `Soft`, `SoftwareFloats` or `Symbolic` is specified, MuPAD computes the result with its software arithmetic without trying to use hardware floats first.

There may be several reasons for hardware arithmetic to fail:

- The current value of `DIGITS` is larger than 15.
- The data contains symbolic objects.
- The data contains numbers larger than 10^{308} or smaller than 10^{-308} that cannot be represented by hardware floats.

If neither `HardwareFloats` nor `SoftwareFloats` is specified, the user is not informed whether hardware floats or software floats are used.

If `HardwareFloats` are specified but fail due to one of the reasons above, a warning is issued that the (much slower) software floating-point arithmetic of the MuPAD kernel is used.

Note that `HardwareFloats` can only be used if all input data can be converted to floating-point numbers.

The trailing digits in floating-point results computed with `HardwareFloats` and `SoftwareFloats` may differ.

Note: For ill conditioned matrices the results returned with `HardwareFloats` and `SoftwareFloats` may differ significantly! See “Example 4” on page 19-68.

Symbolic

Prevents the conversion of the input data to floating-point numbers. Exact arithmetic is used. This option overrides `HardwareFloats` and `SoftwareFloats`.

The usual arithmetic for MuPAD expressions is used. With this option, the matrix A may contain symbolic objects. Note that the option `NoCheck` must be used for the Hermitian factorization when non-numerical symbolic objects are present.

Symmetric

Makes `numeric::factorCholesky` compute a symmetric factorization $A = L\,L^T$ rather than a Hermitian factorization $A = L\,L^H$

The symmetric Cholesky factorization of a square symmetric matrix is $A = L\,L^T$, where L is a regular complex lower triangular matrix and L^T is the transpose of L. The matrix A does not have to be positive definite. Consequently, with the option `Symmetric` no internal check is performed whether A is positive definite. Note that the symmetric factorization with regular L does not exist for all matrices.

For real symmetric positive definite matrices A the Cholesky factor L is real and the Hermitian factorization $A = L\,L^H$ coincides with the symmetric factorization $A = L\,L^T$.

NoCheck

Prevents `numeric::factorCholesky` from checking that the matrix is Hermitian and positive definite

Without the option `Symmetric`, `numeric::factorCholesky` checks that the matrix A is Hermitian and positive definite. The option `NoCheck` may be used to suppress these checks. It must be used when the matrix contains symbolic objects. Elements in the upper triangular part of the matrix will never be touched by the algorithm!

Note: With this option, `numeric::factorCholesky` returns a result for matrices that are not Hermitian or not positive definite (i.e., no Cholesky factorization exists)! When using this option, it is the user's responsibility to make sure that the input matrix is appropriate.

This option has no effect when the option `Symmetric` is used.

NoWarning

Suppresses warnings

If symbolic coefficients are found, `numeric::factorCholesky` automatically switches to the `Symbolic` mode with a warning. With this option, this warning is suppressed; the routine still uses the symbolic mode for symbolic coefficients, i.e., exact arithmetic without floating-point conversions is used.

ReturnType

Option, specified as `ReturnType = d`

Return the Cholesky factor as a matrix of domain type `d`. The following return types are available: `DOM_ARRAY`, `DOM_HFARRAY`, `Dom::Matrix()`, or `Dom::DenseMatrix()`.

Return Values

Depending on the type of the input matrix `A`, the lower triangular Cholesky factor L is returned as a matrix of domain type `DOM_ARRAY`, `DOM_HFARRAY`, `Dom::Matrix()`, or `Dom::DenseMatrix()`. Its components are real or complex floats, unless the option `Symbolic` is used. Without the option `NoCheck`, an error is raised if the matrix is not Hermitian or not positive definite.

See Also

MuPAD Functions

`linalg::factorCholesky` | `numeric::factorLU` | `numeric::factorQR`

numeric::factorLU

LU factorization of a matrix

Syntax

```
numeric::factorLU(A, options)
```

Description

`numeric::factorLU(A)` returns a LU factorization of the matrix A.

The LU factorization of a real or complex $m \times n$ matrix `A` is $PA = LU$. The $m \times m$ matrix L is lower triangular, normalized to 1 along the diagonal. The $m \times n$ matrix U is upper triangular, i.e., $U_{ij} = 0$ for $j < i$. The list `p = [p1, ..., pm]` returned by `numeric::factorLU` is a permutation of the numbers 1, …, m corresponding to row exchanges of A. It represents the following $m \times m$ permutation matrix P (we assume that the matrix indices range from 1 to m):

$$P_{ij} = \delta_{p_i, j} = \begin{cases} 1 & \text{if } j = p_i \\ 0 & \text{if } j \neq p_i \end{cases}.$$

Left multiplication of matrices and vectors with P is realized easily using the permutation list `p`: $Y_{i,j} := X_{p_i,j}$ defines the row permutation $Y = PX$ of a matrix X, $y_i := x_{p_i}$ defines the row permutation $y = Px$ of a vector x.

By default, a numerical factorization with partial pivoting is computed. If the option `Symbolic` is not used, all components of the matrix are converted to floating-point numbers. In this case, the matrix must not contain symbolic objects that cannot be converted to floats. Numerical symbolic expressions such as `PI`, $\sqrt{2}$, e^{-1} etc. are accepted.

The factorization depends on the pivoting strategy. The results obtained with/without the option `Symbolic` may differ. See “Example 2” on page 19-75. For numerical factorizations, the results obtained with `HardwareFloats` and `SoftwareFloats`, respectively, may differ. See “Example 3” on page 19-76.

If no return type is specified via the option `ReturnType = d`, the domain type of the factors L and U depends on the type of the input matrix `A`:

- The factors of an array are returned as arrays.
- The factors of an hfarray are returned as hfarrays.
- The factors of a dense matrix of type `Dom::DenseMatrix()` are again dense matrices of type `Dom::DenseMatrix()` over the ring of MuPAD expressions.
- For all other matrices of category `Cat::Matrix`, the factors are returned as matrices of type `Dom::Matrix()` over the ring of MuPAD expressions. This includes input matrices `A` of type `Dom::Matrix(...)`, `Dom::SquareMatrix(...)`, `Dom::MatrixGroup(...)` etc.

Environment Interactions

Without the optional argument `Symbolic`, the function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We factor a matrix specified by an array:

```
A := array(1..3, 1..3, [[1, 2, 3], [2, 4, 6], [4, 8, 9]]):
[L, U, p] := numeric::factorLU(A)
```

$$\left[\begin{pmatrix} 1.0 & 0 & 0 \\ 0.5 & 1.0 & 0 \\ 0.25 & 0 & 1.0 \end{pmatrix}, \begin{pmatrix} 4.0 & 8.0 & 9.0 \\ 0 & 0 & 1.5 \\ 0 & 0 & 0.75 \end{pmatrix}, [3, 2, 1]\right]$$

The factors (of domain type `DOM_ARRAY`) are converted to elements of the matrix domain `Dom::Matrix()`. After the conversion, the overloaded arithmetical operators `+`, `*`, `^` etc. can be used for further processing:

```
L := matrix(L): U := matrix(U):
```

```
L*U
```

$$\begin{pmatrix} 4.0 & 8.0 & 9.0 \\ 2.0 & 4.0 & 6.0 \\ 1.0 & 2.0 & 3.0 \end{pmatrix}$$

The product $L\,U$ coincides with A after exchanging the rows according to the permutation stored in the list p:

```
PA := array(1..3, 1..3, [[A[p[i], j] $ j=1..3] $ i=1..3])
```

$$\begin{pmatrix} 4 & 8 & 9 \\ 2 & 4 & 6 \\ 1 & 2 & 3 \end{pmatrix}$$

```
delete A, L, U, p, PA:
```

Example 2

We consider a non-square matrix of dimension 3 ×2:

```
A := matrix([[3*I, 10], [I, 1], [I, 1]]):
[L1, U1, p1] := numeric::factorLU(A)
```

$$\left[\begin{pmatrix} 1.0 & 0 & 0 \\ 3.0 & 1.0 & 0 \\ 1.0 & 0 & 1.0 \end{pmatrix}, \begin{pmatrix} 1.0\,i & 1.0 \\ 0 & 7.0 \\ 0 & 0 \end{pmatrix}, [2, 1, 3]\right]$$

Note that the symbolic factorization is different, because a different pivoting strategy is used:

```
[L2, U2, p2] := numeric::factorLU(A, Symbolic)
```

$$\left[\begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{3} & 1 & 0 \\ \frac{1}{3} & 1 & 1 \end{pmatrix}, \begin{pmatrix} 3\,i & 10 \\ 0 & -\frac{7}{3} \\ 0 & 0 \end{pmatrix}, [1, 2, 3]\right]$$

Here, the matrix factors are of type `Dom::Matrix()`, because the input matrix A was of this type. We can use the overloaded arithmetic directly. We convert the permutation lists `p1`, `p2` to matrices and verify the relation $PA = LU$ for the factorization:

```
P1 := matrix(3, 3):
P2 := matrix(3, 3):
for i from 1 to 3 do
  P1[i, p1[i]] := 1;
  P2[i, p2[i]] := 1;
end_for:
P1*A - L1*U1, P2*A - L2*U2
```

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

```
delete A, L1, U1, p1, L2, U2, p2:
```

Example 3

We demonstrate the use of hardware floats. The internal rounding of `HardwareFloats` and `SoftwareFloats` differs. Consequently, the following results do not coincide:

```
n := 14:
A := linalg::hilbert(n):
[L1, U1, p1] := numeric::factorLU(A, HardwareFloats):
[L2, U2, p2] := numeric::factorLU(A, SoftwareFloats):
p1, p2
```

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 13], [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]$$

However, both factorizations satisfy $PA = LU$ numerically:

```
P1A := matrix([[A[p1[i], j] $ j = 1..n] $ i = 1..n]):
P2A := matrix([[A[p2[i], j] $ j = 1..n] $ i = 1..n]):
norm(P1A - L1*U1), norm(P2A - L2*U2)
```

$$8.131516294\ 10^{-17}, 5.421010862\ 10^{-19}$$

```
delete n, A, L1, U1, p1, L2, U2, p2, P1A, P2A:
```

Parameters

A

An $m \times n$ matrix of domain type DOM_ARRAY, DOM_HFARRAY, or of category Cat::Matrix

Options

Hard, HardwareFloats, Soft, SoftwareFloats

With Hard (or HardwareFloats), computations are done using fast hardware float arithmetic from within a MuPAD session. Hard and HardwareFloats are equivalent. With this option, the input data are converted to hardware floats and processed by compiled C code. The result is reconverted to MuPAD floats and returned to the MuPAD session.

With Soft (or SoftwareFloats) computations are dome using software float arithmetic provided by the MuPAD kernel. Soft and SoftwareFloats are equivalent. SoftwareFloats is used by default if the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY.

Compared to the SoftwareFloats used by the MuPAD kernel, the computation with HardwareFloats may be many times faster. Note, however, that the precision of hardware arithmetic is limited to about 15 digits. Further, the size of floating-point numbers may not be larger than approximately 10^{308} and not smaller than approximately 10^{-308}.

If no HardwareFloats or SoftwareFloats are requested explicitly, the following strategy is used: If the current value of DIGITS is smaller than 16 or if the matrix A is a hardware float array of domain type DOM_HFARRAY, then hardware arithmetic is tried. If this is successful, the result is returned.

If the result cannot be computed with hardware floats, software arithmetic by the MuPAD kernel is tried.

If the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY, or if one of the options Soft, SoftwareFloats or Symbolic is specified, MuPAD computes the result with its software arithmetic without trying to use hardware floats first.

There may be several reasons for hardware arithmetic to fail:

- The current value of `DIGITS` is larger than 15.
- The data contains symbolic objects.
- The data contains numbers larger than 10^{308} or smaller than 10^{-308} that cannot be represented by hardware floats.

If neither `HardwareFloats` nor `SoftwareFloats` is specified, the user is not informed whether hardware floats or software floats are used.

If `HardwareFloats` are specified but fail due to one of the reasons above, a warning is issued that the (much slower) software floating-point arithmetic of the MuPAD kernel is used.

Note that `HardwareFloats` can only be used if all input data can be converted to floating-point numbers.

The trailing digits in floating-point results computed with `HardwareFloats` and `SoftwareFloats` may differ.

Note: For ill conditioned matrices the results returned with `HardwareFloats` and `SoftwareFloats` may differ significantly! See “Example 3” on page 19-76.

`Symbolic`

Prevents the conversion of the input data to floating-point numbers. Exact arithmetic is used. This option overrides `HardwareFloats` and `SoftwareFloats`.

The usual arithmetic for MuPAD expressions is used. With this option, the matrix A may contain symbolic objects.

With this option, no row exchanges are performed in the internal Gaussian elimination unless necessary.

`NoWarning`

Suppresses warnings

If symbolic coefficients are found, `numeric::factorLU` automatically switches to the `Symbolic` mode with a warning. With this option, this warning is suppressed;

the routine still uses the symbolic mode for symbolic coefficients, i.e., exact arithmetic without floating-point conversions is used.

`ReturnType`

Option, specified as `ReturnType = d`

Return the factors as matrices of domain type `d`. The following return types are available: `DOM_ARRAY`, `DOM_HFARRAY`, `Dom::Matrix()`, or `Dom::DenseMatrix()`.

Return Values

List `[L, U, p]` is returned. The domain type of the $m \times m$ matrix `L` and the $m \times n$ matrix `U` depends on the type of the input matrix `A`; `p` is a list with m elements consisting of a permutation of the integers 1, …, m. It represents row exchanges in pivoting steps. The components of `L` and `U` are real or complex floats, unless the option `Symbolic` is used.

See Also

MuPAD Functions

`linalg::factorLU` | `numeric::factorCholesky` | `numeric::factorQR`

numeric::factorQR

QR factorization of a matrix

Syntax

```
numeric::factorQR(A, options)
```

Description

`numeric::factorQR(A)` returns a QR factorization $A = Q\,R$ of the matrix A.

The QR factorization of a real/complex $m \times n$ matrix is $A = Q\,R$, where the $m \times m$ matrix Q is orthogonal/unitary and the $m \times n$ matrix R is upper triangular (i.e., $R_{ij} = 0$ for $j < i$).

By default, a numerical factorization is computed. The matrix must not contain symbolic objects that cannot be converted to floats. Numerical symbolic expressions such as π, $\sqrt{2}$, e^{-1} etc. are accepted. They will be converted to floats, unless the option `Symbolic` is used.

The R factor is normalized such that its diagonal elements R_{ii} with $i = 1, \ldots, min(m, n)$ are real and nonnegative.

If no return type is specified via the option `ReturnType = d`, the domain type of the factors Q and R depends on the type of the input matrix `A`:

- The factors of an array are returned as arrays.
- The factors of an hfarray are returned as hfarrays.
- The factors of a dense matrix of type `Dom::DenseMatrix()` are dense matrices of type `Dom::DenseMatrix()` over the ring of expressions.
- For all other matrices of category`Cat::Matrix`, the factors are returned as matrices of type `Dom::Matrix()` over the ring of MuPAD expressions. This includes input matrices `A` of type `Dom::Matrix(...)`, `Dom::SquareMatrix(...)`, `Dom::MatrixGroup(...)`, etc.

Environment Interactions

Without the option `Symbolic`, the function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We consider a quadratic matrix:

```
A := array(1..2, 1..2, [[1, 0] , [1, PI]]):
```

First, we compute a numerical factorization:

```
[Q1, R1] := numeric::factorQR(A)
```

$$\left[\begin{pmatrix} 0.7071067812 & -0.7071067812 \\ 0.7071067812 & 0.7071067812 \end{pmatrix}, \begin{pmatrix} 1.414213562 & 2.221441469 \\ 0 & 2.221441469 \end{pmatrix}\right]$$

Next, the symbolic factorization is computed:

```
[Q2, R2] := numeric::factorQR(A, Symbolic)
```

$$\left[\begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{pmatrix}, \begin{pmatrix} \sqrt{2} & \frac{\pi\sqrt{2}}{2} \\ 0 & \frac{\pi\sqrt{2}}{2} \end{pmatrix}\right]$$

For further processing, the factors (of domain type `DOM_ARRAY`) are converted to elements of the matrix domain `Dom::Matrix()`:

```
Q1 := matrix(Q1): R1 := matrix(R1):
Q2 := matrix(Q2): R2 := matrix(R2):
```

Now, the overloaded arithmetical operators `+`, `*`, `^` etc. can be used for further computations:

```
Q1*R1, Q2*R2
```

$$\begin{pmatrix} 1.0 & -4.926614672\,10^{-16} \\ 1.0 & 3.141592654 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 1 & \pi \end{pmatrix}$$

We finally verify the othogonality of the factors `Q1` and `Q2`:

```
Q1 * linalg::transpose(Q1), Q2 * linalg::transpose(Q2)
```

$$\begin{pmatrix} 1.0 & -1.569924746\,10^{-16} \\ -1.569924746\,10^{-16} & 1.0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

```
delete A, Q1, R1, Q2, R2:
```

Example 2

We consider a non-square matrix of rank 1:

```
A := array(1..3, 1..2, [[0, 0], [I, 1], [I, 1]]):
numeric::factorQR(A, Symbolic)
```

$$\left[\begin{pmatrix} 0 & 1 & 0 \\ \frac{\sqrt{2}\,i}{2} & 0 & \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}\,i}{2} & 0 & -\frac{\sqrt{2}}{2} \end{pmatrix}, \begin{pmatrix} \sqrt{2} & -\sqrt{2}\,i \\ 0 & 0 \\ 0 & 0 \end{pmatrix}\right]$$

In this case, the *QR* factorization is not unique. Note that the numerical factorization yields different factors:

```
numeric::factorQR(A)
```

$$\left[\begin{pmatrix} 0 & 0.7071067812\,i & 0.7071067812\,i \\ 0.7071067812\,i & 0.5 & -0.5 \\ 0.7071067812\,i & -0.5 & 0.5 \end{pmatrix}, \begin{pmatrix} 1.414213562 & -1.414213562\,i \\ 0 & 0 \\ 0 & 0 \end{pmatrix}\right]$$

```
delete A:
```

Example 3

We demonstrate the difference between hardware floats and software floats. For rank deficient matrices, the *QR* factorization is not unique. Depending on the options, different results are returned for the following matrix of rank 1:

```
A := matrix([[1, 1], [10^4, 10^4], [10^8, 10^8]]):
[Q1, R1] := float(numeric::factorQR(A, Symbolic))
```

$$\left[\begin{pmatrix} 0.00000000999999995 & 1.0 & 0 \\ 0.0000999999995 & -9.9999999\,10^{-13} & 0.999999995 \\ 0.999999995 & -0.0000000099999999 & -0.0000999999995 \end{pmatrix}, \begin{pmatrix} 100000000.5 & 100000000.5 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}\right]$$

```
[Q2, R2] := numeric::factorQR(A, SoftwareFloats)
```

$$\left[\begin{pmatrix} 0.00000000999999995 & -0.0000999999995 & -0.999999995 \\ 0.0000999999995 & 0.99999999 & -0.000099999998 \\ 0.999999995 & -0.000099999998 & 0.00000001999999975 \end{pmatrix}, \begin{pmatrix} 100000000.5 & 100000000.5 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}\right]$$

```
[Q3, R3] := numeric::factorQR(A, HardwareFloats)
```

$$\left[\begin{pmatrix} 0.000000009999999939 & 0.9999999998 & -0.00002207031223 \\ 0.0000999999995 & -0.00002207031311 & -0.9999999948 \\ 0.999999995 & -0.000000007792968604 & 0.0000999999997 \end{pmatrix}, \begin{pmatrix} 100000000.5 & 100000000.5 \\ 0 & 0.0000000149011613 \\ 0 & 0 \end{pmatrix}\right]$$

However, all factorizations satisfy $A = Q\,R$ numerically:

```
norm(A - Q1*R1), norm(A - Q2*R2), norm(A - Q3*R3)
```

$$3.469446952\ 10^{-18},\ 7.327374818\ 10^{-11},\ 0.00000001583248377$$

```
delete A, Q1, R1, Q2, R2, Q3, R3:
```

Parameters

A

An $m \times n$ matrix of domain type `DOM_ARRAY`, `DOM_HFARRAY`, or of category `Cat::Matrix`

Options

Hard, HardwareFloats, Soft, SoftwareFloats

With `Hard` (or `HardwareFloats`), computations are done using fast hardware float arithmetic from within a MuPAD session. `Hard` and `HardwareFloats` are equivalent. With this option, the input data are converted to hardware floats and processed by compiled C code. The result is reconverted to MuPAD floats and returned to the MuPAD session.

With `Soft` (or `SoftwareFloats`) computations are dome using software float arithmetic provided by the MuPAD kernel. `Soft` and `SoftwareFloats` are equivalent.

`SoftwareFloats` is used by default if the current value of `DIGITS` is larger than 15 and the input matrix `A` is not of domain type `DOM_HFARRAY`.

Compared to the `SoftwareFloats` used by the MuPAD kernel, the computation with `HardwareFloats` may be many times faster. Note, however, that the precision of hardware arithmetic is limited to about 15 digits. Further, the size of floating-point numbers may not be larger than approximately 10^{308} and not smaller than approximately 10^{-308}.

If no `HardwareFloats` or `SoftwareFloats` are requested explicitly, the following strategy is used: If the current value of `DIGITS` is smaller than 16 or if the matrix `A` is a hardware float array of domain type `DOM_HFARRAY`, then hardware arithmetic is tried. If this is successful, the result is returned.

If the result cannot be computed with hardware floats, software arithmetic by the MuPAD kernel is tried.

If the current value of `DIGITS` is larger than 15 and the input matrix `A` is not of domain type `DOM_HFARRAY`, or if one of the options `Soft`, `SoftwareFloats` or `Symbolic` is specified, MuPAD computes the result with its software arithmetic without trying to use hardware floats first.

There may be several reasons for hardware arithmetic to fail:

- The current value of `DIGITS` is larger than 15.
- The data contains symbolic objects.
- The data contains numbers larger than 10^{308} or smaller than 10^{-308} that cannot be represented by hardware floats.

If neither `HardwareFloats` nor `SoftwareFloats` is specified, the user is not informed whether hardware floats or software floats are used.

If `HardwareFloats` are specified but fail due to one of the reasons above, a warning is issued that the (much slower) software floating-point arithmetic of the MuPAD kernel is used.

Note that `HardwareFloats` can only be used if all input data can be converted to floating-point numbers.

The trailing digits in floating-point results computed with `HardwareFloats` and `SoftwareFloats` may differ.

Note: For ill conditioned matrices the results returned with `HardwareFloats` and `SoftwareFloats` may differ significantly! See "Example 3" on page 19-83.

`Symbolic`

Prevents the conversion of the input data to floating-point numbers. Exact arithmetic is used. This option overrides `HardwareFloats` and `SoftwareFloats`.

The usual arithmetic for MuPAD expressions is used. With this option, the matrix A may contain symbolic objects.

`NoWarning`

Suppresses warnings

If symbolic coefficients are found, `numeric::factorQR` automatically switches to the `Symbolic` mode with a warning. With this option, this warning is suppressed; the routine still uses the symbolic mode for symbolic coefficients, i.e., exact arithmetic without floating-point conversions is used.

`ReturnType`

Option, specified as `ReturnType = d`

Return the Cholesky factor as a matrix of domain type `d`. The following return types are available: `DOM_ARRAY`, `DOM_HFARRAY`, `Dom::Matrix()`, or `Dom::DenseMatrix()`.

Return Values

List `[Q, R]` with matrices `Q` and `R` is returned. The domain type of the orthogonal/unitary $m \times m$ matrix `Q` and the upper triangular $m \times n$ matrix `R` depends on the type of the input matrix `A`. The components of `Q` and `R` are real or complex floats, unless the option `Symbolic` is used.

Algorithms

Householder transformations are used to compute the numerical factorization. With the option `Symbolic`, Gram-Schmidt orthonormalization of the columns of A is used.

For an invertible square matrix A, the QR factorization is unique up to scaling factors of modulus 1. The normalization of R to real positive diagonal elements determines the factorization uniquely. Consequently, the results obtained with/without the option `Symbolic` coincide for invertible square matrices.

For singular or non-square matrices, the factorization is not unique and the results obtained with/without the option `Symbolic` may differ. Cf. “Example 2” on page 19-82.

See Also

MuPAD Functions

`linalg::factorQR` | `numeric::factorCholesky` | `numeric::factorLU`

numeric::fft

Fast Fourier Transform

Syntax

```
numeric::fft(L, <mode>, <ReturnType = t>, <Clean>)
numeric::fft(M, <mode>, <ReturnType = t>, <Clean>)
numeric::fft(A, <mode>, <ReturnType = t>, <Clean>)
```

Description

`numeric::fft(data)` returns the discrete Fourier transform of the `data`.

The 1-dimensional discrete Fourier transform $F = fft(L)$ of N data elements L_j stored in the list $L = [L_1, \ldots, L_N]$ is the list $F = [F_1, \ldots, F_N]$ given by

$$F_k = \sum_{j=1}^{N} L_j \, e^{-\frac{i\,2\,\pi\,(j-1)\,(k-1)}{N}}, \; k = 1, \ldots, N$$

.

`fft` transforms the data by a Fast Fourier Transform (FFT) algorithm.

The d-dimensional discrete Fourier transform $F = fft(A)$ of $N = n_1 \times \cdots \times n_d$ data elements $(A_{j_1, \ldots, j_d})$ stored in the array A is the array $F = (F_{k_1, \ldots, k_d})$ given by

$$F_{k_1, \ldots, k_d} = \sum_{j_1=1}^{n_1} \cdots \sum_{j_d=1}^{n_d} A_{j_1, \ldots, j_d} \, e^{-i\,2\,\pi \left(\frac{(j_1-1)\,(k_1-1)}{n_1} + \ldots + \frac{(j_d-1)\,(k_d-1)}{n_d} \right)}$$

with $k_1 = 1, \ldots, n_1, \ldots, k_d = 1, \ldots, n_d$.

Data provided by lists or 1-dimensional `array`s or `hfarray`s are transformed according to the 1-dimensional transform. Data provided by matrices are transformed according to

the 2-dimensional transform. Data provided by multidimensional arrays or hfarrays are transformed according to the multi-dimensional transform matching the format of the input array.

If the data size factorizes ($N = p\,q$, say), the discrete Fourier transform can be computed by p different Fourier transforms of subsets of the data, each subset having the data size q. The corresponding 'divide and conquer' algorithm is known as FFT ('Fast Fourier Transform'). The `fft` routine employs the FFT algorithm. It is most efficient, when the data size N is an integer power of 2 ('radix 2 FFT'). In this case, the algorithm needs $O(N \log_2(N))$ elementary operations.

Note: More generally, FFT is efficient, if the data size is the product of many small factors!

Following Bluestein, the Fourier transform is written as a convolution if the data size N is a prime. The data are zero-padded to a data length that is an integer power of 2. The convolution is then computed via radix 2 FFTs. Thus, the algorithm needs $O(N \log_2(N))$ elementary operations even if N is a prime.

Environment Interactions

Without the option `Symbolic`, the function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We give a demonstration of 1-dimensional transformations using lists. By default, numerical expressions are converted to floats:

```
L := [1, 2^(1/2), 3*I, PI]:
F := numeric::fft(L)
```

$$[5.555806216 + 3.0\,i,\ 1.0 - 1.272620909\,i,\ -3.555806216 + 3.0\,i,\ 1.0 - 4.727379091\,i]$$

```
numeric::invfft(F)
```

$$\left[1.0,\ 1.414213562 + 2.512840965\ 10^{-17}\ i,\ 3.0\ i,\ 3.141592654 - 2.512840965\ 10^{-17}\ i\right]$$

```
numeric::invfft(F, Clean)
```

$$[1.0,\ 1.414213562,\ 3.0\ i,\ 3.141592654]$$

Exact arithmetic is used with the option `Symbolic`:

```
F := numeric::fft(L, Symbolic)
```

$$[\pi + \sqrt{2} + 1 + 3\ i,\ \pi\ i - \sqrt{2}\ i + 1 - 3\ i,\ 1 - \sqrt{2} - \pi + 3\ i,\ 1 + \sqrt{2}\ i - \pi\ i - 3\ i]$$

```
numeric::invfft(F, Symbolic)
```

$$[1,\ \sqrt{2},\ 3\ i,\ \pi]$$

Symbolic expressions are accepted. Internally, however, the default method `HardwareFloats` (with `DIGITS < 16`) fails because of the symbolic parameter `x`. The following results are computed with the software arithmetic provided by the MuPAD kernel:

```
L := [x, 2, 3, x]:
numeric::fft(L)
```

$$[2\ x + 5.0,\ x\ (1.0 + 1.0\ i) - 3.0 - 2.0\ i,\ 1.0,\ x\ (1.0 - 1.0\ i) - 3.0 + 2.0\ i]$$

```
numeric::fft(L, Symbolic)
```

$$[2\ x + 5,\ x\ (1 + i) - 3 - 2\ i,\ 1,\ x\ (1 - i) - 3 + 2\ i]$$

```
delete L, F:
```

Example 2

We give a demonstration of multi-dimensional transformations. First, a 2-dimensional transformation is computed by using an array with 2 indices:

```
A := array(1..2, 1..4, [[1, 2, 3, 4], [a, b, c, d]]):
numeric::fft(A, Symbolic)
```

$$\begin{pmatrix} a+b+c+d+10 & a-b\,i-c+d\,i-2+2\,i & a-b+c-d-2 & a+b\,i-c-d\,i-2-2\,i \\ 10-b-c-d-a & c+b\,i-a-d\,i-2+2\,i & b-a-c+d-2 & c-b\,i-a+d\,i-2-2\,i \end{pmatrix}$$

```
numeric::invfft(%, Symbolic)
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ a & b & c & d \end{pmatrix}$$

The next example is 3-dimensional as indicated by the format of the array:

```
A := array(1..2, 1..4, 1..2,
          [[[sin(j1*PI/2)*cos(j2*3*PI/4)*sin(j3*PI/2)
          $ j3 = 1..2 ] $ j2 = 1..4 ] $ j1 = 1..2]):
numeric::fft(A)
```

```
array(1..2, 1..4, 1..2,
  (1, 1, 1) = -1.0

  (1, 1, 2) = -1.0

  (1, 2, 1) = - 1.414213562 - 1.0 I

  (1, 2, 2) = - 1.414213562 - 1.0 I

  (1, 3, 1) = 1.0

  (1, 3, 2) = 1.0

  (1, 4, 1) = - 1.414213562 + 1.0 I

  (1, 4, 2) = - 1.414213562 + 1.0 I

  (2, 1, 1) = -1.0

  (2, 1, 2) = -1.0

  (2, 2, 1) = - 1.414213562 - 1.0 I
```

```
  (2, 2, 2) = - 1.414213562 - 1.0 I

  (2, 3, 1) = 1.0

  (2, 3, 2) = 1.0

  (2, 4, 1) = - 1.414213562 + 1.0 I

  (2, 4, 2) = - 1.414213562 + 1.0 I
)
```

```
delete A:
```

Example 3

Data of arbitrary length can be transformed:

```
L := [1, 2 + I, PI/3]:
numeric::fft(L)
```

$$[4.047197551 + 1.0\,i,\ 0.3424266282 - 1.325151125\,i,\ -1.389624179 + 0.3251511255\,i]$$

```
delete L:
```

Parameters

L

A list or a 1-dimensional array(`1 .. N, [Symbol::hellip]`) or a 1-dimensional hfarray(`1 .. N, [Symbol::hellip]`) of arithmetical expressions.

M

A `matrix` of category `Cat::Matrix` of arithmetical expressions.

A

A *d*-dimensional `array( 1..n_1,Symbol::hellip,1..n_d, [Symbol::hellip] )` or a *d*-dimensional `hfarray( 1..n_1,Symbol::hellip,1..n_d, [Symbol::hellip] )` of arithmetical expressions.

mode

One of the flags Hard, HardwareFloats, Soft, SoftwareFloats, or Symbolic

Options

Hard, HardwareFloats, Soft, SoftwareFloats

With Hard (or HardwareFloats), computations are done using fast hardware float arithmetic from within a MuPAD session. Hard and HardwareFloats are equivalent. With this option, the input data are converted to hardware floats and processed by compiled C code. The result is reconverted to MuPAD floats and returned to the MuPAD session.

With Soft (or SoftwareFloats) computations are dome using software float arithmetic provided by the MuPAD kernel. Soft and SoftwareFloats are equivalent. SoftwareFloats is used by default if the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY.

Compared to the SoftwareFloats used by the MuPAD kernel, the computation with HardwareFloats may be many times faster. Note, however, that the precision of hardware arithmetic is limited to about 15 digits. Further, the size of floating-point numbers may not be larger than approximately 10^{308} and not smaller than approximately 10^{-308}.

If no HardwareFloats or SoftwareFloats are requested explicitly, the following strategy is used: If the current value of DIGITS is smaller than 16 or if the matrix A is a hardware float array of domain type DOM_HFARRAY, then hardware arithmetic is tried. If this is successful, the result is returned.

If the result cannot be computed with hardware floats, software arithmetic by the MuPAD kernel is tried.

If the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY, or if one of the options Soft, SoftwareFloats or Symbolic is specified, MuPAD computes the result with its software arithmetic without trying to use hardware floats first.

There may be several reasons for hardware arithmetic to fail:

- The current value of `DIGITS` is larger than 15.
- The data contains symbolic objects.
- The data contains numbers larger than 10^{308} or smaller than 10^{-308} that cannot be represented by hardware floats.

If neither `HardwareFloats` nor `SoftwareFloats` is specified, the user is not informed whether hardware floats or software floats are used.

If `HardwareFloats` are specified but fail due to one of the reasons above, a warning is issued that the (much slower) software floating-point arithmetic of the MuPAD kernel is used.

Note that `HardwareFloats` can only be used if all input data can be converted to floating-point numbers.

With `Soft` and `SoftwareFloats`, symbolic objects are accepted even if they cannot be converted to floating-point numbers. The result consists of arithmetical expressions involving both floating-point numbers as well as symbolic objects. See. “Example 1” on page 19-89.

The trailing digits in floating-point results computed with `HardwareFloats` and `SoftwareFloats` may differ.

`Symbolic`

Without this option, the floating-point converter `float` is applied to all input data. Use this option if no such conversion is desired. Exact arithmetic is used to compute the Fourier transformation.

This option prevents conversion of the input data to floats.

`ReturnType`

Option, specified as `ReturnType = t`

Return the result in a container of domain type `t`. The following return types `t` are available: `DOM_LIST`, or `DOM_ARRAY`, or `DOM_HFARRAY`, or `matrix`, or `densematrix`.

This option determines the domain type `t` of the result.

If no return type is specified by this option, the result if of the same type and format as the input data.

If the return type `DOM_LIST` is specified, the result is always a plain list of floating-point numbers. If the input data are given by a matrix or a multi-dimensional array, the returned list represents the operands of the multi-dimensional Fourier data. E.g., if an $n_1 \times n_2$ matrix is entered, the return value is a list with $n_1\, n_2$ values representing the entries of a $n_1 \times n_2$ matrix. The first n_2 entries of the list represent the first row of the result, the next n_2 entries represent the second row, etc.

With `ReturnType = matrix` or `ReturnType = densematrix`, only the results of 1 and 2 dimensional Fourier transformations can be represented.

`Clean`

Reduce roundoff garbage in the result. All entries of the result with absolute values smaller than `10^(-DIGITS)` times the maximal absolute value of all operands of the result are set to `0.0`. Further, the routine `numeric::complexRound` is applied to all entries of the result.

Note: The postprocessing of the result is done on the software float level. When using hardware floats, this option may increase the runtime significantly!

This option is ignored when used in conjunction with the option `Symbolic`.

Return Values

List/array/hfarray/matrix of the same length and format as the first input parameter L/A/M. The type of the return value can be changed with the option `ReturnType`.

See Also

MuPAD Functions
`numeric::invfft`

More About

- "Discrete Fourier Transforms"

numeric::invfft

Inverse Fast Fourier Transform

Syntax

```
numeric::invfft(L, <mode>, <ReturnType = t>, <Clean>)
numeric::invfft(M, <mode>, <ReturnType = t>, <Clean>)
numeric::invfft(A, <mode>, <ReturnType = t>, <Clean>)
```

Description

`numeric::invfft(data)` returns the inverse discrete Fourier transform.

The inverse discrete Fourier transform $L = invfft(F)$ of N data elements F_k stored in the list $F = [F_1, \ldots, F_N]$ is the list $L = [L_1, \ldots, L_N]$ given by

$$L_j = \frac{1}{N} \sum_{k=1}^{N} F_k \, e^{\frac{i\,2\,\pi\,(j-1)\,(k-1)}{N}}, \; j = 1, \ldots, N$$

.

`invfft` transforms the data by a Fast Fourier Transform (FFT) algorithm.

The d-dimensional inverse discrete Fourier transform $A = invfft(F)$ is given by

$$A_{j_1,\ldots,j_d} = \frac{1}{N} \sum_{k_1=1}^{n_1} \cdots \sum_{j_d=1}^{n_d} F_{k_1,\ldots,k_d} \, e^{i\,2\,\pi \left(\frac{(j_1-1)\,(k_1-1)}{n_1} + \ldots + \frac{(j_d-1)\,(k_d-1)}{n_d} \right)}$$

with $j_1 = 1, \ldots, n_1, \ldots, j_d = 1, \ldots, n_d$.

Data provided by lists or 1-dimensional `array`s or `hfarray`s are transformed according to the 1-dimensional transform. Data provided by matrices are transformed according to the 2-dimensional transform. Data provided by multidimensional arrays or hfarrays

are transformed according to the multidimensional transform matching the format of the input array.

If the data size factorizes ($N = p\,q$, say), the inverse discrete Fourier transform can be computed by p different inverse Fourier transforms of subsets of the data, each subset having the data size q. The corresponding 'divide and conquer' algorithm is known as FFT ('Fast Fourier Transform'). The `invfft` routine employs the FFT algorithm. It is most efficient, when the data size N is an integer power of 2 ('radix 2 FFT'). In this case, the algorithm needs $O(N \log_2(N))$ elementary operations.

Note: More generally, FFT is efficient, if the data size is the product of many small factors!

Following Bluestein, the inverse Fourier transform is written as a convolution if the data size N is a prime. The data are zero-padded to a data length that is an integer power of 2. The convolution is then computed via radix 2 FFTs. Thus, the algorithm needs $O(N \log_2(N))$ elementary operations even if N is a prime.

Environment Interactions

Without the option `Symbolic`, the function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We give a demonstration of 1-dimensional transformations using lists. By default, numerical expressions are converted to floats:

```
L := [1, 2^(1/2), 3*I, PI]:
F := numeric::fft(L)
```

$$[5.555806216 + 3.0\,i,\ 1.0 - 1.272620909\,i,\ -3.555806216 + 3.0\,i,\ 1.0 - 4.727379091\,i]$$

```
numeric::invfft(F)
```

$$\left[1.0,\ 1.414213562 + 2.512840965\ 10^{-17}\ i,\ 3.0\ i,\ 3.141592654 - 2.512840965\ 10^{-17}\ i\right]$$

```
numeric::invfft(F, Clean)
```

$$[1.0,\ 1.414213562,\ 3.0\ i,\ 3.141592654]$$

Exact arithmetic is used with the option `Symbolic`:

```
F := numeric::fft(L, Symbolic)
```

$$[\pi + \sqrt{2} + 1 + 3\ i,\ \pi\ i - \sqrt{2}\ i + 1 - 3\ i,\ 1 - \sqrt{2} - \pi + 3\ i,\ 1 + \sqrt{2}\ i - \pi\ i - 3\ i]$$

```
numeric::invfft(F, Symbolic)
```

$$[1,\ \sqrt{2},\ 3\ i,\ \pi]$$

Symbolic expressions are accepted. Internally, however, the default method `HardwareFloats` (with `DIGITS < 16`) fails because of the symbolic parameter `x`. The following results are computed with the software arithmetic provided by the MuPAD kernel:

```
L := [x, 2, 3, x]:
numeric::fft(L)
```

$$[2\ x + 5.0,\ x\ (1.0 + 1.0\ i) - 3.0 - 2.0\ i,\ 1.0,\ x\ (1.0 - 1.0\ i) - 3.0 + 2.0\ i]$$

```
numeric::fft(L, Symbolic)
```

$$[2\ x + 5,\ x\ (1 + i) - 3 - 2\ i,\ 1,\ x\ (1 - i) - 3 + 2\ i]$$

```
delete L, F:
```

Example 2

We give a demonstration of multi-dimensional transformations. First, a 2-dimensional transformation is computed by using an array with 2 indices:

```
A := array(1..2, 1..4, [[1, 2, 3, 4], [a, b, c, d]]):
numeric::fft(A, Symbolic)
```

$$\begin{pmatrix} a+b+c+d+10 & a-b\,i-c+d\,i-2+2\,i & a-b+c-d-2 & a+b\,i-c-d\,i-2-2\,i \\ 10-b-c-d-a & c+b\,i-a-d\,i-2+2\,i & b-a-c+d-2 & c-b\,i-a+d\,i-2-2\,i \end{pmatrix}$$

```
numeric::invfft(%, Symbolic)
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ a & b & c & d \end{pmatrix}$$

The next example is 3-dimensional as indicated by the format of the array:

```
A := array(1..2, 1..4, 1..2,
          [[[sin(j1*PI/2)*cos(j2*3*PI/4)*sin(j3*PI/2)
          $ j3 = 1..2 ] $ j2 = 1..4 ] $ j1 = 1..2]):
numeric::fft(A)
```

```
array(1..2, 1..4, 1..2,
  (1, 1, 1) = -1.0

  (1, 1, 2) = -1.0

  (1, 2, 1) = - 1.414213562 - 1.0 I

  (1, 2, 2) = - 1.414213562 - 1.0 I

  (1, 3, 1) = 1.0

  (1, 3, 2) = 1.0

  (1, 4, 1) = - 1.414213562 + 1.0 I

  (1, 4, 2) = - 1.414213562 + 1.0 I

  (2, 1, 1) = -1.0

  (2, 1, 2) = -1.0

  (2, 2, 1) = - 1.414213562 - 1.0 I
```

```
  (2, 2, 2) = - 1.414213562 - 1.0 I

  (2, 3, 1) = 1.0

  (2, 3, 2) = 1.0

  (2, 4, 1) = - 1.414213562 + 1.0 I

  (2, 4, 2) = - 1.414213562 + 1.0 I
)
```

```
delete A:
```

Example 3

Data of arbitrary length can be transformed:

```
L := [1, 2 + I, PI/3]:
numeric::fft(L)
```

$$[4.047197551 + 1.0\,i,\ 0.3424266282 - 1.325151125\,i,\ -1.389624179 + 0.3251511255\,i]$$

```
delete L:
```

Parameters

L

A list or a 1-dimensional array(`1 .. N, [Symbol::hellip]`) or a 1-dimensional hfarray(`1 .. N, [Symbol::hellip]`) of arithmetical expressions.

M

A `matrix` of category `Cat::Matrix` of arithmetical expressions.

A

A *d*-dimensional `array( 1..n_1,Symbol::hellip,1..n_d, [Symbol::hellip] )` or a *d*-dimensional `hfarray( 1..n_1,Symbol::hellip,1..n_d, [Symbol::hellip] )` of arithmetical expressions.

mode

One of the flags Hard, HardwareFloats, Soft, SoftwareFloats, or Symbolic

Options

Hard, HardwareFloats, Soft, SoftwareFloats

With Hard (or HardwareFloats), computations are done using fast hardware float arithmetic from within a MuPAD session. Hard and HardwareFloats are equivalent. With this option, the input data are converted to hardware floats and processed by compiled C code. The result is reconverted to MuPAD floats and returned to the MuPAD session.

With Soft (or SoftwareFloats) computations are dome using software float arithmetic provided by the MuPAD kernel. Soft and SoftwareFloats are equivalent. SoftwareFloats is used by default if the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY.

Compared to the SoftwareFloats used by the MuPAD kernel, the computation with HardwareFloats may be many times faster. Note, however, that the precision of hardware arithmetic is limited to about 15 digits. Further, the size of floating-point numbers may not be larger than approximately 10^{308} and not smaller than approximately 10^{-308}.

If no HardwareFloats or SoftwareFloats are requested explicitly, the following strategy is used: If the current value of DIGITS is smaller than 16 or if the matrix A is a hardware float array of domain type DOM_HFARRAY, then hardware arithmetic is tried. If this is successful, the result is returned.

If the result cannot be computed with hardware floats, software arithmetic by the MuPAD kernel is tried.

If the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY, or if one of the options Soft, SoftwareFloats or Symbolic is specified, MuPAD computes the result with its software arithmetic without trying to use hardware floats first.

There may be several reasons for hardware arithmetic to fail:

- The current value of `DIGITS` is larger than 15.
- The data contains symbolic objects.
- The data contains numbers larger than 10^{308} or smaller than 10^{-308} that cannot be represented by hardware floats.

If neither `HardwareFloats` nor `SoftwareFloats` is specified, the user is not informed whether hardware floats or software floats are used.

If `HardwareFloats` are specified but fail due to one of the reasons above, a warning is issued that the (much slower) software floating-point arithmetic of the MuPAD kernel is used.

Note that `HardwareFloats` can only be used if all input data can be converted to floating-point numbers.

With `Soft` and `SoftwareFloats`, symbolic objects are accepted even if they cannot be converted to floating-point numbers. The result consists of arithmetical expressions involving both floating-point numbers as well as symbolic objects. See. “Example 1” on page 19-97.

The trailing digits in floating-point results computed with `HardwareFloats` and `SoftwareFloats` may differ.

`Symbolic`

Without this option, the floating-point converter `float` is applied to all input data. Use this option if no such conversion is desired. Exact arithmetic is used to compute the Fourier transformation.

This option prevents conversion of the input data to floats.

`ReturnType`

Option, specified as `ReturnType = t`

Return the result in a container of domain type `t`. The following return types `t` are available: `DOM_LIST`, or `DOM_ARRAY`, or `DOM_HFARRAY`, or `matrix`, or `densematrix`.

This option determines the domain type `t` of the result.

If no return type is specified by this option, the result if of the same type and format as the input data.

If the return type `DOM_LIST` is specified, the result is always a plain list of floating-point numbers. If the input data are given by a matrix or a multi-dimensional array, the returned list represents the operands of the multi-dimensional Fourier data. E.g., if an $n_1 \times n_2$ matrix is entered, the return value is a list with $n_1\, n_2$ values representing the entries of a $n_1 \times n_2$ matrix. The first n_2 entries of the list represent the first row of the result, the next n_2 entries represent the second row, etc.

With `ReturnType = matrix` or `ReturnType = densematrix`, only the results of 1 and 2 dimensional Fourier transformations can be represented.

`Clean`

Reduce roundoff garbage in the result. All entries of the result with absolute values smaller than `10^(-DIGITS)` times the maximal absolute value of all operands of the result are set to `0.0`. Further, the routine `numeric::complexRound` is applied to all entries of the result.

Note: The postprocessing of the result is done on the software float level. When using hardware floats, this option may increase the runtime significantly!

This option is ignored when used in conjunction with the option `Symbolic`.

Return Values

List/array/hfarray/matrix of the same length and format as the first input parameter L/A/M. The type of the return value can be changed with the option `ReturnType`.

See Also

MuPAD Functions
`numeric::fft`

More About

- "Discrete Fourier Transforms"

numeric::fMatrix

Functional calculus for numerical square matrices

Syntax

```
numeric::fMatrix(f, A, p1, p2, …, options)
```

Description

`numeric::fMatrix(f, A)` computes the matrix $f(A)$ with a function f and a square matrix A.

If no return type is specified via the option `ReturnType = d`, the domain type of the result depends on the type of the input matrix `A`:

- For an array A, the result is returned as an array.
- For an hfarray A, the result is returned as an array.
- For a dense matrix A of type `Dom::DenseMatrix()` the result is a dense matrix of type `Dom::DenseMatrix()` over the ring of MuPAD expressions.
- For all other matrices A of category `Cat::Matrix`, the result is returned as a `matrix` of type `Dom::Matrix()` over the ring of MuPAD expressions. This includes input matrices `A` of type `Dom::Matrix(...)`, `Dom::SquareMatrix(...)`, `Dom::MatrixGroup(...)` etc.

The components of A must not contain symbolic objects which cannot be converted to numerical values via `float`. Numerical symbolic expressions such as π, $\sqrt{2}$, e^{-1} etc. are accepted. They are converted to floats.

Note: When you use `numeric::fMatrix`, the matrix must be diagonalizable, and all its elements must be convertible to floating-point numbers. Otherwise, use `funm`.

If `numeric::fMatrix` detects numerically that A is not diagonalizable, it aborts with an error message . Nevertheless, the numerical algorithm often fails to detect that the matrix is not diagonalizable, and the returned matrix is dominated by round-off

effects. When using`numeric::fMatrix`, ensure the diagonalization is possible and well conditioned.

Symmetric/Hermitian and skew/skew Hermitian matrices can always be diagonalized in a numerically stable way; `numeric::fMatrix` produces reliable numerical results for such matrices.

The procedure *f* must accept complex floating-point numbers as first argument. It may return arbitrary MuPAD expressions, provided these can be multiplied with floating-point numbers.

The parameters p_1, p_2, ... may be numerical or symbolic objects. They must be accepted by `f` as 2nd argument, 3rd argument etc.

In contrast to the components of *A*, numerical symbolic objects such as π, $\sqrt{2}$ etc. passed as parameters p_1, p_2, ... are not converted to floats.

Inversion or exponentiation of a matrix may be realized with the functions $a \to \frac{1}{a}$ and `exp`, respectively. However, it is recommended to use the specialized algorithms `numeric::inverse` and `numeric::expMatrix` instead. Also matrix evaluation of low degree polynomials should be done with standard matrix arithmetic rather than with `numeric::fMatrix`.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We compute the matrix power A^{100}:

```
A := array(1..2, 1..2, [[2, PI], [exp(-10), 0]]):
numeric::fMatrix(x -> x^100, A)
```

$$\begin{pmatrix} 1.272133133\ 10^{30} & 1.998190806\ 10^{30} \\ 2.887634784\ 10^{25} & 4.535724387\ 10^{25} \end{pmatrix}$$

Alternatively, you may use the function `_power` which takes the exponent as a second parameter:

```
numeric::fMatrix(_power, A, 100)
```

$$\begin{pmatrix} 1.272133133\ 10^{30} & 1.998190806\ 10^{30} \\ 2.887634784\ 10^{25} & 4.535724387\ 10^{25} \end{pmatrix}$$

```
delete A:
```

Example 2

We compute the square root of a matrix:

```
A := matrix([[0, 1], [-1, 1]]):
B := numeric::fMatrix(sqrt, A)
```

$$\begin{pmatrix} 0.5773502692 & 0.5773502692 + 1.110223025\ 10^{-16}\,i \\ -0.5773502692 + 5.551115123\ 10^{-17}\,i & 1.154700538 - 1.110223025\ 10^{-16}\,i \end{pmatrix}$$

The small imaginary parts are caused by numerical round-off. We eliminate them by extracting the real parts of the components:

```
B := map(B, Re)
```

$$\begin{pmatrix} 0.5773502692 & 0.5773502692 \\ -0.5773502692 & 1.154700538 \end{pmatrix}$$

We verify that `B^2` is `A`. Since A was passed as a matrix of type `Dom::Matrix()`, the matrix B is also of this type. We may compute the square by the overloaded standard arithmetic using the operator `^`:

```
B^2
```

$$\begin{pmatrix} -1.283695372\,10^{-16} & 1.0 \\ -1.0 & 1.0 \end{pmatrix}$$

```
delete A, B:
```

Example 3

We compute $e^{t\,\pi\,A}$ with a symbolic parameter t:

```
A := array(1..2, 1..2, [[0, 1], [-1, 0]]):
numeric::fMatrix(exp@_mult, A, t*PI)
```

$$\begin{pmatrix} \sigma_1 & -0.5\,e^{1.0\,\pi\,t\,i}\,i + 0.5\,e^{-1.0\,\pi\,t\,i}\,i \\ 0.5\,e^{1.0\,\pi\,t\,i}\,i - 0.5\,e^{-1.0\,\pi\,t\,i}\,i & \sigma_1 \end{pmatrix}$$

where

$$\sigma_1 = 0.5\,e^{1.0\,\pi\,t\,i} + 0.5\,e^{-1.0\,\pi\,t\,i}$$

```
delete A:
```

Example 4

We demonstrate the difference between `HardwareFloats` and `SoftwareFloats`. The diagonalization of the following matrix is ill-conditioned. The result is dominated by round-off effects:

```
A := array(1..3, 1..3, [[10, 1,    1    ],
                        [ 0, 1,    1    ],
                        [ 1, 0, 10^(-14)]]):
numeric::fMatrix(ln, A, SoftwareFloats)
```

$$\begin{pmatrix} 2.284572396 & 0.2635466905 & 0.2635466905 \\ -4.117444729 & 4.03009691 & 41.43799398 \\ 4.380991419 & -4.117444729 & -41.5253418 \end{pmatrix}$$

```
numeric::fMatrix(ln, A, HardwareFloats)
```

$$\begin{pmatrix} 2.284572396 & 0.2635466905 & 0.2635466905 \\ -3.532353481 & 3.445005663 & 35.5870815 \\ 3.795900172 & -3.532353481 & -35.67442932 \end{pmatrix}$$

In the following case, the round-off effects of `SoftwareFloats` makes the algorithm think that the matrix cannot be diagonalized. Consequently, `FAIL` is returned. With `HardwareFloats`, however, a result is computed:

```
A := array(1..3, 1..3, [[   1     , 1 ,    1    ],
                        [   0     , 1 ,    1    ],
                        [ 10^(-30), 0 , 10^(-30)]]):
numeric::fMatrix(ln, A, SoftwareFloats)
```

FAIL

```
numeric::fMatrix(ln, A, HardwareFloats)
```

$$\begin{pmatrix} -5.469808086\ 10^{-17} & 1.000000477 & 0.9999345993 \\ 9.98400946\ 10^{-31} & -5.632422161\ 10^{-17} & 69.07755279 \\ 0 & 0 & -69.07755279 \end{pmatrix}$$

```
delete A:
```

Parameters

f

A procedure representing a scalar function $f\colon \mathbb{C} \to \mathbb{C}$ or $f\colon \mathbb{C} \times P \times \cdots \times P \to \mathbb{C}$, where P is a set of parameters

A

A square matrix of domain type DOM_ARRAY, DOM_HFARRAY, or of category Cat::Matrix

p_1, p_2, …

Arbitrary MuPAD objects accepted by *f* as additional input parameters

Options

Hard, HardwareFloats, Soft, SoftwareFloats

With Hard (or HardwareFloats), computations are done using fast hardware float arithmetic from within a MuPAD session. Hard and HardwareFloats are equivalent. With this option, the input data are converted to hardware floats and processed by compiled C code. The result is reconverted to MuPAD floats and returned to the MuPAD session.

With Soft (or SoftwareFloats) computations are dome using software float arithmetic provided by the MuPAD kernel. Soft and SoftwareFloats are equivalent. SoftwareFloats is used by default if the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY.

Compared to the SoftwareFloats used by the MuPAD kernel, the computation with HardwareFloats may be many times faster. Note, however, that the precision of hardware arithmetic is limited to about 15 digits. Further, the size of floating-point numbers may not be larger than approximately 10^{308} and not smaller than approximately 10^{-308}.

If no HardwareFloats or SoftwareFloats are requested explicitly, the following strategy is used: If the current value of DIGITS is smaller than 16 or if the matrix A is a hardware float array of domain type DOM_HFARRAY, then hardware arithmetic is tried. If this is successful, the result is returned.

If the result cannot be computed with hardware floats, software arithmetic by the MuPAD kernel is tried.

If the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY, or if one of the options Soft, SoftwareFloats or Symbolic is

specified, MuPAD computes the result with its software arithmetic without trying to use hardware floats first.

There may be several reasons for hardware arithmetic to fail:

- The current value of `DIGITS` is larger than 15.
- The data contains symbolic objects.
- The data contains numbers larger than 10^{308} or smaller than 10^{-308} that cannot be represented by hardware floats.

If neither `HardwareFloats` nor `SoftwareFloats` is specified, the user is not informed whether hardware floats or software floats are used.

If `HardwareFloats` are specified but fail due to one of the reasons above, a warning is issued that the (much slower) software floating-point arithmetic of the MuPAD kernel is used.

Note: For ill-conditioned matrices, the result is subject to round-off errors. The results returned with `HardwareFloats` and `SoftwareFloats` may differ! See "Example 4" on page 19-107.

`NoWarning`

Suppresses warnings

`ReturnType`

Option, specified as `ReturnType = d`

Return the result as a matrix of domain type `d`. The following return types are available: `DOM_ARRAY`, `DOM_HFARRAY`, `Dom::Matrix()`, or `Dom::DenseMatrix()`.

Return Values

Depending on the type of the input matrix `A`, the matrix $f(A)$ is returned as a matrix of type `DOM_ARRAY`, `DOM_HFARRAY`, `Dom::Matrix()` or `Dom::DenseMatrix()`. If the algorithm thinks that `A` is not diagonalizable, then `FAIL` is returned.

Algorithms

A numerical diagonalization $A = X\,diag(\lambda_1, \lambda_2, \ldots)\,X^{-1}$ is computed. The columns of X are the (right) eigenvectors of A, the diagonal entries $\lambda_1, \lambda_2, \ldots$ are the corresponding eigenvalues. The function f is mapped to the eigenvalues, the matrix result is computed by

$$f(A, p_1, p_2, \ldots) = X\ \mathrm{diag}\big(f(\lambda_1, p_1, p_2, \ldots), f(\lambda_2, p_1, p_2, \ldots), \ldots\big)\ X^{-1}$$

.

The eigenvector matrix X may be obtained via `numeric::eigenvectors(A)[2]`.

The condition number $\|X\|_\infty \left\|\frac{1}{X}\right\|_\infty$ of the eigenvector matrix is a measure indicating how well conditioned the diagonalization of the matrix A is. If this number is larger than 10^{DIGITS}, then not a single digit of the diagonalization data is trustworthy.

The call `numeric::fMatrix(exp, A)` corresponds to `numeric::expMatrix(A, Diagonalization)`.

See Also

MuPAD Functions

`funm` | `linalg::sqrtMatrix` | `numeric::expMatrix` | `numeric::inverse`

numeric::fsolve

Search for a numerical root of a system of equations

Syntax

```
numeric::fsolve(eq, x, options)
numeric::fsolve(eq, x = a, options)
numeric::fsolve(eq, x = a .. b, options)
numeric::fsolve(eqs, [x1, x2, …], options)
numeric::fsolve(eqs, {x1, x2, …}, options)
numeric::fsolve(eqs, [x1 = a1, x2 = a2, …], options)
numeric::fsolve(eqs, {x1 = a1, x2 = a2, …}, options)
numeric::fsolve(eqs, [x1 = a1 .. b1, x2 = a2 .. b2, …], options)
numeric::fsolve(eqs, {x1 = a1 .. b1, x2 = a2 .. b2, …}, options)
```

Description

`numeric::fsolve(eqs, ...)` returns a numerical approximation of a solution of the system of equations `eqs`.

This is the MuPAD numerical solver for non-linear systems of equations.

Note: By default, this routine returns only *one* numerical solution!

The equations must not contain symbolic objects other than the unknowns that cannot be converted to numerical values via `float`. Symbolic objects such as π or $\sqrt{2}$ etc. are accepted. The same holds true for starting values and search ranges. Search ranges may contain $\pm\infty$. Cf. “Example 2” on page 19-116.

`numeric::fsolve` implements a purely numerical Newton type root search with a working precision set by the environment variable `DIGITS`. Well separated simple roots

should be exact within this precision. However, multiple roots or badly separated roots may be computed with a restricted precision. Cf. "Example 3" on page 19-116.

Note: For systems of equations, the expressions defining the equations must have a symbolic derivative!

Overdetermined systems (i.e., more equations than indeterminates) are not accepted. However, there may be more indeterminates than equations. Cf. "Example 4" on page 19-117.

Specifying indeterminates `[x1, x2, ...]` without starting values or search ranges is equivalent to the search ranges `[x_1 = -infinity .. infinity, x_2 = -infinity .. infinity, dots]`. Note, however, that the user should assist `numeric::fsolve` by providing specific search ranges whenever possible! If a complex starting point or a search range involving a complex number is specified for at least one of the unknowns, the search is extended to the entire complex plane for all variables for which no explicit search interval is given.

For real equations and real starting points or search ranges, the internal Newton iteration will usually produce real values, i.e., `numeric::fsolve` searches for real roots only (unless square roots, logarithms etc. happen to produce complex values from real input). Use complex starting points or search ranges to search for complex roots of real equations. Cf. "Example 5" on page 19-117.

Starting values and search ranges can be mixed. Cf. "Example 6" on page 19-118.

Search ranges should only be provided if a solution is known to exist inside the search range. Otherwise, the search may take some time before `numeric::fsolve` gives up.

Specification of a search range primarily means that starting points from this range are used for the internal Newton search. For sufficiently small search ranges enclosing a solution the search will usually pick out this solution. However, it may also happen that the Newton iteration drifts towards other solutions.

With the default search strategy `RestrictedSearch`, only solutions from the search range are accepted, even if solutions outside the search range are found internally. More specifically, if a search range such as `x = a .. b` is specified for the variable `x`, then solutions satisfying `min(#(a), #(b)) ≤ #(x) ≤ max(#(a), #(b))` and `min(#(a), #(b)) ≤ #(x) ≤ max(#(a), #(b))` are searched for. Thus, the values `a, b` specify

the bottom left and top right corner of a rectangular search area in the complex plane when the `RestrictedSearch` strategy is used.

With the search strategy `UnrestrictedSearch`, any solution inside or outside the search range is accepted and returned. Cf. "Example 7" on page 19-118.

If starting values for all indeterminates are provided, then a *single* Newton iteration with these initial data is launched. It either leads to a solution or `numeric::fsolve` gives up and returns `FAIL`. The same holds true if search ranges `x = a .. a` or `[x_1 = a_1 .. a_1, x_2 = a_2 .. a_2, dots]` of zero length are specified.

Note: The risk of failure is high when providing bad starting values! Starting values are appropriate only if a sufficiently good approximation of the solution is known! On the other hand, providing good starting values is the fastest way to a solution. Cf. "Example 8" on page 19-119.

If at least one of the indeterminates has a non-trivial search range, then `numeric::fsolve` uses *several* Newton iterations with different starting values from the search range. Cf. "Example 9" on page 19-119. Search ranges in conjunction with the option `UnrestrictedSearch` provide a higher chance of detecting roots than (bad) starting values!

Note: User defined assumptions such as `assume(x > 0)` are not taken into account in the numerical search! Provide search ranges instead! Cf. "Example 2" on page 19-116.

Note: Convergence may be slow for multiple roots! Furthermore, `numeric::fsolve` may fail to detect such roots!

Use `linsolve` or `numeric::linsolve` for systems of *linear* equations.

Use `numeric::realroots`, if *all real roots* of a single non-polynomial real equation in a finite range are desired.

Use `polylib::realroots`, if *all real roots* of a real univariate polynomial are desired.

Use `numeric::polyroots`, if *all real and complex roots* of a univariate polynomial are desired.

Use `numeric::solve`, if *all roots* of a multivariate polynomial system are desired.

The routine `numeric::solve` provides a common interface to all these numerical solvers.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We compute roots of the sine function:

```
numeric::fsolve(sin(x) = 0, x)
```

$$[x = 0.0]$$

With the option `Random`, several calls may result in different roots:

```
numeric::fsolve(sin(x), x, Random)
```

$$[x = -226.1946711]$$

```
numeric::fsolve(sin(x), x, Random)
```

$$[x = 97.38937226]$$

Particular solutions can be chosen by an appropriate starting point close to the wanted solution, or by a search interval:

```
numeric::fsolve(sin(x), x = 3),
numeric::fsolve(sin(x), x = -4 .. -3)
```

$$[x = 3.141592653], [x = -3.141592654]$$

The solutions found by `numeric::fsolve` can be used in `subs` and `assign` to substitute or assign the indeterminates:

```
eqs := [x^2 = sin(y), y^2 = cos(x)]:
solution := numeric::fsolve(eqs, [x, y])
```

$$[x = -0.8517004887,\ y = 0.8116062151]$$

```
eval(subs(eqs, solution))
```

$$[0.7253937224 = 0.7253937224,\ 0.6587046485 = 0.6587046485]$$

```
assign(solution): x, y
```

$$-0.8517004887,\ 0.8116062151$$

```
delete eqs, solution, x, y:
```

Example 2

We demonstrate the use of search ranges. The following system has solutions with positive and negative x. The solution with $x \geq 0$ is obtained with the search interval `x = 0 .. infinity`:

```
numeric::fsolve([x^2 = exp(x*y), x^2 = y^2], [x = 0 .. infinity, y])
```

$$[x = 0.753089165,\ y = -0.753089165]$$

We search for a solution with $x \leq 0$:

```
numeric::fsolve([x^2 = exp(x*y), x^2 = y^2], [x = -infinity .. 0, y])
```

$$[x = -0.753089165,\ y = 0.753089165]$$

Example 3

Multiple roots can only be computed with a restricted precision:

```
numeric::fsolve(expand((x - 1/3)^5), x = 0.3)
```

$$[x = 0.3333672906]$$

Example 4

The following system of equations is degenerate and has a 1-parameter family of solutions. Each call to `numeric::fsolve` picks out one random solution:

```
numeric::fsolve([x^2 - y^2, x^2 - y^2], [x, y], Random) $ i = 1 .. 3
```

$$[x = 34.70258251,\ y = 34.70258251],\ [x = -29.16650501,\ y = 29.16650501],\ [x = 5.933941324,\ y = -5.933941324]$$

The equation may also be specified as an underdetermined system:

```
numeric::fsolve([x^2 - y^2], [x, y])
```

$$[x = 0.0,\ y = 0.0]$$

Example 5

The following equation has no real solution. Consequently, the numerical search with real starting values fails:

```
numeric::fsolve(sin(x) + cos(x)^2 = 3, x)
```

FAIL

With a complex starting value, a solution is found:

```
numeric::fsolve(sin(x) + cos(x)^2 = 3, x = I)
```

$$[x = 0.2972513613 + 1.128383965\ i]$$

Also complex search ranges may be specified. In the following, the internal starting point is a random value on the line from `2 + I` to `3 + 2*I`. Solutions are accepted if they lie in the complex rectangle with the bottom left corner `2 + I` and the top right corner `3 + 2*I`:

```
numeric::fsolve(sin(x) + cos(x)^2 = 3, x = 2 + I .. 3 + 2*I)
```

$$[x = 2.844341292 + 1.128383965\,i]$$

Example 6

Starting values and search intervals can be mixed:

```
numeric::fsolve([x^2 + y^2 = 1, y^2 + z^2 = 1, x^2 + z^2 = 1],
                [x = 1, y = 0 .. 10, z])
```

$$[x = 0.7071067812,\ y = 0.7071067812,\ z = 0.7071067812]$$

Example 7

With `UnrestrictedSearch`, search intervals are only used for choosing starting values for the internal Newton search. The numerical iteration may drift towards a solution outside the search range:

```
eqs := [x*sin(10*x) = y^3, y^2 = exp(-2*x/3)]:
numeric::fsolve(eqs, [x = 0 .. 1, y = -1 .. 0], UnrestrictedSearch)
```

$$[x = 1.232766201,\ y = -0.6630386021]$$

With the default strategy `RestrictedSearch`, only solutions inside the search range are accepted:

```
numeric::fsolve(eqs, [x = 0 .. 1, y = -1 .. 0])
```

$$[x = 0.9816416007,\ y = -0.7209295436]$$

In the last search, also the previous solution outside the search range was found. With the option `MultiSolutions`, `numeric::fsolve` returns a sequence of all solutions that were found in the internal search:

```
numeric::fsolve(eqs, [x = 0 .. 1, y = -1 .. 0], MultiSolutions)
```

$$[x = 0.9816416007,\ y = -0.7209295436],\ [x = 1.232766201,\ y = -0.6630386021]$$

```
delete eqs:
```

Example 8

Usually, most of the time is spent internally searching for some (crude) approximations of the root. If high precision roots are required, it is recommended to compute first approximations with moderate values of `DIGITS` and use them as starting values for a refined search:

```
eq := exp(-x) = x:
DIGITS := 10:
firstApprox := numeric::fsolve(eq, x)
```

$$[x = 0.5671432904]$$

This output is suitable as input defining a starting value for x:

```
DIGITS := 1000: numeric::fsolve(eq, firstApprox)
```

$$[x = 0.5671432904097838729999686622103555497538\ldots]$$

[x =
0.5671432904097838729999686622103555497538157871865125081351310792230457930866845666

```
delete eq, firstApprox, DIGITS:
```

Example 9

Specifying starting values for the indeterminates launches a *single* Newton iteration. This may fail, if the starting values are not sufficiently close to the solution:

```
eq := [x*y = x + y - 4, x/y = x - y + 4]:
numeric::fsolve(eq, [x = 1, y = 1])
```

$$\text{FAIL}$$

If a search range is specified for at least one of the unknowns, then *several* Newton iterations with random starting values in the search range are used, until a solution is found or until `numeric::fsolve` gives up:

```
numeric::fsolve(eq, [x = 1, y = 0 .. 10])
```

$$\left[x = 4.02644145\ 10^{-14},\ y = 4.0\right]$$

```
delete eq:
```

Parameters

eq

An arithmetical expression or an equation in one indeterminate `x`. An expression `eq` is interpreted as the equation `eq = 0`.

eqs

A list, set, `array`, or `matrix` (`Cat::Matrix`) of expressions or equations in several indeterminates `x1, x2, ...`. Expressions are interpreted as homogeneous equations.

x, x_1, x_2, …

Identifiers or indexed identifiers to be solved for.

a, a_1, a_2, …

Real or complex numerical starting values for the internal search. Typically, crude approximations of solution.

a .. b, a_1 .. b_1, a_2 .. b_2, …

Ranges of numerical values defining search intervals for the numerical root.

Options

RestrictedSearch

Makes `numeric::fsolve` return only numerical roots in the user-defined search range `x = a .. b` and `[x_1 = a_1 .. b_1, x_2 = a_2 .. b_2, Symbol::hellip]`, respectively. This is the default search strategy, if a search range is specified for at least one of the unknowns.

Once a root with components (r_1, r_2, …) is found, it is checked whether $min(\Re(a_i), \Re(b_i)) \le \Re(r_i) \le max(\Re(a_i), \Re(b_i))$ and $min(\Im(a_i), \Im(b_i)) \le \Im(r_i) \le max(\Im(a_i), \Im(b_i))$ is satisfied. If the root is not inside the search range, the search is continued. Note that solutions outside the search range may be found internally. These may be accessed with the option `MultiSolutions`. See "Example 7" on page 19-118.

`UnrestrictedSearch`

Allows `numeric::fsolve` to find and return solutions outside the specified search range. With this option, the search range is only used to choose random starting points for the internal numerical search.

This option switches off the search strategy `RestrictedSearch`. With `UnrestrictedSearch`, `numeric::fsolve` stops its internal search whenever a root is found, even if the root is not inside the specified search range. Starting points for the internal Newton search are taken from the search range.

`MultiSolutions`

Makes `numeric::fsolve` return all solutions found in the internal search

This option only has an effect when used with the default search strategy `RestrictedSearch`. A sequence of all roots found in the internal search is returned. Cf. "Example 7" on page 19-118.

`Random`

With this option, several calls to `numeric::fsolve` with the same input parameters may produce different roots.

With this option, random starting values are chosen for the internal search. Consequently, calling `numeric::fsolve` several times with the same parameters may lead to different solutions. This may be useful when several roots of one and the same equation or set of equations are desired.

Return Values

Single numerical root is returned as a list of equations `[x = value]` or `[`x_1 `= value`$_1$`,` x_2 `= value`$_2$`, …]`, respectively. `FAIL` is returned if no solution is found. With the option `MultiSolutions`, sequences of solutions may be returned.

Algorithms

Internally the set of equations $f(x) = 0$ is solved by a modified Newton iteration $x \to x - t\, f'(x)^{-1}\, f(x)$ with some adaptively chosen step size t. For degenerate or ill-conditioned Jacobians f' a minimization strategy for $\langle f, f \rangle$ is implemented. For scalar real equations, `numeric::realroot` is used, if a real finite search range is specified.

See Also

MuPAD Functions

`linsolve` | `numeric::linsolve` | `numeric::polyroots` | `numeric::polysysroots` | `numeric::realroot` | `numeric::realroots` | `numeric::solve` | `polylib::realroots` | `solve`

More About

- "Solve Equations Numerically"

numeric::gaussAGM

Gauss' arithmetic geometric mean

Syntax

```
numeric::gaussAGM(a, b)
```

Description

`numeric::gaussAGM(a, b)` computes the arithmetic geometric mean of the numbers a and b.

The iteration

$$a_{n+1} = \frac{a_n + b_n}{2}, \quad b_{n+1} = (a_n + b_n)\sqrt{\frac{a_n b_n}{(a_n + b_n)^2}}$$

with the starting values $a_0 = a$, $b_0 = b$ converges quadratically to some value $\lim_{n\to\infty} a_n = \lim_{n\to\infty} b_n$. This limit is called Gauss' arithmetic geometric mean of the starting values a, b.

If both arguments a and b can be converted to real or complex floating-point numbers, then a floating point value is computed and returned. Otherwise, the symbolic call `numeric::gaussAGM(a, b)` is returned.

If $a = 0$ or $b = 0$ or $a + b = 0$, then 0.0 is returned, even if a or b are symbolic objects.

The following relation to elliptic integrals holds for all complex values a and b:

$$\text{numeric::gaussAGM}(a, b) = \frac{\pi}{4}\,\frac{a+b}{\mathrm{K}\left(\sqrt{1-\left(\frac{a-b}{a+b}\right)^2}\right)}.$$

Environment Interactions

The function is sensitive to the environment variable `DIGITS`.

Examples

Example 1

A floating-point number is returned if the arguments can be converted to floating-point numbers:

```
numeric::gaussAGM(0, 5)
```

$$0.0$$

```
numeric::gaussAGM(sqrt(2), PI)
```

$$2.192033978$$

```
numeric::gaussAGM(-10, PI)
```

$$-2.377943461 - 2.966350545\,i$$

```
numeric::gaussAGM(1 + I, 1 + 2*I)
```

$$1.020054126 + 1.471349363\,i$$

A symbolic call is returned if one of the arguments cannot be converted to a float:

```
numeric::gaussAGM(1, b)
```

$$\text{numeric::gaussAGM}(1, b)$$

For the special cases $a = 0$, $b = 0$ and $a + b = 0$, the result 0.0 is returned even for symbolic arguments:

```
numeric::gaussAGM(a, 0)
```

```
    0.0
```

```
numeric::gaussAGM(a, -a)
```

```
    0.0
```

Parameters

a, b

arithmetical expressions

Return Values

Floating point number or a symbolic call `numeric::gaussAGM(a, b)`.

See Also

MuPAD Functions
`ellipticK`

numeric::gldata

Weights and abscissae of Gauss-Legendre quadrature

Syntax

```
numeric::gldata(n, digits)
```

Description

`numeric::gldata(n, digits)` returns the weights and the abscissae of the Gauss-Legendre quadrature rule with n nodes with a precision of `digits` decimal digits.

The Gauss-Legendre quadrature rule $\sum_{i=1}^{n} b_i f(c_i)$ produces the exact integral $\int_0^1 f(x)\,\mathrm{d}x$ for all polynomial integrands $f(x)$ through degree $2n - 1$. The weights b_i and abscissae c_i are related to the roots of the n-th Legendre polynomial.

The weights and abscissae are computed by a straightforward numerical algorithm with a working precision set by the argument `digits`. The resulting floating-point numbers are correct to `digits` leading decimal places.

Typically, the argument `digits` is chosen as the current value of the environment variable `DIGITS`.

The data for `n = 20, 40, 80, 160` with `digits <= 200` are stored internally. They are returned immediately without any computational costs.

Due to the internal remember mechanism, only the first call to `numeric::gldata` leads to computational costs. For any further call with the same arguments, the data are returned immediately.

For odd n, the abscissa $c_{\frac{n+1}{2}} = \frac{1}{2}$ and the corresponding weight $b_{\frac{n+1}{2}}$ are rational numbers.

Environment Interactions

`numeric::gldata` is *not* sensitive to changes of the environment variable `DIGITS`, because the numerical working precision is specified by the second argument `digits`.

The function uses `option remember`.

Examples

Example 1

The following call computes the Gauss-Legendre data with a precision given by the current value of the environment variable `DIGITS` (the default value is `DIGITS = 10`):

```
[b, c] := numeric::gldata(4, DIGITS)
```

$$[[0.1739274226, 0.3260725774, 0.3260725774, 0.1739274226],\ [0.0694318442, 0.3300094782, 0.6699905218, 0.9305681558]]$$

The Gauss-Legendre data with 4 nodes provide exact numerical quadrature results for polynomials through degree 7:

```
f := x -> x^7:
int(f(x), x= 0..1) = _plus(b[i]*f(c[i]) $ i=1..4)
```

$$\frac{1}{8} = 0.125$$

```
delete b, c, f:
```

Example 2

For odd n, exact rational data for $c_{\frac{n+1}{2}}$ and $b_{\frac{n+1}{2}}$ are returned. The other data are computed as floating-point approximations:

```
DIGITS := 4: numeric::gldata(5, DIGITS)
```

$$\left[\left[0.1185,\ 0.2393,\ \frac{64}{225},\ 0.2393,\ 0.1185\right],\ \left[0.04691,\ 0.2308,\ \frac{1}{2},\ 0.7692,\ 0.9531\right]\right]$$

```
delete DIGITS:
```

Parameters

`n`

The number of nodes: a positive integer

`digits`

The number of decimal digits: a positive integer

Return Values

List `[b, c]` is returned. The lists `b = [b`$_1$`, …, b`$_n$`]` and `c = [c`$_1$`, …, c`$_n$`]` are numerical approximations of the weights and abscissae with `digits` significant digits.

Algorithms

The numerical integrator `numeric::quadrature` calls `numeric::gldata` to provide the data for Gaussian quadrature.

See Also

MuPAD Functions

`numeric::gtdata` | `numeric::int` | `numeric::ncdata` | `numeric::quadrature`

numeric::gtdata

Weights and abscissae of Gauss-Tschebyscheff quadrature

Syntax

```
numeric::gtdata(n)
```

Description

`numeric::gtdata(n)` returns the weights and the abscissae of the Gauss-Tschebyscheff quadrature rule with n nodes.

The Gauss-Tschebyscheff quadrature rule $\sum_{i=1}^{n} b_i f(c_i)$ produces the exact integral $\int_0^1 f(x)\,\mathrm{d}x$ for all integrands of the form $f(x) = \frac{p(x)}{\sqrt{x\,(1-x)}}$ with polynomials $p(x)$ through degree $2\,n$ - 1.

The exact weights $b = [b_1, \ldots, b_n]$ and abscissae $c = [c_1, \ldots, c_n]$ are given by

$$b_i = \frac{\pi}{2\,n}\,\sin\left(\frac{(2\,i-1)\,\pi}{2\,n}\right),\quad c_i = \frac{1+\cos\left(\frac{(2\,i-1)\,\pi}{2\,n}\right)}{2}.$$

Environment Interactions

`numeric::gtdata` is not sensitive to the environment variable `DIGITS`.

The function uses `option remember`.

Examples

Example 1

The following call produces exact data for the quadrature rule with two nodes:

```
numeric::gtdata(2)
```

$$\left[\left[\frac{\pi\sqrt{2}}{8}, \frac{\pi\sqrt{2}}{8}\right], \left[\frac{\sqrt{2}}{4}+\frac{1}{2}, \frac{1}{2}-\frac{\sqrt{2}}{4}\right]\right]$$

Parameters

n

The number of nodes: a positive integer

Return Values

List `[b,c]` is returned. The lists `b = [`b_1`, …, `b_n`]` and `c = [`c_1`, …, `c_n`]` are the exact weights and abscissae of the Gauss-Tschebyscheff quadrature rule, respectively.

Algorithms

The numerical integrator `numeric::quadrature` calls `numeric::gtdata` to provide the data for Gauss-Tschebyscheff quadrature.

See Also

MuPAD Functions

`numeric::gldata` | `numeric::int` | `numeric::ncdata` | `numeric::quadrature`

numeric::indets

Search for indeterminates

Syntax

```
numeric::indets(object)
```

Description

`numeric::indets(object)` returns a set of the indeterminates contained in the `object`.

This is an auxiliary routine used by `numeric::polyroots`, `numeric::quadrature`, `numeric::realroots`, `numeric::solve` etc. to find indeterminates.

It recursively searches the operands of `object` for indeterminates. In particular, the search is applied to the elements of lists, sets, arrays, tables, etc.

Following objects are regarded as indeterminates: identifiers, indexed identifiers and the indeterminates of `DOM_POLY` objects. Also coefficients of such polynomials are searched for indeterminates.

The following objects are *not* regarded as indeterminates: the numerical constants PI, EULER, and CATALAN (cf. `Type::ConstantIdents`) and zero operands of expressions and subexpressions (i.e., the function names in unevaluated function calls such as `f(2)`, `sin(PI/13)` etc.). Also integration variables in unevaluated calls of `int`, `numeric::int` and `numeric::quadrature` and summation indices in unevaluated calls of `sum` and `numeric::sum` are not considered.

Examples

Example 1

Identifiers and indexed identifiers are regarded as indeterminates:

```
numeric::indets([{a + b*PI}, sin(c + sqrt(2) + EULER),
                 table(1 = d - cos(e), 2 = f + 0.1*I),
                 array(1..2, [g, h]), F(i[2], i[2]),
                 D([1], G)(j[1]), k[3 + L[4]]])
```

$$\left\{a, b, c, d, e, f, g, h, i_2, j_1, k_{L_4+3}\right\}$$

Both indeterminates as well as symbolic coefficients are considered in polynomials of domain type `DOM_POLY`:

```
numeric::indets(poly(a[1]*x^2 + a[2]*x +a, [x, y]))
```

$$\{a, x, y, a_1, a_2\}$$

Example 2

The zero operands of unevaluated function calls such as `f(…)` or `sin(…)` are not regarded as indeterminates:

```
numeric::indets(f(a + sin(b) + PI + EULER))
```

$$\{a, b\}$$

Integration variables and summation indices are not regarded as indeterminates:

```
numeric::indets({int(f(x), x = a..b),
                 sum(f(i), i = c..infinity)})
```

$$\{a, b, c\}$$

Parameters

`object`

An arbitrary MuPAD object

Return Values

Set of indeterminates is returned, if the argument is an object of some basic data type of the kernel. The empty set is returned, if the object is from some library domain.

See Also

MuPAD Functions
`freeIndets` | `indets`

numeric::int

Numerical integration (the Float attribute of Int)

Syntax

```
numeric::int(f(x), x = a .. b, options)
```

```
float(holdint(f(x), x = a .. b, options))
```

```
float(freezeint(f(x), x = a .. b, options))
```

Description

`numeric::int(f(x), x = a..b)` computes a numerical approximation of $\int_a^b f(x)\,\mathrm{d}x$.

The calls `numeric::int(...)`, `float ( freeze(int)(...))`, and `float ( hold(int)(...))` are equivalent.

The calls `numeric::int(...)` and `numeric::quadrature(...)` are almost equivalent: `numeric::int` calls `numeric::quadrature`. A numerical result produced by `numeric::quadrature` is returned as is. Otherwise, `hold(numeric::int)(...)` is returned.

See the help page of `numeric::quadrature` for details.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We demonstrate some equivalent calls for numerical integration:

```
numeric::int(exp(x^2), x = -1..1),
float(hold(int)(exp(x^2), x = -1..1)),
float(freeze(int)(exp(x^2), x = -1..1)),
numeric::quadrature(exp(x^2), x = -1..1)
```

$$2.925303492,\ 2.925303492,\ 2.925303492,\ 2.925303492$$

```
numeric::int(max(1/10, cos(PI*x)), x = -2..0.0123),
float(hold(int)(max(1/10, cos(PI*x)), x = -2..0.0123)),
float(freeze(int)(max(1/10, cos(PI*x)), x = -2..0.0123)),
numeric::quadrature(max(1/10, cos(PI*x)), x = -2..0.0123)
```

$$0.752102471,\ 0.752102471,\ 0.752102471,\ 0.752102471$$

```
numeric::int(exp(-x^2), x = -2..infinity),
float(hold(int)(exp(-x^2), x = -2..infinity)),
float(freeze(int)(exp(-x^2), x = -2..infinity)),
numeric::quadrature(exp(-x^2), x = -2..infinity)
```

$$1.768308316,\ 1.768308316,\ 1.768308316,\ 1.768308316$$

```
numeric::int(sin(x)/x, x = -1..10, GaussLegendre = 5),
float(hold(int)(sin(x)/x, x = -1..10, GaussLegendre = 5)),
float(freeze(int)(sin(x)/x, x = -1..10, GaussLegendre = 5)),
numeric::quadrature(sin(x)/x, x = -1..10, GaussLegendre = 5)
```

$$2.604430665,\ 2.604430665,\ 2.604430665,\ 2.604430665$$

The calls `numeric::int(...)`, `float(hold(int)(...))`, and `numeric::quadrature(...)` are equivalent in multiple numerical integrations, too:

```
numeric::int(numeric::int(x*y, x = 0..y), y = 0..1),
numeric::int(numeric::quadrature(x*y, x = 0..y), y = 0..1),
float(freeze(int)(numeric::int(x*y, x = 0..y), y = 0..1)),
float(hold(int)(numeric::quadrature(x*y, x = 0..y), y = 0..1)),
numeric::quadrature(numeric::int(x*y, x = 0..y), y = 0..1),
numeric::quadrature(numeric::quadrature(x*y, x = 0..y), y = 0..1)
```

$$0.125,\ 0.125,\ 0.125,\ 0.125,\ 0.125,\ 0.125$$

Example 2

The following integral do not exist. Consequently, numerical integration runs into problems:

```
numeric::quadrature(1/x, x = 0..infinity)
```

```
Warning: Precision goal is not achieved after 10000 function calls. Increase 'MaxCalls
```

$$172.711431$$

Note that `numeric::int` handles errors produced by `numeric::quadrature` and returns a symbolic call to `numeric::int`:

```
numeric::int(1/x, x = 0..infinity)
```

$$\text{numeric::int}\left(\frac{1}{x}, x = 0..\infty\right)$$

Parameters

`f(x)`

An arithmetical expression in x

`x`

An identifier or an indexed identifier

`a, b`

arithmetical expressions

Options

All options of `numeric::quadrature` can be used.

Return Values

Floating point number or a symbolic call `numeric::int(f(x), x = a..b)` if the integral cannot be evaluated numerically.

See Also

MuPAD Functions

`int` | `numeric::quadrature`

More About

- "Integration"

numeric::inverse

Inverse of a matrix

Syntax

```
numeric::inverse(A, options)
```

Description

`numeric::inverse(A)` returns the inverse of the matrix `A`.

If no return type is specified via the option `ReturnType = t`, the domain type of the inverse depends on the type of the input matrix `A`:

- The inverse of an array is returned as an array.
- The inverse of an hfarray is returned as an hfarray.
- The inverse of a dense matrix of type `Dom::DenseMatrix()` is a dense matrix of type `Dom::DenseMatrix()` over the ring of MuPAD expressions.
- For all other matrices of category `Cat::Matrix`, the inverse is returned as a `matrix` of type `Dom::Matrix()` over the ring of MuPAD expressions. This includes input matrices `A` of type `Dom::Matrix(...)`, `Dom::SquareMatrix(...)`, `Dom::MatrixGroup(...)` etc.

The option `Symbolic` should be used if the matrix contains symbolic objects that cannot be converted to floating point numbers.

Without the option `Symbolic`, all entries of `A` must be numerical. Floating point arithmetic is used, the working precision is set by the environment variable `DIGITS`. Exact numerical expressions such as $\pi + \sqrt{2}$, `sin(3)` etc. are accepted and converted to floats. If symbolic entries are found in the matrix, `numeric::inverse` automatically switches to `Symbolic`, issuing a warning. This warning may be suppressed via the option `NoWarning`.

Note: Invertibility of the matrix can only be safely detected with exact arithmetic, i.e., using the option `Symbolic`. See “Example 2” on page 19-140.

Note: Matrices `A` of a matrix domain such as `Dom::Matrix(..)` or `Dom::SquareMatrix(..)` are internally converted to arrays over expressions via `expr(A)`. Note that `Symbolic` should be used if the entries cannot be converted to numerical expressions.

Note that `1/A` must be used, when the inverse is to be computed over the component domain. See "Example 3" on page 19-142.

We recommend to use `numeric::linsolve` or `numeric::matlinsolve` if a sparse system of linear equations is to be solved. In particular, these routines are more efficient than `numeric::inverse` for large sparse systems.

`numeric::linsolve` uses sparse input and output via symbolic equations and features internal sparse arithmetic.

Alternatively, sparse matrices of domain type `Dom::Matrix()` may be used with `numeric::matlinsolve`.

Environment Interactions

Without the option `Symbolic`, the function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

Numerical matrices can be processed with or without the option `Symbolic`. In the following, the inverses are returned as arrays because the input matrix is an array:

```
A := array(1..2, 1..2, [[1, 2], [3, PI]]):
```

```
numeric::inverse(A), numeric::inverse(A, Symbolic)
```

$$\begin{pmatrix} -1.099071012 & 0.6996903372 \\ 1.049535506 & -0.3498451686 \end{pmatrix}, \begin{pmatrix} \frac{\pi}{\pi-6} & -\frac{2}{\pi-6} \\ -\frac{3}{\pi-6} & \frac{1}{\pi-6} \end{pmatrix}$$

Matrices of category `Cat::Matrix` are accepted. The inverse is returned as a corresponding matrix:

```
A := Dom::Matrix()([[2, PI], [0, 1]]):
numeric::inverse(A); domtype(%)
```

$$\begin{pmatrix} 0.5 & -1.570796327 \\ 0 & 1.0 \end{pmatrix}$$

Dom::Matrix()

```
delete A:
```

Example 2

The following matrix is not invertible:

```
A := linalg::hilbert(6):
A[6,6] := 5773/63504:
A
```

$$\begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} \\ \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{5773}{63504} \end{pmatrix}$$

With exact arithmetic, `numeric::inverse` detects this fact:

```
numeric::det(A, Symbolic), numeric::inverse(A, Symbolic)
```

0, FAIL

Due to internal round-off, the matrix is regarded as invertible if float arithmetic is used:

```
numeric::det(A, HardwareFloats), numeric::inverse(A, HardwareFloats);
```

$$-3.239912837\,10^{-28},\ \left[\left[-1.339975706\,10^{11},\ 4.019927119\,10^{12},\ \sigma_3,\ 7.503863956\,10^{13},\ \sigma_5,\ 3.37673878\,10^{13}\right],\right.$$

$$\left[4.019927119\,10^{12},\ -1.205978136\,10^{14},\ 8.441846951\,10^{14},\ \sigma_4,\ 2.532554085\,10^{15},\ \sigma_6\right],$$

$$\left[\sigma_3,\ 8.441846951\,10^{14},\ -5.909292865\,10^{15},\ 1.575811431\,10^{16},\ \sigma_2,\ 7.091151439\,10^{15}\right],$$

$$\left[7.503863956\,10^{13},\ \sigma_4,\ 1.575811431\,10^{16},\ -4.202163815\,10^{16},\ 4.727434292\,10^{16},\ \sigma_1\right],$$

$$\left[\sigma_5,\ 2.532554085\,10^{15},\ \sigma_2,\ 4.727434292\,10^{16},\ -5.318363579\,10^{16},\ 2.127345432\,10^{16}\right],$$

$$\left.\left[3.37673878\,10^{13},\ \sigma_6,\ 7.091151439\,10^{15},\ \sigma_1,\ 2.127345432\,10^{16},\ -8.509381726\,10^{15}\right]\right]$$

where

$$\sigma_1 = -1.890973717\,10^{16}$$

$$\sigma_2 = -1.77278786\,10^{16}$$

$$\sigma_3 = -2.813948983\,10^{13}$$

$$\sigma_4 = -2.251159187\,10^{15}$$

$$\sigma_5 = -8.441846951\,10^{13}$$

$$\sigma_6 = -1.013021634\,10^{15}$$

With `SoftwareFloats`, the internal rounding is slightly different and the kernel of the matrix is detected:

```
numeric::det(A, SoftwareFloats), numeric::inverse(A, SoftwareFloats)
```

$$-3.731550906\,10^{-30},\ \text{FAIL}$$

```
delete A:
```

Example 3

The following matrix has domain components:

```
A := Dom::Matrix(Dom::IntegerMod(7))([[6, -1], [1, 6]])
```

$$\begin{pmatrix} 6 \bmod 7 & 6 \bmod 7 \\ 1 \bmod 7 & 6 \bmod 7 \end{pmatrix}$$

Note that `numeric::inverse` computes the inverse of the following matrix:

```
expr(A), numeric::inverse(A)
```

$$\begin{pmatrix} 6 & 6 \\ 1 & 6 \end{pmatrix}, \begin{pmatrix} 0.2 & -0.2 \\ -0.03333333333 & 0.2 \end{pmatrix}$$

The overloaded arithmetic should be used if the inverse is to be computed over the component domain `Dom::IntegerMod(7)`:

```
1/A
```

$$\begin{pmatrix} 3 \bmod 7 & 4 \bmod 7 \\ 3 \bmod 7 & 3 \bmod 7 \end{pmatrix}$$

```
delete A:
```

Example 4

The option `Symbolic` should not be used for float matrices because no internal pivoting is used to stabilize the numerical algorithm:

```
A := matrix([[1.0/10^20, 1.0], [1.0, 1.0]]):
bad = numeric::inverse(A, Symbolic),
good = numeric::inverse(A)
```

$$\text{bad} = \begin{pmatrix} 0 & 1.0 \\ 1.0 & -1.0\,10^{-20} \end{pmatrix},\ \text{good} = \begin{pmatrix} -1.0 & 1.0 \\ 1.0 & -1.0\,10^{-20} \end{pmatrix}$$

```
delete A:
```

Example 5

We demonstrate the use of hardware floats. Hilbert matrices are notoriously ill-conditioned and difficult to invert with low values of `DIGITS`. The following results, both with `HardwareFloats` as well as with `SoftwareFloats`, are marred by numerical round-off. Consequently, the inverses with and without hardware floats, respectively, differ significantly:

```
A := linalg::hilbert(10):
DIGITS := 10:
B1 := numeric::inverse(A, HardwareFloats):
B2 := numeric::inverse(A, SoftwareFloats):
B1[8, 8] <> B2[8, 8]
```

$$3.267405722\,10^{12} \neq 3.26785917\,10^{12}$$

```
norm(B1 - B2)
```

$$1681640193.0$$

```
delete A, B1, B2:
```

Parameters

A

A square matrix of domain type `DOM_ARRAY`, `DOM_HFARRAY`, or of category `Cat::Matrix`

Options

Hard, HardwareFloats, Soft, SoftwareFloats

With `Hard` (or `HardwareFloats`), computations are done using fast hardware float arithmetic from within a MuPAD session. `Hard` and `HardwareFloats` are equivalent.

With this option, the input data are converted to hardware floats and processed by compiled C code. The result is reconverted to MuPAD floats and returned to the MuPAD session.

With `Soft` (or `SoftwareFloats`) computations are dome using software float arithmetic provided by the MuPAD kernel. `Soft` and `SoftwareFloats` are equivalent. `SoftwareFloats` is used by default if the current value of `DIGITS` is larger than 15 and the input matrix `A` is not of domain type `DOM_HFARRAY`.

Compared to the `SoftwareFloats` used by the MuPAD kernel, the computation with `HardwareFloats` may be many times faster. Note, however, that the precision of hardware arithmetic is limited to about 15 digits. Further, the size of floating-point numbers may not be larger than approximately 10^{308} and not smaller than approximately 10^{-308}.

If no `HardwareFloats` or `SoftwareFloats` are requested explicitly, the following strategy is used: If the current value of `DIGITS` is smaller than 16 or if the matrix `A` is a hardware float array of domain type `DOM_HFARRAY`, then hardware arithmetic is tried. If this is successful, the result is returned.

If the result cannot be computed with hardware floats, software arithmetic by the MuPAD kernel is tried.

If the current value of `DIGITS` is larger than 15 and the input matrix `A` is not of domain type `DOM_HFARRAY`, or if one of the options `Soft`, `SoftwareFloats` or `Symbolic` is specified, MuPAD computes the result with its software arithmetic without trying to use hardware floats first.

There may be several reasons for hardware arithmetic to fail:

- The current value of `DIGITS` is larger than 15.
- The data contains symbolic objects.
- The data contains numbers larger than 10^{308} or smaller than 10^{-308} that cannot be represented by hardware floats.

If neither `HardwareFloats` nor `SoftwareFloats` is specified, the user is not informed whether hardware floats or software floats are used.

If `HardwareFloats` are specified but fail due to one of the reasons above, a warning is issued that the (much slower) software floating-point arithmetic of the MuPAD kernel is used.

Note that `HardwareFloats` can only be used if all input data can be converted to floating-point numbers.

The trailing digits in floating-point results computed with `HardwareFloats` and `SoftwareFloats` may differ.

Note: For ill-conditioned matrices, the result is subject to round-off errors. The results returned with `HardwareFloats` and `SoftwareFloats` may differ! See "Example 2" on page 19-140 and "Example 5" on page 19-143.

`Symbolic`

Prevents the conversion of the input data to floating-point numbers. Exact arithmetic is used. This option overrides `HardwareFloats` and `SoftwareFloats`.

This option prevents conversion of the input data to floats. With this option, symbolic entries are accepted.

Note: This option should not be used for floating-point matrices! No internal pivoting is used, unless necessary. Consequently, numerical instabilities may occur in floating-point operations. See "Example 4" on page 19-142.

`NoWarning`

Without the option `Symbolic`, `numeric::inverse` automatically switches to the `Symbolic` mode with a warning if symbolic coefficients are found. With the option `NoWarning`, this warning is suppressed. Note, however, that `numeric::inverse` still uses the symbolic mode for symbolic coefficients, i.e., exact arithmetic without floating-point conversions is used.

`ReturnType`

Option, specified as `ReturnType = t`

Return the inverse as a matrix of domain type `t`. The following return types are available: `DOM_ARRAY`, `DOM_HFARRAY`, `Dom::Matrix()`, or `Dom::DenseMatrix()`.

Return Values

Depending on the type of the input matrix A, the inverse is returned as a matrix of domain type DOM_ARRAY, DOM_HFARRAY, Dom::Matrix(), or Dom::DenseMatrix(). FAIL is returned if the inverse cannot be computed.

Algorithms

Gaussian elimination with partial pivoting is used. Partial pivoting is switched off by the option Symbolic.

See Also

MuPAD Functions

linalg::matlinsolve | linsolve | numeric::linsolve | numeric::matlinsolve | solve

numeric::leastSquares

Least squares solution of linear equations

Syntax

```
numeric::leastSquares(A, B, <mode>, <method>, options)
```

Description

`numeric::leastSquares(A, B)` computes a matrix X that solves the linear matrix equation $A\,X = B$ in the least squares sense: the columns X_j of `X` minimize $\left\| A\,X_j - B_j \right\|_2$ where the B_j are the columns of `B`.

For a given vector B, a vector X minimizes $\left\| A\,X - B \right\|_2$ if and only if X is a solution of the "normal equations" $A^H A\,X = A^H B$, where A^H is the Hermitian transpose of the $m \times n$ matrix A. The solution is unique if $rank(A) = n$.

`numeric::leastSquares` allows to solve several least squares problems simultaneously by combining several 'right hand sides' B_j columnwise to a matrix `B`.

If no return type is specified via the option `ReturnType = d`, the domain type of the return data depends on the type of the input matrix `A`:

- The special solution X as well as the kernel of an array A are returned as arrays.
- The special solution and the kernel of an hfarray of domain type `DOM_HFARRAY` are returned as hfarrays.
- For a dense matrix A of type `Dom::DenseMatrix()`, both the special solution X as well as the kernel are returned as matrices of type `Dom::DenseMatrix()` over the ring of MuPAD expressions.
- For all other matrices of category `Cat::Matrix`, both the special solution X as well as the kernel are returned as matrices of type `Dom::Matrix()` over the ring of MuPAD expressions. This includes input matrices `A` of type `Dom::Matrix(...)`, `Dom::SquareMatrix(...)`, `Dom::MatrixGroup(...)` etc.

Without `Symbolic`, the input data are converted to floating-point numbers. The matrix A must not contain non-convertible parameters, unless `Symbolic` is used. If such objects are found, `numeric::leastSquares` automatically switches to its symbolic mode, issuing a warning. This warning may be suppressed via `NoWarning`.

Symbolic parameters in B are accepted without warning. However, `HardwareFloats` cannot be used if there are any symbolic parameters in A or B.

If $A^H A$ has a non-trivial kernel, the least squares solution X is not unique. The return value `X` is a *special* solution of the equation $A^H A X = A^H B$. With the `SVD` method, `X` is the special solution with columns of minimal Euclidean length.

Note: The result computed with `HardwareFloats` may differ from the solution computed with `SoftwareFloats` or `Symbolic`! In particular, this is the case for systems with a non-trivial kernel. Further, the The results computed with `QRD` and `SVD` may differ.

The kernel is computed only in the symbolic mode (option `Symbolic`). All floating-point methods return the value `NIL` for the kernel.

With `Symbolic`, the $n \times d$ matrix `KernelBasis` is the most general solution of $A^H A X = 0$. Its columns span the d-dimensional kernel of $A^H A$.

If the kernel is 0-dimensional, the return value of `KernelBasis` is the integer 0. If `KernelBasis` is returned as an array, the dimension d of the kernel is `d = op(KernelBasis, [0, 3, 2])`. If `KernelBasis` is returned as a matrix of type `Dom::Matrix()` or `Dom::DenseMatrix()`, the dimension d of the kernel is `d = KernelBasis::dom::matdim(KernelBasis)[2]`.

Note: Without the option `Symbolic`, the implemented algorithms take care of numerical stabilization.

With `Symbolic`, exact data are assumed. The least squares solutions is computed via `numeric::matlinsolve( A^H A, A^H B , Symbolic)`. The symbolic strategy tries do maximize speed and does not take care of numerical stabilization! Do not use `Symbolic` for systems involving floating-point entries! In particluar, due to round-off, it may happen that no solution of $A^H A X = A^H B$ is found. In such a case, `[FAIL, NIL, NIL]` is returned. Cf. “Example 4” on page 19-152.

All entries of A and B must be arithmetical expressions.

Note: Apart from matrices of type `Dom::Matrix(...)`, `Cat::Matrix` objects A from matrix domains such as `Dom::DenseMatrix(...)` or `Dom::SquareMatrix(...)` are internally converted to arrays over expressions via `expr(A)`. Note that the option `Symbolic` should be used if the entries cannot be converted to numerical expressions.

The same holds true for matrices B passed as `Cat::Matrix` objects.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We consider a matrix A of rank 1:

```
A := array(1..3, 1..2, [[1, 2], [1, 2], [1, 2]]):
B := [3, 4, 5]:
```

The normal equations have a 1-parameter set of of solutions:

```
[X, KernelBasis, Res] := numeric::leastSquares(A, B, Symbolic)
```

$$\left[\begin{pmatrix} 4 \\ 0 \end{pmatrix}, \begin{pmatrix} -2 \\ 1 \end{pmatrix}, [\sqrt{2}]\right]$$

The numerical method `QRD` produces a special solution:

```
[X, KernelBasis, Res] := numeric::leastSquares(A, B, QRD)
```

$$\left[\begin{pmatrix} 4.0 \\ 0 \end{pmatrix}, \text{NIL}, [1.414213562]\right]$$

The numerical method SVD produces a solution X of minimal norm:

```
[X, KernelBasis, Res] := numeric::leastSquares(A, B, SVD)
```

$$\left[\begin{pmatrix} 0.8 \\ 1.6 \end{pmatrix}, \text{NIL}, [1.414213562]\right]$$

```
delete A, B, X, KernelBasis, Res:
```

Example 2

We consider an ill-conditioned least squares problem. By construction, the following overdetermined system has an exact solution $X = [1, 2, \ldots, n]$:

```
m := 10: n := 8:
A := array(1..m, 1..n, [[1/(i + j + 100) $ j=1..n] $ i=1..m]):
B := array(1..m, [_plus(A[i,j]*j $ j=1..n) $ i=1..m]):
numeric::leastSquares(A, B, Symbolic)
```

$$\left[\begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{pmatrix}, 0, [0]\right]$$

The coefficient matrix A is rather ill-conditioned:

```
singvals := numeric::singularvalues(A):
conditionOfA := max(op(singvals))/min(op(singvals))
```

$$6.31539107\ 10^{16}$$

Consequently, round-off has a drastic effect in a numerical approximation. The methods yield results of different quality:

```
numeric::leastSquares(A, B, QRD)
```

$$\left[\begin{pmatrix} 205.7634088 \\ -506.7540812 \\ 336.9881724 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \text{NIL}, [0.000000007174860922]\right]$$

```
numeric::leastSquares(A, B, SVD)
```

$$\left[\begin{pmatrix} 44440.34961 \\ -36432.38255 \\ -44826.69677 \\ -14839.48585 \\ 23895.8882 \\ 45730.9635 \\ 28571.67227 \\ -46506.25357 \end{pmatrix}, \text{NIL}, [0.0000007416898216]\right]$$

```
delete m, n, A, B, singvals, conditionOfA:
```

Example 3

This example involves a symbolic parameter `c` in the matrix *A*. The option `Symbolic` must be used:

```
A:= matrix([[c, 2], [1/3, 2/3], [1/7, 2/7]]):
B:= [1, 2, 3]:
numeric::leastSquares(A, B, Symbolic)
```

$$\left[\begin{pmatrix} -\frac{425}{58\,(c-1)} \\ \frac{483\,c-58}{116\,(c-1)} \end{pmatrix}, 0, \left[\text{sqrt}\left(\left|\frac{425\,c}{58\,(c-1)} - \frac{483\,c-58}{58\,(c-1)} + 1\right|^2 + \left|\frac{425}{174\,(c-1)} - \frac{483\,c-58}{174\,(c-1)} + 2\right|^2 + \left|\frac{425}{406\,(c-1)} - \frac{483\,c-58}{406\,(c-1)} + 3\right|^2\right)\right]\right]$$

```
normal(%)
```

$$\left[\left(\begin{array}{c} -\frac{425}{58\,(c-1)} \\ \\ \frac{483\,c-58}{116\,(c-1)} \end{array}\right),\ 0,\ \left[\frac{15\,\sqrt{58}}{58}\right]\right]$$

```
delete A, B:
```

Example 4

Floating point entries may cause problems in conjunction with the option `Symbolic`, because the computation is not stabilized numerically in the symbolic node. The following matrix A has rank 2:

```
A := matrix([[1, 30], [10.0^(-15), 31*10.0^(-15)]]):
```

However, due to round-off, the 'normal matrix' $A^H A$ has rank 1. No solution is found with `Symbolic`:

```
A::dom::transpose(A) * A
```

$$\left(\begin{array}{cc} 1.0 & 30.0 \\ 30.0 & 900.0 \end{array}\right)$$

```
numeric::leastSquares(A, [31, 32*10^5], Symbolic)
```

$$[\text{FAIL},\ \text{NIL},\ \text{NIL}]$$

No such problem arises in the numerical schemes. Note, however, that the numerical methods yield different results in this extremely ill-conditioned problem:

```
numeric::leastSquares(A, [31, 32*10^5], QRD)
```

$$\left[\left(\begin{array}{c} -9.6\,10^{22} \\ \\ 3.2\,10^{21} \end{array}\right),\ \text{NIL},\ [31.0]\right]$$

```
numeric::leastSquares(A, [31, 32*10^5], SVD)
```

$$\left[\begin{pmatrix} 0.03440621532 \\ 1.03218646 \end{pmatrix},\ \text{NIL},\ [3200000.0]\right]$$

```
delete A:
```

Parameters

A

An $m \times n$ matrix of domain type `DOM_ARRAY` or of category `Cat::Matrix`

B

An $m \times p$ matrix of domain type `DOM_ARRAY` or of category `Cat::Matrix`. Column vectors `B` may also be represented by a 1-dimensional `array(1..m, [B1, B2, …] )` or by a list `[B1, B2, …]`.

mode

One of the flags `Hard`, `HardwareFloats`, `Soft`, `SoftwareFloats`, or `Symbolic`

method

One of the flags `QRD`, `SVD`

Options

Hard, HardwareFloats, Soft, SoftwareFloats

With `Hard` (or `HardwareFloats`), computations are done using fast hardware float arithmetic from within a MuPAD session. `Hard` and `HardwareFloats` are equivalent. With this option, the input data are converted to hardware floats and processed by compiled C code. The result is reconverted to MuPAD floats and returned to the MuPAD session.

With `Soft` (or `SoftwareFloats`) computations are dome using software float arithmetic provided by the MuPAD kernel. `Soft` and `SoftwareFloats` are equivalent.

SoftwareFloats is used by default if the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY.

Compared to the SoftwareFloats used by the MuPAD kernel, the computation with HardwareFloats may be many times faster. Note, however, that the precision of hardware arithmetic is limited to about 15 digits. Further, the size of floating-point numbers may not be larger than approximately 10^{308} and not smaller than approximately 10^{-308}.

If no HardwareFloats or SoftwareFloats are requested explicitly, the following strategy is used: If the current value of DIGITS is smaller than 16 or if the matrix A is a hardware float array of domain type DOM_HFARRAY, then hardware arithmetic is tried. If this is successful, the result is returned.

If the result cannot be computed with hardware floats, software arithmetic by the MuPAD kernel is tried.

If the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY, or if one of the options Soft, SoftwareFloats or Symbolic is specified, MuPAD computes the result with its software arithmetic without trying to use hardware floats first.

There may be several reasons for hardware arithmetic to fail:

- The current value of DIGITS is larger than 15.
- The data contains symbolic objects.
- The data contains numbers larger than 10^{308} or smaller than 10^{-308} that cannot be represented by hardware floats.

If neither HardwareFloats nor SoftwareFloats is specified, the user is not informed whether hardware floats or software floats are used.

If HardwareFloats are specified but fail due to one of the reasons above, a warning is issued that the (much slower) software floating-point arithmetic of the MuPAD kernel is used.

Note that HardwareFloats can only be used if all input data can be converted to floating-point numbers.

The trailing digits in floating-point results computed with HardwareFloats and SoftwareFloats may differ.

Symbolic

Prevents the conversion of the input data to floating-point numbers. Exact arithmetic is used. This option overrides `HardwareFloats` and `SoftwareFloats`.

This option *must* be used, if the matrix A contains symbolic parameters that cannot be converted to floating-point numbers.

The normal equations $A^H A X = A^H B$ are passed to `numeric::matlinsolve` with the option `Symbolic`.

If the least squares problem does not have a unique solution, a special solution X is returned together with the kernel of $A^H A$. Cf. "Example 1" on page 19-149.

Note: This option should not be used for systems with floating-point coefficients! Numerical instabilities may occur in floating-point operations. Further, if the rank of A is not maximal, then `numeric::leastSquares` may fail to find a solution due to numerical round-off. In such a case, `[FAIL, NIL, NIL]` is returned. Cf. "Example 4" on page 19-152.

QRD

Use a QR decomposition. All entries of A must be convertible to floating-point values.

This is the default method.

The matrix A must not contain symbolic parameters that cannot be converted to floating point numbers. If such objects are found, then `numeric::leastSquares` automatically switches to its `Symbolic` mode, issuing a warning. The computation proceeds with exact arithmetic, using the input data without floating-point conversions.

The warning may be suppressed by the option `NoWarning`.

Symbolic parameters in B are accepted without warning. They are processed by the floating-point algorithm.

Numerical expressions such as e^{π}, $\sqrt{2}$ etc. are accepted and converted to floats.

If the least squares problem does not have a unique solution, only a special solution is returned. The kernel is not computed: it is returned as `NIL`.

The method `QRD` provides a numerically stable way of solving the normal equations $A^H A X = A^H B$ by a QR decomposition. In extremely ill-conditioned situations, it may be worthwhile to consider the slower, yet more stable method `SVD`.

The conditioning is given by the ratio of the largest singular value of A divided by the smallest singular value of A. If this value is large, the problem is ill-conditioned.

Cf. “Example 2” on page 19-150.

`SVD`

Use a singular value decomposition. All entries of A must be convertible to floating-point values.

The matrix A must not contain symbolic parameters that cannot be converted to floating point numbers. If such objects are found, then `numeric::leastSquares` automatically switches to its symbolic mode, issuing a warning. The computation proceeds with exact arithmetic, using the input data without floating point conversions.

The warning may be suppressed by the option `NoWarning`.

Symbolic parameters in B are accepted without warning. They are processed by the floating-point algorithm.

Numerical expressions such as e^{π}, $\sqrt{2}$ etc. are accepted and converted to floats.

If the least squares problem does not have a unique solution, the columns X_j of the solution X have a minimal Euclidean length $\|X_j\|_2$.

The kernel is not computed: it is returned as `NIL`.

A singular value decomposition $A = U D V^H$ is used to solve the normal equations in the form $D^2 V^H X = D U^H B$. For small or zero singular values d_j in $D = diag(d_1, d_2, \ldots)$, the corresponding components of $V^H x$ are set to zero.

Usually, the numerical method `SVD` is slower than the default method `QRD`. However, in ill-conditioned situations, it is numerically more stable.

The conditioning is given by the ratio of the largest singular value of A divided by the smallest singular value of A. If this value is large, the problem is ill-conditioned.

NoWarning

Suppresses warnings

If symbolic coefficients are found in A, `numeric::leastSquares` automatically switches to the `Symbolic` mode with a warning. With this option, this warning is suppressed.

ReturnType

Option, specified as `ReturnType = d`

Return the (special) solution and the kernel as matrices of domain type `d`. The following return types d are available: `DOM_ARRAY`, or `DOM_HFARRAY`, `Dom::Matrix()`, or `Dom::DenseMatrix()`.

Return Values

A list `[X, KernelBasis, Residues]` is returned.

The (special) least squares solution `X` is an $n \times p$ matrix.

With `Symbolic`, `KernelBasis` is an $n \times d$ matrix (d is the dimension of the kernel of $A^H A$). Its columns span the kernel of $A^H A$. If the kernel is trivial, `KernelBasis` is the integer 0.

Without `Symbolic`, the kernel is not computed. The value `NIL` is returned for the `KernelBasis`.

The list of arithmetical expressions `Residues` consists of the minimized least squares deviations $\left\| A\,X_j - B_j \right\|_2$ corresponding to the columns of `X` and `B`.

See Also

MuPAD Functions

`numeric::factorQR` | `numeric::linsolve` | `numeric::matlinsolve` | `numeric::singularvalues` | `numeric::singularvectors`

numeric::linsolve

Solve a system of linear equations

Syntax

```
numeric::linsolve(eqs, <vars>, options)
```

Description

`numeric::linsolve(eqs, vars)` solves a system of linear equations `eqs` for the unknowns `vars`.

`numeric::linsolve` is a fast numerical linear solver. It is also a recommended solver for linear systems with exact or symbolic coefficients (using `Symbolic`).

Expressions are interpreted as homogeneous equations. E.g., the input `[x = y - 1, x - y]` is interpreted as the system of equations `[x = y - 1, x - y = 0]`.

Note: Without the option `Symbolic`, the input data are converted to floating-point numbers. The coefficient matrix A of the system $A\,x = b$ represented by `eqs` must not contain non-convertible parameters, unless the option `Symbolic` is used! If such objects are found, then `numeric::linsolve` automatically switches to its symbolic mode, issuing a warning. This warning may be suppressed via `NoWarning`. Symbolic parameters in the "right hand side" b are accepted without warning.

The numerical working precision is set by the environment variable `DIGITS`.

The solutions are returned as a list of solved equations of the form

$$[x_1 = \text{value}_1, x_2 = \text{value}_2, \ldots]\,,$$

where x_1, x_2, … are the unknowns. These simplified equations should be regarded as constraints on the unknowns. E.g., if an unknown x_1, say, does not turn up in the form [x_1

= …, …] in the solution, then there is no constraint on this unknown; it is an arbitrary parameter. Generally, all unknowns that do not turn up on the left hand side of the solved equations are arbitrary parameters spanning the solution space. Cf. “Example 9” on page 19-170.

In particular, if the empty list is returned as the solution, there are no constraints whatsoever on the unknowns, i.e., the system is trivial.

The ordering of the solved equations corresponds to the ordering of the unknowns `vars`. It is recommended that the user specifies `vars` by a *a list* of unknowns. This guarantees that the solved equations are returned in the expected order. If `vars` are specified by a set, or if no `vars` are specified at all, then an internal ordering is used.

If no unknowns are specified by `vars`, `numeric::linsolve` solves for all symbolic objects in `eqs`. The unknowns are determined internally by `indets(eqs, PolyExpr)`.

`numeric::linsolve` returns the general solution of the system `eqs`. It is valid for arbitrary complex values of the symbolic parameters which may be present in `eqs`. If no such solution exists, `FAIL` is returned. Solutions that are valid only for special values of the symbolic parameters may be obtained with the option `ShowAssumptions`. See “Example 2” on page 19-161, “Example 3” on page 19-161, “Example 4” on page 19-162, and “Example 11” on page 19-172.

The solved equations representing the solution are suitable as input for `assign` and `subs`. See “Example 8” on page 19-169.

`numeric::linsolve` is suitable for solving large sparse systems. See “Example 6” on page 19-163.

If `eqs` represents a system with a banded coefficient matrix, then this is detected and used by `numeric::linsolve`. Note that in this case, it is important to specify both the equations as well as the unknowns by lists to guarantee the desired form of the coefficient matrix. When using sets, the data may be reordered internally leading to a loss of band structure and, consequently, of efficiency. See “Example 6” on page 19-163.

Note: `numeric::linsolve` is tuned for speed. For this reason, it does not check systematically that the equations `eqs` are indeed linear in the unknowns! For non-linear equations, strange things may happen; `numeric::linsolve` might even return wrong results! See “Example 5” on page 19-162.

Note: `numeric::linsolve` does not react to any properties of the unknowns or of symbolic parameters that are set via `assume`.

Note: Gaussian elimination with partial pivoting is used. Without the option `Symbolic`, floating-point arithmetic is used and the pivoting strategy takes care of numerical stabilization. With `Symbolic`, exact data are assumed and the pivoting strategy tries do maximize speed, not taking care of numerical stabilization! See "Example 7" on page 19-169.

Environment Interactions

Without the option `Symbolic`, the function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

Equations and variables may be entered as sets or lists:

```
numeric::linsolve({x = y - 1, x + y = z}, {x, y});
numeric::linsolve([x = y - 1, x + y = z], {x, y});
numeric::linsolve({x = y - 1, x + y = z}, [x, y]);
numeric::linsolve([x = y - 1, x + y = z], [x, y])
```

$$[x = 0.5\,z - 0.5,\ y = 0.5\,z + 0.5]$$

$$[x = 0.5\,z - 0.5,\ y = 0.5\,z + 0.5]$$

$$[x = 0.5\,z - 0.5,\ y = 0.5\,z + 0.5]$$

$$[x = 0.5\,z - 0.5,\ y = 0.5\,z + 0.5]$$

With the option `Symbolic`, exact arithmetic is used. The following system has a 1-parameter set of solution; the unknown x_3 is arbitrary:

```
numeric::linsolve([x[1] + x[2] = 2, x[1] - x[2] = 2*x[3]],
```

```
                    [x[1], x[2], x[3]], Symbolic)
```

$$[x_1 = x_3 + 1,\ x_2 = 1 - x_3]$$

The unknowns may be expressions:

```
numeric::linsolve([f(0) - sin(x + 1) = 2, f(0) = 1 - sin(x + 1)],
                  [f(0), sin(x + 1)])
```

$$[f(0) = 1.5,\ \sin(x + 1) = -0.5]$$

The following system does not have a solution:

```
numeric::linsolve([x + y = 1, x + y = 2], [x, y])
```

FAIL

Example 2

We demonstrate some examples with symbolic coefficients. Note that the option `Symbolic` has to be used:

```
eqs := [x + a*y = b, x + A*y = b]:
numeric::linsolve(eqs, [x, y], Symbolic)
```

$$[x = b,\ y = 0]$$

Note that for $a = A$, this is not the general solution. Using the option `ShowAssumptions`, it turns out that the above result is the general solution subject to the assumption $a \neq A$:

```
numeric::linsolve(eqs, [x, y], Symbolic, ShowAssumptions)
```

$$[[x = b,\ y = 0],\ [],\ [A - a \neq 0]]$$

```
delete eqs:
```

Example 3

We give a further demonstration of the option `ShowAssumptions`. The following system does not have a solution for all values of the parameter `a`:

```
numeric::linsolve([x + y = 1, x + y = a], [x, y], Symbolic)
```

```
FAIL
```

With `ShowAssumptions`, `numeric::linsolve` investigates under which conditions (on the parameter `a`) there is a solution:

```
numeric::linsolve([x + y = 1, x + y = a], [x, y], Symbolic,
                  ShowAssumptions)
```

$$[[x = 1 - y], [a - 1 = 0], []]$$

We conclude that there is a 1-parameter set of solutions for $a = 1$. The constraint in `a` is a linear equation, since the parameter `a` enters the equations linearly. If `a` is regarded as an unknown rather than as a parameter, the constraint becomes part of the solution:

```
numeric::linsolve([x + y = 1, x + y = a], [x, y, a], Symbolic,
                  ShowAssumptions)
```

$$[[x = 1 - y, a = 1], [], []]$$

Example 4

With exact arithmetic, PI is regarded as a symbolic parameter. The following system has a solution subject to the constraint `PI = 1`:

```
numeric::linsolve([x = x - y + 1, y = PI], [x, y],
                  Symbolic, ShowAssumptions)
```

$$[[y = \pi], [1 - \pi = 0], []]$$

With floating-point arithmetic, PI is converted to `3.1415...`. The system has no solution:

```
numeric::linsolve([x = x - y + 1, y = PI], [x, y],
                  ShowAssumptions)
```

```
[FAIL, [], []]
```

Example 5

Since `numeric::linsolve` does not do a systematic internal check for non-linearities, the user should make sure that the equations to be solved are indeed linear in the

unknowns. Otherwise, strange things may happen. Garbage is produced for the following non-linear systems:

```
a := sin(x):
numeric::linsolve([y = 1 - a, x = y], [x, y], Symbolic)
```

$$[x = 1 - \sin(x17^3),\ y = 1 - \sin(x17^3)]$$

```
numeric::linsolve([a*x + y = 1, x = y], [x, y], Symbolic)
```

$$\left[x = \frac{1}{\sin(x18^3) + 1},\ y = \frac{1}{\sin(x18^3) + 1}\right]$$

Polynomial non-linearities are usually detected. Regarding `x, y, c` as unknowns, the following quadratic system yields an error:

```
numeric::linsolve([x*c + y = 1, x = y], Symbolic)
```

```
Error: This system does not seem to be linear. [numeric::linsolve]
```

```
 Error:
This system does not seem to be linear. [numeric::linsolve]
```

This system is linear in `x, y` if `c` is regarded as a parameter:

```
numeric::linsolve([x*c + y = 1, x = y], [x, y], Symbolic)
```

$$\left[x = \frac{1}{c + 1},\ y = \frac{1}{c + 1}\right]$$

```
delete a:
```

Example 6

We solve a large sparse system. The coefficient matrix has only 3 diagonal bands. Note that both the equations as well as the variables are passed as lists. This guarantees that the band structure is not lost internally:

```
n := 500: x[0] := 0: x[n + 1] := 0:
eqs := [x[i-1] - 2*x[i] + x[i+1] = 1 $ i = 1..n]:
vars := [x[i] $ i = 1..n]:
numeric::linsolve(eqs, vars)
```

$$[x_1 = -250.0,\ x_2 = -499.0,\ x_3 = -747.0,\ x_4 = -994.0,\ \ldots,\ x_{498} = -747.0,\ x_{499} = -499.0,\ x_{500} = -250.0]$$

[x[1] = -250.0, x[2] = -499.0, x[3] = -747.0, x[4] = -994.0, x[5] = -1240.0, x[6] = -1485.0, x[7] = -1729.0, x[8] = -1972.0, x[9] = -2214.0, x[10] = -2455.0, x[11] = -2695.0, x[12] = -2934.0, x[13] = -3172.0, x[14] = -3409.0, x[15] = -3645.0, x[16] = -3880.0, x[17] = -4114.0, x[18] = -4347.0, x[19] = -4579.0, x[20] = -4810.0, x[21] = -5040.0, x[22] = -5269.0, x[23] = -5497.0, x[24] = -5724.0, x[25] = -5950.0, x[26] = -6175.0, x[27] = -6399.0, x[28] = -6622.0, x[29] = -6844.0, x[30] = -7065.0, x[31] = -7285.0, x[32] = -7504.0, x[33] = -7722.0, x[34] = -7939.0, x[35] = -8155.0, x[36] = -8370.0, x[37] = -8584.0, x[38] = -8797.0, x[39] = -9009.0, x[40] = -9220.0, x[41] = -9430.0, x[42] = -9639.0, x[43] = -9847.0, x[44] = -10054.0, x[45] = -10260.0, x[46] = -10465.0, x[47] = -10669.0, x[48] = -10872.0, x[49] = -11074.0, x[50] = -11275.0, x[51] = -11475.0, x[52] = -11674.0, x[53] = -11872.0, x[54] = -12069.0, x[55] = -12265.0, x[56] = -12460.0, x[57] = -12654.0, x[58] = -12847.0, x[59] = -13039.0, x[60] = -13230.0, x[61] = -13420.0, x[62] = -13609.0, x[63] = -13797.0, x[64] = -13984.0, x[65] = -14170.0, x[66] = -14355.0, x[67] = -14539.0, x[68] = -14722.0, x[69] = -14904.0, x[70] = -15085.0, x[71] = -15265.0, x[72] = -15444.0, x[73] = -15622.0, x[74] = -15799.0, x[75] = -15975.0, x[76] = -16150.0, x[77] = -16324.0, x[78] = -16497.0, x[79] = -16669.0, x[80] = -16840.0, x[81] = -17010.0, x[82] = -17179.0, x[83] = -17347.0, x[84] = -17514.0, x[85] = -17680.0, x[86] = -17845.0, x[87] = -18009.0, x[88] = -18172.0, x[89] = -18334.0, x[90] = -18495.0, x[91] = -18655.0, x[92] = -18814.0, x[93] = -18972.0, x[94] = -19129.0, x[95] = -19285.0, x[96] = -19440.0, x[97] = -19594.0, x[98] = -19747.0, x[99] = -19899.0, x[100] = -20050.0, x[101] = -20200.0, x[102] = -20349.0, x[103] = -20497.0, x[104] = -20644.0, x[105] = -20790.0, x[106] = -20935.0, x[107] = -21079.0, x[108] = -21222.0, x[109] = -21364.0, x[110] = -21505.0, x[111] = -21645.0, x[112] = -21784.0, x[113] = -21922.0, x[114] = -22059.0, x[115] = -22195.0, x[116] = -22330.0, x[117] = -22464.0, x[118] = -22597.0, x[119] = -22729.0, x[120] = -22860.0, x[121] = -22990.0, x[122] = -23119.0, x[123] = -23247.0, x[124] = -23374.0, x[125] = -23500.0, x[126] = -23625.0, x[127] = -23749.0, x[128] = -23872.0, x[129] = -23994.0, x[130] = -24115.0, x[131] = -24235.0, x[132] = -24354.0, x[133] = -24472.0, x[134] = -24589.0, x[135] = -24705.0, x[136] = -24820.0, x[137] = -24934.0, x[138] = -25047.0, x[139] = -25159.0, x[140] = -25270.0, x[141] = -25380.0, x[142] = -25489.0, x[143] = -25597.0, x[144] = -25704.0, x[145] = -25810.0, x[146] = -25915.0, x[147] = -26019.0, x[148] = -26122.0, x[149] = -26224.0, x[150] = -26325.0, x[151] = -26425.0, x[152] = -26524.0, x[153] = -26622.0, x[154] = -26719.0, x[155] = -26815.0, x[156] = -26910.0, x[157] = -27004.0, x[158] = -27097.0, x[159] = -27189.0, x[160] = -27280.0, x[161] = -27370.0, x[162] = -27459.0, x[163] = -27547.0, x[164] = -27634.0, x[165] = -27720.0, x[166] = -27805.0, x[167] = -27889.0, x[168] = -27972.0, x[169] = -28054.0, x[170] = -28135.0, x[171] = -28215.0, x[172] =

-28294.0, x[173] = -28372.0, x[174] = -28449.0, x[175] = -28525.0, x[176] = -28600.0,
x[177] = -28674.0, x[178] = -28747.0, x[179] = -28819.0, x[180] = -28890.0, x[181] =
-28960.0, x[182] = -29029.0, x[183] = -29097.0, x[184] = -29164.0, x[185] = -29230.0,
x[186] = -29295.0, x[187] = -29359.0, x[188] = -29422.0, x[189] = -29484.0, x[190] =
-29545.0, x[191] = -29605.0, x[192] = -29664.0, x[193] = -29722.0, x[194] = -29779.0,
x[195] = -29835.0, x[196] = -29890.0, x[197] = -29944.0, x[198] = -29997.0, x[199] =
-30049.0, x[200] = -30100.0, x[201] = -30150.0, x[202] = -30199.0, x[203] = -30247.0,
x[204] = -30294.0, x[205] = -30340.0, x[206] = -30385.0, x[207] = -30429.0, x[208] =
-30472.0, x[209] = -30514.0, x[210] = -30555.0, x[211] = -30595.0, x[212] = -30634.0,
x[213] = -30672.0, x[214] = -30709.0, x[215] = -30745.0, x[216] = -30780.0, x[217] =
-30814.0, x[218] = -30847.0, x[219] = -30879.0, x[220] = -30910.0, x[221] = -30940.0,
x[222] = -30969.0, x[223] = -30997.0, x[224] = -31024.0, x[225] = -31050.0, x[226] =
-31075.0, x[227] = -31099.0, x[228] = -31122.0, x[229] = -31144.0, x[230] = -31165.0,
x[231] = -31185.0, x[232] = -31204.0, x[233] = -31222.0, x[234] = -31239.0, x[235] =
-31255.0, x[236] = -31270.0, x[237] = -31284.0, x[238] = -31297.0, x[239] = -31309.0,
x[240] = -31320.0, x[241] = -31330.0, x[242] = -31339.0, x[243] = -31347.0, x[244] =
-31354.0, x[245] = -31360.0, x[246] = -31365.0, x[247] = -31369.0, x[248] = -31372.0,
x[249] = -31374.0, x[250] = -31375.0, x[251] = -31375.0, x[252] = -31374.0, x[253] =
-31372.0, x[254] = -31369.0, x[255] = -31365.0, x[256] = -31360.0, x[257] = -31354.0,
x[258] = -31347.0, x[259] = -31339.0, x[260] = -31330.0, x[261] = -31320.0, x[262] =
-31309.0, x[263] = -31297.0, x[264] = -31284.0, x[265] = -31270.0, x[266] = -31255.0,
x[267] = -31239.0, x[268] = -31222.0, x[269] = -31204.0, x[270] = -31185.0, x[271] =
-31165.0, x[272] = -31144.0, x[273] = -31122.0, x[274] = -31099.0, x[275] = -31075.0,
x[276] = -31050.0, x[277] = -31024.0, x[278] = -30997.0, x[279] = -30969.0, x[280] =
-30940.0, x[281] = -30910.0, x[282] = -30879.0, x[283] = -30847.0, x[284] = -30814.0,
x[285] = -30780.0, x[286] = -30745.0, x[287] = -30709.0, x[288] = -30672.0, x[289] =
-30634.0, x[290] = -30595.0, x[291] = -30555.0, x[292] = -30514.0, x[293] = -30472.0,
x[294] = -30429.0, x[295] = -30385.0, x[296] = -30340.0, x[297] = -30294.0, x[298] =
-30247.0, x[299] = -30199.0, x[300] = -30150.0, x[301] = -30100.0, x[302] = -30049.0,
x[303] = -29997.0, x[304] = -29944.0, x[305] = -29890.0, x[306] = -29835.0, x[307] =
-29779.0, x[308] = -29722.0, x[309] = -29664.0, x[310] = -29605.0, x[311] = -29545.0,
x[312] = -29484.0, x[313] = -29422.0, x[314] = -29359.0, x[315] = -29295.0, x[316] =
-29230.0, x[317] = -29164.0, x[318] = -29097.0, x[319] = -29029.0, x[320] = -28960.0,
x[321] = -28890.0, x[322] = -28819.0, x[323] = -28747.0, x[324] = -28674.0, x[325] =
-28600.0, x[326] = -28525.0, x[327] = -28449.0, x[328] = -28372.0, x[329] = -28294.0,
x[330] = -28215.0, x[331] = -28135.0, x[332] = -28054.0, x[333] = -27972.0, x[334] =
-27889.0, x[335] = -27805.0, x[336] = -27720.0, x[337] = -27634.0, x[338] = -27547.0,
x[339] = -27459.0, x[340] = -27370.0, x[341] = -27280.0, x[342] = -27189.0, x[343] =
-27097.0, x[344] = -27004.0, x[345] = -26910.0, x[346] = -26815.0, x[347] = -26719.0,
x[348] = -26622.0, x[349] = -26524.0, x[350] = -26425.0, x[351] = -26325.0, x[352] =

-26224.0, x[353] = -26122.0, x[354] = -26019.0, x[355] = -25915.0, x[356] = -25810.0,
x[357] = -25704.0, x[358] = -25597.0, x[359] = -25489.0, x[360] = -25380.0, x[361] =
-25270.0, x[362] = -25159.0, x[363] = -25047.0, x[364] = -24934.0, x[365] = -24820.0,
x[366] = -24705.0, x[367] = -24589.0, x[368] = -24472.0, x[369] = -24354.0, x[370] =
-24235.0, x[371] = -24115.0, x[372] = -23994.0, x[373] = -23872.0, x[374] = -23749.0,
x[375] = -23625.0, x[376] = -23500.0, x[377] = -23374.0, x[378] = -23247.0, x[379] =
-23119.0, x[380] = -22990.0, x[381] = -22860.0, x[382] = -22729.0, x[383] = -22597.0,
x[384] = -22464.0, x[385] = -22330.0, x[386] = -22195.0, x[387] = -22059.0, x[388] =
-21922.0, x[389] = -21784.0, x[390] = -21645.0, x[391] = -21505.0, x[392] = -21364.0,
x[393] = -21222.0, x[394] = -21079.0, x[395] = -20935.0, x[396] = -20790.0, x[397] =
-20644.0, x[398] = -20497.0, x[399] = -20349.0, x[400] = -20200.0, x[401] = -20050.0,
x[402] = -19899.0, x[403] = -19747.0, x[404] = -19594.0, x[405] = -19440.0, x[406] =
-19285.0, x[407] = -19129.0, x[408] = -18972.0, x[409] = -18814.0, x[410] = -18655.0,
x[411] = -18495.0, x[412] = -18334.0, x[413] = -18172.0, x[414] = -18009.0, x[415] =
-17845.0, x[416] = -17680.0, x[417] = -17514.0, x[418] = -17347.0, x[419] = -17179.0,
x[420] = -17010.0, x[421] = -16840.0, x[422] = -16669.0, x[423] = -16497.0, x[424] =
-16324.0, x[425] = -16150.0, x[426] = -15975.0, x[427] = -15799.0, x[428] = -15622.0,
x[429] = -15444.0, x[430] = -15265.0, x[431] = -15085.0, x[432] = -14904.0, x[433] =
-14722.0, x[434] = -14539.0, x[435] = -14355.0, x[436] = -14170.0, x[437] = -13984.0,
x[438] = -13797.0, x[439] = -13609.0, x[440] = -13420.0, x[441] = -13230.0, x[442] =
-13039.0, x[443] = -12847.0, x[444] = -12654.0, x[445] = -12460.0, x[446] = -12265.0,
x[447] = -12069.0, x[448] = -11872.0, x[449] = -11674.0, x[450] = -11475.0, x[451] =
-11275.0, x[452] = -11074.0, x[453] = -10872.0, x[454] = -10669.0, x[455] = -10465.0,
x[456] = -10260.0, x[457] = -10054.0, x[458] = -9847.0, x[459] = -9639.0, x[460] = -9430.0,
x[461] = -9220.0, x[462] = -9009.0, x[463] = -8797.0, x[464] = -8584.0, x[465] = -8370.0,
x[466] = -8155.0, x[467] = -7939.0, x[468] = -7722.0, x[469] = -7504.0, x[470] = -7285.0,
x[471] = -7065.0, x[472] = -6844.0, x[473] = -6622.0, x[474] = -6399.0, x[475] = -6175.0,
x[476] = -5950.0, x[477] = -5724.0, x[478] = -5497.0, x[479] = -5269.0, x[480] = -5040.0,
x[481] = -4810.0, x[482] = -4579.0, x[483] = -4347.0, x[484] = -4114.0, x[485] = -3880.0,
x[486] = -3645.0, x[487] = -3409.0, x[488] = -3172.0, x[489] = -2934.0, x[490] = -2695.0,
x[491] = -2455.0, x[492] = -2214.0, x[493] = -1972.0, x[494] = -1729.0, x[495] = -1485.0,
x[496] = -1240.0, x[497] = -994.0, x[498] = -747.0, x[499] = -499.0, x[500] = -250.0]

The band structure is lost if the equations or the unknowns are specified by sets. The following call takes more time than the previous call:

```
numeric::linsolve({op(eqs)}, {x[i] $ i = 1..n})
```

$$[x_1 = -250.0,\ x_2 = -499.0,\ x_3 = -747.0,\ x_4 = -994.0,\ \ldots,\ x_{498} = -747.0,\ x_{499} = -499.0,\ x_{500} = -250.0]$$

$$[x_1 = -250.0, x_2 = -499.0, x_3 = -747.0, x_4 = -994.0, x_5 = -1240.0, x_6 = -1485.0,$$
$$x_7 = -1729.0, x_8 = -1972.0, x_9 = -2214.0, x_{10} = -2455.0, x_{11} = -2695.0, x_{12} = -2934.0,$$
$$x_{13} = -3172.0, x_{14} = -3409.0, x_{15} = -3645.0, x_{16} = -3880.0, x_{17} = -4114.0, x_{18} = -4347.0,$$
$$x_{19} = -4579.0, x_{20} = -4810.0, x_{21} = -5040.0, x_{22} = -5269.0, x_{23} = -5497.0, x_{24} = -5724.0,$$
$$x_{25} = -5950.0, x_{26} = -6175.0, x_{27} = -6399.0, x_{28} = -6622.0, x_{29} = -6844.0, x_{30} = -7065.0,$$
$$x_{31} = -7285.0, x_{32} = -7504.0, x_{33} = -7722.0, x_{34} = -7939.0, x_{35} = -8155.0, x_{36} = -8370.0,$$
$$x_{37} = -8584.0, x_{38} = -8797.0, x_{39} = -9009.0, x_{40} = -9220.0, x_{41} = -9430.0, x_{42} = -9639.0,$$
$$x_{43} = -9847.0, x_{44} = -10054.0, x_{45} = -10260.0, x_{46} = -10465.0, x_{47} = -10669.0,$$
$$x_{48} = -10872.0, x_{49} = -11074.0, x_{50} = -11275.0, x_{51} = -11475.0, x_{52} = -11674.0,$$
$$x_{53} = -11872.0, x_{54} = -12069.0, x_{55} = -12265.0, x_{56} = -12460.0, x_{57} = -12654.0,$$
$$x_{58} = -12847.0, x_{59} = -13039.0, x_{60} = -13230.0, x_{61} = -13420.0, x_{62} = -13609.0,$$
$$x_{63} = -13797.0, x_{64} = -13984.0, x_{65} = -14170.0, x_{66} = -14355.0, x_{67} = -14539.0,$$
$$x_{68} = -14722.0, x_{69} = -14904.0, x_{70} = -15085.0, x_{71} = -15265.0, x_{72} = -15444.0,$$
$$x_{73} = -15622.0, x_{74} = -15799.0, x_{75} = -15975.0, x_{76} = -16150.0, x_{77} = -16324.0,$$
$$x_{78} = -16497.0, x_{79} = -16669.0, x_{80} = -16840.0, x_{81} = -17010.0, x_{82} = -17179.0,$$
$$x_{83} = -17347.0, x_{84} = -17514.0, x_{85} = -17680.0, x_{86} = -17845.0, x_{87} = -18009.0,$$
$$x_{88} = -18172.0, x_{89} = -18334.0, x_{90} = -18495.0, x_{91} = -18655.0, x_{92} = -18814.0,$$
$$x_{93} = -18972.0, x_{94} = -19129.0, x_{95} = -19285.0, x_{96} = -19440.0, x_{97} = -19594.0,$$
$$x_{98} = -19747.0, x_{99} = -19899.0, x_{100} = -20050.0, x_{101} = -20200.0, x_{102} = -20349.0,$$
$$x_{103} = -20497.0, x_{104} = -20644.0, x_{105} = -20790.0, x_{106} = -20935.0, x_{107} = -21079.0,$$
$$x_{108} = -21222.0, x_{109} = -21364.0, x_{110} = -21505.0, x_{111} = -21645.0, x_{112} = -21784.0,$$

```
delete n, x, eqs, vars:
```

Example 7

The option `Symbolic` should not be used for equations with floating-point coefficients, because the symbolic pivoting strategy favors efficiency instead of numerical stability.

```
eqs := [x + 10^20*y = 10^20, x + y = 0]:
```

The float approximation of the exact solution is:

```
map(numeric::linsolve(eqs, [x, y], Symbolic), map, float)
```

$$[x = -1.0,\ y = 1.0]$$

We now convert the exact coefficients to floating-point numbers:

```
feqs := map(eqs, map, float)
```

$$[x + 1.0\ 10^{20}\ y = 1.0\ 10^{20},\ x + y = 0.0]$$

The default pivoting strategy stabilizes floating-point operations. Consequently, one gets a correct result:

```
numeric::linsolve(feqs, [x, y])
```

$$[x = -1.0,\ y = 1.0]$$

With `Symbolic`, the pivoting strategy optimizes speed, assuming exact arithmetic. Numerical instabilities may occur if floating-point coefficients are involved. The following incorrect result is caused by internal round-off effects (“cancellation”):

```
numeric::linsolve(feqs, [x, y], Symbolic)
```

$$[x = 0.0,\ y = 1.0]$$

```
delete eqs, feqs:
```

Example 8

We demonstrate that the simplified equations representing the solution can be used for further processing with `subs`:

```
eqs := [x + y = 1, x + y = a]:
[Solution, Constraints, Pivots] :=
  numeric::linsolve(eqs, [x, y], ShowAssumptions)
```

$$[[x = 1.0 - 1.0\,y], [1.0\,a - 1.0 = 0], []]$$

```
subs(eqs, Solution)
```

$$[1.0 = 1, 1.0 = a]$$

The solution can be assigned to the unknowns via `assign`:

```
assign(Solution):
x, y, eqs
```

$$1.0 - 1.0\,y,\ y,\ [1.0 = 1, 1.0 = a]$$

```
delete eqs, Solution, Constraints, Pivots, x:
```

Example 9

If the solution of the linear system is not unique, then some of the unknowns are used as "free parameters" spanning the solution space. In the following example, the unknowns `z, w` are such parameters. They do not turn up on the left hand side of the solved equations:

```
eqs := [x + y = z, x + 2*y = 0, 2*x - z = -3*y, y + z = 0]:
vars := [x, y, z, w]:
Solution := numeric::linsolve(eqs, vars, Symbolic)
```

$$[x = 2\,z,\ y = -z]$$

You may define a function such as the following `NewSolutionList` to rename your free parameters to "myName1", "myName2" etc. and fill up your list of solved equations accordingly:

```
NewSolutionList :=
proc(Solution : DOM_LIST, vars : DOM_LIST, myName : DOM_STRING)
local i, solvedVars, newEquation;
begin
  solvedVars := map(Solution, op, 1);
```

```
  for i from 1 to nops(vars) do
     if not has(solvedVars, vars[i]) then
        newEquation := vars[i] = genident(myName);
        Solution := listlib::insertAt(
            subs(Solution, newEquation), newEquation, i)
     end_if
  end_for:
  Solution
end_proc:

NewSolutionList(Solution, vars, "FreeParameter")
```

$$[x = 2\,\text{FreeParameter1},\ y = -\text{FreeParameter1},\ z = \text{FreeParameter1},\ w = \text{FreeParameter2}]$$

```
delete eqs, vars, Solution, NewSolutionList:
```

Example 10

We demonstrate the difference between hardware and software arithmetic. The following problem is very ill-conditioned. The results, both with `HardwareFloats` as well as with `SoftwareFloats`, are marred by numerical round-off:

```
n:= 10:
eqs:= [(_plus(x[j]/(i + j -1) $ j = 1..n) = 1) $ i = 1..n]:
vars:= [x[i] $ i = 1..n]:
numeric::linsolve(eqs, vars, SoftwareFloats);
numeric::linsolve(eqs, vars, HardwareFloats)
```

$$[x_1 = -9.999986342,\ x_2 = 989.9988244,\ x_3 = -23759.97502,\ x_4 = 240239.7733,$$
$$x_5 = -1261258.92,\ x_6 = 3783777.035,\ x_7 = -6726715.139,\ x_8 = 7001275.306,$$
$$x_9 = -3938217.537,\ x_{10} = 923779.4586]$$

$$[x_1 = -9.997364824,\ x_2 = 989.7718609,\ x_3 = -23755.13378,\ x_4 = 240195.7143,$$
$$x_5 = -1261048.597,\ x_6 = 3783198.501,\ x_7 = -6725765.49,\ x_8 = 7000357.238,$$
$$x_9 = -3937735.418,\ x_{10} = 923673.4085]$$

This is the exact solution:

```
numeric::linsolve(eqs, vars, Symbolic);
```

$$[x_1 = -10,\ x_2 = 990,\ x_3 = -23760,\ x_4 = 240240,\ x_5 = -1261260,\ x_6 = 3783780,\ x_7 = -6726720,$$
$$x_8 = 7001280,\ x_9 = -3938220,\ x_{10} = 923780]$$

```
delete eqs, vars:
```

Example 11

We demonstrate how a complete solution of the following linear system in `x`, `y` with symbolic parameters `a`, `b`, `c`, `d` may be found:

```
eqs := [x + y = d, a*x + b*y = 1, x + c*y = 1]:
numeric::linsolve(eqs, [x, y], Symbolic, ShowAssumptions)
```

$$\left[\left[x = -\frac{b\,d-1}{a-b},\ y = \frac{a\,d-1}{a-b}\right],\ [a-b+c+b\,d-a\,c\,d-1=0],\ [b-a\neq 0]\right]$$

This is the general solution, assuming $a \neq b$. We now set $b = a$ to investigate further solution branches:

```
eqs := subs(eqs, b = a):
numeric::linsolve(eqs, [x, y], Symbolic, ShowAssumptions)
```

$$\left[\left[x = \frac{c\,d-1}{c-1},\ y = -\frac{d-1}{c-1}\right],\ [1-a\,d=0],\ [c-1\neq 0]\right]$$

This is the general solution for $a = b$, assuming $c \neq 1$. We finally set $c = 1$ to obtain the last solution branch:

```
eqs := subs(eqs, c = 1):
numeric::linsolve(eqs, [x, y], Symbolic, ShowAssumptions)
```

$$[[x = d-y],\ [1-a\,d=0,\ 1-d=0],\ []]$$

From the constraints on the symbolic parameters `a` and `d`, we conclude that there is a special 1-parameter solution $x = 1 - y$ for $a = b = c = d = 1$.

```
delete eqs:
```

Parameters

eqs

A list, set, `array`, or `matrix` (`Cat::Matrix`) of linear equations or arithmetical expressions

vars

A list or set of unknowns to solve for. Unknowns may be identifiers or indexed identifiers or arithmetical expressions.

Options

Hard, HardwareFloats, Soft, SoftwareFloats

With `Hard` (or `HardwareFloats`), computations are done using fast hardware float arithmetic from within a MuPAD session. `Hard` and `HardwareFloats` are equivalent. With this option, the input data are converted to hardware floats and processed by compiled C code. The result is reconverted to MuPAD floats and returned to the MuPAD session.

With `Soft` (or `SoftwareFloats`) computations are dome using software float arithmetic provided by the MuPAD kernel. `Soft` and `SoftwareFloats` are equivalent. `SoftwareFloats` is used by default if the current value of `DIGITS` is larger than 15 and the input matrix `A` is not of domain type `DOM_HFARRAY`.

Compared to the `SoftwareFloats` used by the MuPAD kernel, the computation with `HardwareFloats` may be many times faster. Note, however, that the precision of hardware arithmetic is limited to about 15 digits. Further, the size of floating-point numbers may not be larger than approximately 10^{308} and not smaller than approximately 10^{-308}.

If no `HardwareFloats` or `SoftwareFloats` are requested explicitly, the following strategy is used: If the current value of `DIGITS` is smaller than 16 or if the matrix `A` is a hardware float array of domain type `DOM_HFARRAY`, then hardware arithmetic is tried. If this is successful, the result is returned.

If the result cannot be computed with hardware floats, software arithmetic by the MuPAD kernel is tried.

If the current value of `DIGITS` is larger than 15 and the input matrix `A` is not of domain type `DOM_HFARRAY`, or if one of the options `Soft`, `SoftwareFloats` or `Symbolic` is specified, MuPAD computes the result with its software arithmetic without trying to use hardware floats first.

There may be several reasons for hardware arithmetic to fail:

- The current value of `DIGITS` is larger than 15.
- The data contains symbolic objects.
- The data contains numbers larger than 10^{308} or smaller than 10^{-308} that cannot be represented by hardware floats.

If neither `HardwareFloats` nor `SoftwareFloats` is specified, the user is not informed whether hardware floats or software floats are used.

If `HardwareFloats` are specified but fail due to one of the reasons above, a warning is issued that the (much slower) software floating-point arithmetic of the MuPAD kernel is used.

Note that `HardwareFloats` can only be used if all input data can be converted to floating-point numbers.

The trailing digits in floating-point results computed with `HardwareFloats` and `SoftwareFloats` may differ.

Note: For ill-conditioned systems, the result is subject to round-off errors. The results returned with `HardwareFloats` and `SoftwareFloats` may differ significantly! See“Example 10” on page 19-171.

`Symbolic`

Prevents conversion of input data to floating-point numbers. This option overrides `HardwareFloats` and `SoftwareFloats`.

This option *must* be used if the coefficients of the equations contain symbolic parameters that cannot be converted to floating-point numbers.

Note: This option should not be used for equations with floating-point coefficients! Numerical instabilities may occur in floating-point operations. See “Example 7” on page 19-169.

`ShowAssumptions`

Returns information on internal assumptions on symbolic parameters in `eqs`.

This option is only useful if the equations contain symbolic parameters. Consequently, it should only be used in conjunction with the option `Symbolic`.

Note: The format of the return value is changed to `[Solution, Constraints, Pivots]`.

`Solution` is a set of simplified equations representing the general solution subject to `Constraints` and `Pivots`.

`Constraints` is a list of equations for symbolic parameters in `eqs`, which are necessary and sufficient to make the system solvable.

Such constraints arise if Gaussian elimination of the original equations leads to equations of the form $0 = c$, where c is some expression involving symbolic parameters in the “right hand side” of the system. All such equations are collected in `Constraints`. `numeric::linsolve` assumes that these equations are satisfied and returns a solution.

If no such constraints arise, the return value of `Constraints` is the empty list.

`Pivots` is a list of inequalities involving symbolic parameters in the coefficient matrix A of the linear system $A\,x = b$ represented by `eqs`. Internally, division by pivot elements occurs in the Gaussian elimination. The expressions collected in `Pivots` are the numerators of the pivot elements that contain symbolic parameters. If only numerical pivot elements were used, the return value of `Pivots` is the empty list.

Note: The option `ShowAssumptions` changes the return strategy for “unsolvable” systems. Without the option `Symbolic`, `FAIL` is returned whenever Gaussian elimination produces an equation $0 = c$ with non-zero c. With `ShowAssumptions`, such equations are returned via `Constraints`, provided c involves symbolic parameters.

If *c* is a purely numerical value, then `[FAIL, [], []]` is returned.

See "Example 2" on page 19-161, "Example 3" on page 19-161, "Example 4" on page 19-162, and "Example 11" on page 19-172.

`NoWarning`

Suppresses warnings

If symbolic coefficients are found, `numeric::linsolve` automatically switches to the `Symbolic` mode with a warning. With this option, this warning is suppressed; `numeric::linsolve` still uses the symbolic mode for symbolic coefficients, i.e., exact arithmetic without floating-point conversions is used.

Return Values

Without the option `ShowAssumptions`, a list of simplified equations is returned. It represents the general solution of the system `eqs`. `FAIL` is returned if the system is not solvable.

With `ShowAssumptions`, a list `[Solution, Constraints, Pivots]` is returned. `Solution` is a list of simplified equations representing the general solution of `eqs`. The lists `Constraints` and `Pivots` contain equations and inequalities involving symbolic parameters in `eqs`. Internally, these were assumed to hold true when solving the system.

`[FAIL, [], []]` is returned if the system is not solvable.

See Also

MuPAD Functions

`isolate` | `linalg::matlinsolve` | `linsolve` | `numeric::fsolve` | `numeric::inverse` | `numeric::matlinsolve` | `numeric::polyroots` | `numeric::polysysroots` | `numeric::realroots` | `polylib::realroots` | `solve`

numeric::matlinsolve

Solve a linear matrix equation

Syntax

```
numeric::matlinsolve(A, B, options)
```

Description

`numeric::matlinsolve(A, B)` returns the matrix solution X of the matrix equation $A\,X = B$ together with the kernel of the matrix A.

`numeric::matlinsolve` is a fast numerical linear solver for both sparse and dense systems. It is also a recommended solver for linear systems with exact or symbolic coefficients (use option `Symbolic`).

If no return type is specified via the option `ReturnType = d`, the domain type of the return data depends on the type of the input matrix `A`:

- The special solution X as well as the kernel of an array A are returned as arrays.
- The special solution X as well as the kernel of an hfarray A are returned as hfarrays.
- For a dense matrix$A$ of type `Dom::DenseMatrix()`, both the special solution X as well as the kernel of A are returned as matrices of type `Dom::DenseMatrix()` over the ring of MuPAD expressions.
- For all other matrices of category `Cat::Matrix`, both the special solution X as well as the kernel of A are returned as matrices of type `Dom::Matrix()` over the ring of MuPAD expressions. This includes input matrices `A` of type `Dom::Matrix(...)`, `Dom::SquareMatrix(...)`, `Dom::MatrixGroup(...)` etc.

Without `Symbolic`, exact numerical input data such as `PI + sqrt(2)`, `sin(3)` etc. are converted to floating-point numbers. Floating point arithmetic is used. Its precision is given the environment variable `DIGITS`. If symbolic data are found that cannot be converted to floating-point numbers, `numeric::matlinsolve` automatically switches to its symbolic mode, issuing a warning. This warning may be suppressed via `NoWarning`.

With `Symbolic`, symbolic parameters in the coefficient matrix `A` as well as in the right hand side `B` are accepted and processed without a warning.

With SofwareFloats, the right hand side B may contain symbolic parameters that cannot be converted to floating-point numbers. All entries of the coefficient matrix A, however, must be convertible to floating-point numbers.

With HardwareFloats, neither A nor B must contain symbolic parameters that cannot be converted to floating-point numbers.

X is a *special* solution of the equation $A\,X = B$. If A has a non-trivial kernel, the solution X is not unique.

Note: The result computed with HardwareFloats may differ from the solution computed with SoftwareFloats or Symbolic! In particular, this is the case for systems with a non-trivial kernel.

Cf. "Example 9" on page 19-186.

The $n \times d$ matrix KernelBasis is the most general solution of $A\,X = 0$. Its columns span the d-dimensional kernel of A.

Thus, the kernel of A may be computed via `numeric::matlinsolve(A, [0, ..., 0])[2]`.

If the kernel is 0-dimensional, the return value of KernelBasis is the integer 0. If KernelBasis is returned as an array, the dimension d of the kernel is `d = op(KernelBasis, [0, 3, 2])`. If KernelBasis is returned as a matrix of type Dom::Matrix() or Dom::DenseMatrix(), the dimension d of the kernel is `d = KernelBasis::dom::matdim(KernelBasis)[2]`.

Note: Due to round-off errors, some or all basis vectors in the kernel of A may be missed in the numerical modes.

The special solution X in conjunction with KernelBasis provides the general solution of $A\,X = B$. Solutions generated without the option ShowAssumptions are valid for arbitrary complex values of the symbolic parameters which may be present in A and B. If no such solution exists, then [FAIL,NIL] is returned. Solutions that are valid only for special values of the symbolic parameters may be obtained with ShowAssumptions. See "Example 3" on page 19-181, "Example 4" on page 19-182, and "Example 7" on page 19-184.

`numeric::matlinsolve` internally uses a sparse representation of the matrices. It is suitable for solving large sparse systems. See “Example 5” on page 19-182.

Note: `numeric::matlinsolve` does not react to any assumptions on symbolic parameters in `A,B` that are set via `assume`.

Note: Gaussian elimination with partial pivoting is used. Without the option `Symbolic`, the pivoting strategy takes care of numerical stabilization. With `Symbolic`, exact data are assumed. The symbolic pivoting strategy tries do maximize speed and does not take care of numerical stabilization! Do not use `Symbolic` for linear systems involving floating-point entries! See “Example 6” on page 19-183.

Note: Apart from matrices of type `Dom::Matrix(...)`, `Cat::Matrix` objects `A` from matrix domains such as `Dom::DenseMatrix(...)` or `Dom::SquareMatrix(...)` are internally converted to arrays over expressions via `expr(A)`. Note that the option `Symbolic` should be used if the entries cannot be converted to numerical expressions.

Note that `linalg::matlinsolve` must be used, when the solution is to be computed over the component domain. See . “Example 8” on page 19-185.

Environment Interactions

Without the option `Symbolic`, the function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We use equivalent input formats `(B1, B2)` to represent a vector with components [a, π]. First, this vector is defined as a 2-dimensional array:

```
A := array(1..2, 1..3, [[1, 2, 3],[1, 1, 2]]):
B1 := array(1..2, 1..1, [[a], [PI]]):
```

```
numeric::matlinsolve(A, B1)
```

$$\left[\begin{pmatrix} 6.283185307 - 1.0\,a \\ 1.0\,a - 3.141592654 \\ 0 \end{pmatrix}, \begin{pmatrix} -1.0 \\ -1.0 \\ 1.0 \end{pmatrix}\right]$$

Next, we use a 1-dimensional array and compute an exact solution:

```
B2 := array(1..2, [a, PI]):
numeric::matlinsolve(A, B2, Symbolic)
```

$$\left[\begin{pmatrix} 2\,\pi - a \\ a - \pi \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ -1 \\ 1 \end{pmatrix}\right]$$

Now, a list is used to specify the vector. No internal assumptions were used by `numeric::matlinsolve` to obtain the solution:

```
B3 := [a, PI]:
numeric::matlinsolve(A, B3, ShowAssumptions)
```

$$\left[\begin{pmatrix} 6.283185307 - 1.0\,a \\ 1.0\,a - 3.141592654 \\ 0 \end{pmatrix}, \begin{pmatrix} -1.0 \\ -1.0 \\ 1.0 \end{pmatrix}, [\,], [\,]\right]$$

Finally, we use `Dom::Matrix` objects to specify the system. Note that the results are returned as corresponding matrix objects:

```
A := matrix([[1, 2, 3],[1, 1, 2]]):
B4 := matrix([a, PI]):
numeric::matlinsolve(A, B4)
```

$$\left[\begin{pmatrix} 6.283185307 - 1.0\,a \\ 1.0\,a - 3.141592654 \\ 0 \end{pmatrix}, \begin{pmatrix} -1.0 \\ -1.0 \\ 1.0 \end{pmatrix}\right]$$

```
delete A, B1, B2, B3, B4:
```

Example 2

We invert a matrix by solving $A\,X = 1$:

```
A := hfarray(1..3, 1..3, [[1, 1, 0], [0, 1, 1], [0, 0, 1]]):
B := matrix::identity(3, 3):
InverseOfA := numeric::matlinsolve(A, B, Symbolic)[1]
```

$$\begin{pmatrix} 1.0 & -1.0 & 1.0 \\ 0.0 & 1.0 & -1.0 \\ 0.0 & 0.0 & 1.0 \end{pmatrix}$$

```
delete A, B, InverseOfA:
```

Example 3

We solve an equation with a symbolic parameter x:

```
A := matrix([[2, 2, 3], [1, 1, 2], [3, 3, 5]]):
B := matrix([sin(x)^2, cos(x)^2, 0]):
[X, Kernel, Constraints, Pivots] :=
 numeric::matlinsolve(A, B, Symbolic, ShowAssumptions)
```

$$\left[\begin{pmatrix} 5\sin(x)^2 \\ 0 \\ -3\sin(x)^2 \end{pmatrix}, \begin{pmatrix} -1 \\ 1 \\ 0 \end{pmatrix}, [\cos(x)^2 + \sin(x)^2 = 0], []\right]$$

This solution holds subject to the constraint $sin(x)^2 + cos(x)^2 = 0$ on the parameter x. `numeric::matlinsolve` does not investigate the `Constraints` and does not realize that they cannot be satisfied. We check the consistency of the "result" by inserting the solution into the original system. Since the input matrix A was of type `Dom::Matrix()`, the results `X` and `Kernel` were returned as corresponding matrices. The overloaded operators `*` and `-` for matrix multiplication and subtraction can be used:

```
A*X - B, A*Kernel
```

$$\left(\begin{matrix} 0 \\ -\cos(x)^2 - \sin(x)^2 \\ 0 \end{matrix}\right), \left(\begin{matrix} 0 \\ 0 \\ 0 \end{matrix}\right)$$

```
delete A, B, X, Kernel, Constraints, Pivots:
```

Example 4

We give a further demonstration of the option `ShowAssumptions`. The following system does not have a solution for all values of the parameter `a`:

```
A := array(1..2, 1..2, [[1, 1], [1, 1]]):
B := array(1..2, 1..1, [[1], [a]]):
numeric::matlinsolve(A, B, Symbolic)
```

$$[\text{FAIL}, \text{NIL}]$$

With `ShowAssumptions`, `numeric::matlinsolve` investigates under which conditions (on the parameter a `a`) there is a solution:

```
numeric::matlinsolve(A, B, Symbolic, ShowAssumptions)
```

$$\left[\left(\begin{matrix} 1 \\ 0 \end{matrix}\right), \left(\begin{matrix} -1 \\ 1 \end{matrix}\right), [a - 1 = 0], []\right]$$

We conclude that there is a 1-dimensional solution space for $a = 1$.

```
delete A, B:
```

Example 5

We solve a sparse system with 3 diagonal bands:

```
n := 100:
A := matrix(n, n, [1, -2, 1], Banded):
B := array(1..n, [1 $ n]):
numeric::matlinsolve(A, B)
```

$$\left[\begin{pmatrix} -50.0 \\ -99.0 \\ -147.0 \\ \cdots \\ -147.0 \\ -99.0 \\ -50.0 \end{pmatrix}, 0\right]$$

[matrix([[-50.0], [-99.0], [-147.0], [-194.0], [-240.0], [-285.0], [-329.0], [-372.0], [-414.0], [-455.0], [-495.0], [-534.0], [-572.0], [-609.0], [-645.0], [-680.0], [-714.0], [-747.0], [-779.0], [-810.0], [-840.0], [-869.0], [-897.0], [-924.0], [-950.0], [-975.0], [-999.0], [-1022.0], [-1044.0], [-1065.0], [-1085.0], [-1104.0], [-1122.0], [-1139.0], [-1155.0], [-1170.0], [-1184.0], [-1197.0], [-1209.0], [-1220.0], [-1230.0], [-1239.0], [-1247.0], [-1254.0], [-1260.0], [-1265.0], [-1269.0], [-1272.0], [-1274.0], [-1275.0], [-1275.0], [-1274.0], [-1272.0], [-1269.0], [-1265.0], [-1260.0], [-1254.0], [-1247.0], [-1239.0], [-1230.0], [-1220.0], [-1209.0], [-1197.0], [-1184.0], [-1170.0], [-1155.0], [-1139.0], [-1122.0], [-1104.0], [-1085.0], [-1065.0], [-1044.0], [-1022.0], [-999.0], [-975.0], [-950.0], [-924.0], [-897.0], [-869.0], [-840.0], [-810.0], [-779.0], [-747.0], [-714.0], [-680.0], [-645.0], [-609.0], [-572.0], [-534.0], [-495.0], [-455.0], [-414.0], [-372.0], [-329.0], [-285.0], [-240.0], [-194.0], [-147.0], [-99.0], [-50.0]]), 0]

```
delete n, A, B:
```

Example 6

The option `Symbolic` should not be used for equations with floating-point coefficients, because the symbolic pivoting strategy favors efficiency instead of numerical stability.

```
A := array(1..2, 1..2, [[1, 10^20], [1, 1]]):
B := array(1..2, 1..1, [[10^20], [0]]):
```

The float approximation of the exact solution is:

```
map(numeric::matlinsolve(A, B, Symbolic)[1], float)
```

$$\begin{pmatrix} -1.0 \\ 1.0 \end{pmatrix}$$

We now convert the exact input data to floating-point approximations:

```
A := map(A, float): B := map(B, float):
```

The default pivoting strategy of the floating-point algorithm stabilizes floating-point operations. Consequently, one gets a correct result:

```
numeric::matlinsolve(A, B)[1]
```

$$\begin{pmatrix} -1.0 \\ 1.0 \end{pmatrix}$$

With the option `Symbolic`, however, the pivoting strategy optimizes speed, assuming exact arithmetic. Numerical instabilities may occur if floating-point coefficients are involved. The following result is caused by internal round-off effects (“cancellation”):

```
numeric::matlinsolve(A, B, Symbolic)[1]
```

$$\begin{pmatrix} 0 \\ 1.0 \end{pmatrix}$$

We need to increase `DIGITS` to obtain a better result:

```
DIGITS := 20:
numeric::matlinsolve(A, B, Symbolic)[1]
```

$$\begin{pmatrix} -1.0000000149011611938 \\ 1.0 \end{pmatrix}$$

```
delete A, B, DIGITS:
```

Example 7

We demonstrate how a complete solution of the following linear system with symbolic parameters may be found:

```
A := array(1..3, 1..2, [[1, 1], [a, b], [1, c]]):
B := array(1..3, 1..1, [[1], [1], [1]]):
numeric::matlinsolve(A, B, Symbolic, ShowAssumptions)
```

$$\left[\begin{pmatrix} -\frac{b-1}{a-b} \\ \\ \frac{a-1}{a-b} \end{pmatrix}, 0, [-(a-1)(c-1)=0], [b-a \neq 0]\right]$$

This is the general solution assuming $a \neq b$. We now set $b = a$ to investigate further solution branches:

```
A := subs(A, b = a):
numeric::matlinsolve(A, B, Symbolic, ShowAssumptions)
```

$$\left[\begin{pmatrix}1\\0\end{pmatrix}, 0, [1-a=0], [c-1\neq 0]\right]$$

This is the general solution for $a = b$, assuming $c \neq 1$. We finally set $c = 1$ to obtain the last solution branch:

```
A := subs(A, c = 1):
numeric::matlinsolve(A, B, Symbolic, ShowAssumptions)
```

$$\left[\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}-1\\1\end{pmatrix}, [1-a=0], []\right]$$

From the constraint on `a`, we conclude that there is a 1-dimensional solution space for the special values $a = b = c = 1$ of the symbolic parameters.

```
delete A, B:
```

Example 8

Matrices from a domain such as `Dom::Matrix(...)` are converted to arrays with numbers or expressions. Hence, `numeric::matlinsolve` finds no solution for the following system:

```
M := Dom::Matrix(Dom::IntegerMod(7)):
A := M([[1, 4], [6, 3], [3, 2]]):
B := M([[9], [5], [0]]):
numeric::matlinsolve(A, B)
```

$$[\text{FAIL}, \text{NIL}]$$

Use `linalg::matlinsolve` to solve the system over the coefficient field of the matrices. A solution does exist over the field `Dom::IntegerMod(7)`:

```
linalg::matlinsolve(A, B)
```

$$\begin{pmatrix} 1 \bmod 7 \\ 2 \bmod 7 \end{pmatrix}$$

```
delete M, A, B:
```

Example 9

We demonstrate the difference between `Symbolic`, `HardwareFloats`, and `SoftwareFloats`. The following matrix A has a 1-dimensional kernel. Due to round-off, a further spurious kernel vector appears with `SoftwareFloats`. No kernel vector is detected with `HardwareFloats`:

```
A := matrix([[2*10^14 + 2, 2*10^(-9), 2*10^(-4)],
             [3*10^15 + 3, 3*10^(-8), 3*10^(-3)],
             [4*10^16 + 4, 4*10^(-7), 4*10^(-2)]
            ]):
b := matrix([2*10^(-9), 3*10^(-8), 4*10^(-7)]):
float(numeric::matlinsolve(A, b, Symbolic));
numeric::matlinsolve(A, b, SoftwareFloats);
numeric::matlinsolve(A, b, HardwareFloats)
```

$$\left[\begin{pmatrix} 0 \\ 1.0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ -100000.0 \\ 1.0 \end{pmatrix}\right]$$

$$\left[\begin{pmatrix} 1.0\,10^{-23} \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} -1.0\,10^{-23} & -1.0\,10^{-18} \\ 1.0 & 0 \\ 0 & 1.0 \end{pmatrix}\right]$$

$$\left[\begin{pmatrix} 0 \\ 1.0 \\ 0 \end{pmatrix}, 0\right]$$

```
delete A, b:
```

Parameters

A

An $m \times n$ matrix of domain type DOM_ARRAY, DOM_HFARRAY, or of category Cat::Matrix

B

An $m \times p$ matrix of domain type DOM_ARRAY, DOM_HFARRAY, or of category Cat::Matrix. Column vectors B may also be represented by a 1-dimensional array(1..m, [B_1, B_2, …]), a 1-dimensional hfarray(1..m, [B_1, B_2, …]), or by a list [B_1, B_2, …].

Options

Hard, HardwareFloats, Soft, SoftwareFloats

With Hard (or HardwareFloats), computations are done using fast hardware float arithmetic from within a MuPAD session. Hard and HardwareFloats are equivalent. With this option, the input data are converted to hardware floats and processed by compiled C code. The result is reconverted to MuPAD floats and returned to the MuPAD session.

With Soft (or SoftwareFloats) computations are dome using software float arithmetic provided by the MuPAD kernel. Soft and SoftwareFloats are equivalent. SoftwareFloats is used by default if the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY.

Compared to the SoftwareFloats used by the MuPAD kernel, the computation with HardwareFloats may be many times faster. Note, however, that the precision of hardware arithmetic is limited to about 15 digits. Further, the size of floating-point numbers may not be larger than approximately 10^{308} and not smaller than approximately 10^{-308}.

If no HardwareFloats or SoftwareFloats are requested explicitly, the following strategy is used: If the current value of DIGITS is smaller than 16 or if the matrix A is a hardware float array of domain type DOM_HFARRAY, then hardware arithmetic is tried. If this is successful, the result is returned.

If the result cannot be computed with hardware floats, software arithmetic by the MuPAD kernel is tried.

If the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY, or if one of the options Soft, SoftwareFloats or Symbolic is specified, MuPAD computes the result with its software arithmetic without trying to use hardware floats first.

There may be several reasons for hardware arithmetic to fail:

- The current value of DIGITS is larger than 15.
- The data contains symbolic objects.
- The data contains numbers larger than 10^{308} or smaller than 10^{-308} that cannot be represented by hardware floats.

If neither HardwareFloats nor SoftwareFloats is specified, the user is not informed whether hardware floats or software floats are used.

If HardwareFloats are specified but fail due to one of the reasons above, a warning is issued that the (much slower) software floating-point arithmetic of the MuPAD kernel is used.

Note that HardwareFloats can only be used if all input data can be converted to floating-point numbers.

The trailing digits in floating-point results computed with HardwareFloats and SoftwareFloats may differ.

Note: For ill-conditioned matrices, the results returned with HardwareFloats and SoftwareFloats may differ significantly! See “Example 9” on page 19-186.

Symbolic

Prevents the conversion of the input data to floating-point numbers. Exact arithmetic is used. This option overrides HardwareFloats and SoftwareFloats.

This option *must* be used if the matrix *A* contains symbolic parameters that cannot be converted to floating-point numbers.

Note: This option should not be used for matrices with floating-point entries! Numerical instabilities may occur in floating-point operations. Cf. “Example 6” on page 19-183.

ShowAssumptions

Returns information on internal assumptions on symbolic parameters in A and B. With this option, either exact arithmetic or SoftwareFloats are used.

This option is only useful if the matrices contain symbolic parameters. Consequently, it should only be used in conjunction with the option Symbolic.

Note: This option changes the format of the return value to [X, KernelBasis, Constraints, Pivots].

X and KernelBasis represent the general solution subject to Constraints and Pivots.

Constraints is a list of equations for symbolic parameters in B which are necessary and sufficient for $A\,X = B$ to be solvable.

Such constraints arise if Gaussian elimination leads to equations of the form $0 = c$, where c is some expression involving symbolic parameters contained in B. All such equations are collected in Constraints; numeric::matlinsolve assumes that these equations are satisfied and returns a special solution X.

If no such constraints arise, the return value of Constraints is the empty list.

Pivots is a list of inequalities involving symbolic parameters in A. Internally, division by pivot elements occurs in the Gaussian elimination. The expressions collected in Pivots are the numerators of those pivot elements that involve symbolic parameters contained in A. If only numerical pivot elements are used, then the return value of Pivots is the empty list.

Note: Constraints usually is a list of *non-linear* equations for the symbolic parameters. It is not investigated by numeric::matlinsolve, i.e., solutions may be returned, even if the Constraints cannot be satisfied. See "Example 3" on page 19-181.

Note: This option changes the return strategy for "unsolvable" systems. Without the option ShowAssumptions, the result [FAIL,NIL] is returned, whenever Gaussian elimination produces an equation $0 = c$ with non-zero c. With ShowAssumptions, such

equations are returned via `Constraints`, provided *c* involves symbolic parameters. If *c* is a purely numerical value, then `[FAIL, NIL, [], []]` is returned.

See "Example 3" on page 19-181, "Example 4" on page 19-182, and "Example 7" on page 19-184.

`NoWarning`

Suppresses warnings

If symbolic coefficients are found, `numeric::matlinsolve` automatically switches to the `Symbolic` mode with a warning. With this option, this warning is suppressed; `numeric::matlinsolve` still uses the symbolic mode for symbolic coefficients, i.e., exact arithmetic without floating-point conversions is used.

`ReturnType`

Option, specified as `ReturnType = d`

Return the (special) solution and the kernel as matrices of domain type `d`. The following return types *d* are available: `DOM_ARRAY`, `DOM_HFARRAY`, `Dom::Matrix()`, or `Dom::DenseMatrix()`.

`Sparse`

Use a sparse internal representation for matrices.

This option only has an effect when used in conjunction with `HardwareFloats`. With the `Sparse` option, the linear solver uses a sparse representation of the matrices to save memory and increase efficiency. However, if the coefficient matrix is not sparse, this option will cost some additional memory and runtime.

Return Values

Without the option `ShowAssumptions`, a list `[X, KernelBasis]` is returned. The (special) solution `X` is an $n \times p$ matrix. `KernelBasis` is an $n \times d$ matrix (d is the dimension of the kernel of A). Its columns span the kernel of A. If the kernel is trivial, `KernelBasis` is the integer 0.

`[FAIL, NIL]` is returned if the system is not solvable.

With `ShowAssumptions`, a list `[X, KernelBasis, Constraints, Pivots]` is returned. The lists `Constraints` and `Pivots` contain equations and inequalities involving symbolic parameters in `A` and `B`. Internally these were assumed to hold true when solving the system. `[FAIL, NIL, [], []]` is returned if the system is not solvable.

See Also

MuPAD Functions

`linalg::matlinsolve` | `linsolve` | `numeric::inverse` | `numeric::linsolve` | `solve`

numeric::ncdata

Weights and abscissae of Newton-Cotes quadrature

Syntax

```
numeric::ncdata(n)
```

Description

`numeric::ncdata(n)` returns the weights and the abscissae of the Newton-Cotes quadrature rule with n equidistant nodes.

The Newton-Cotes quadrature rule $\sum_{i=1}^{n} b_i f(c_i)$ produces the exact integral $\int_0^1 f(x)\,\mathrm{d}x$ for all polynomials f through degree n - 1. If n is odd, then the quadrature rule is exact through degree n.

The equidistant abscissae $c = [c_1, \ldots, c_n]$ are given by $c_i = \frac{i-1}{n-1}$.

Environment Interactions

`numeric::ncdata` is not sensitive to the environment variable `DIGITS`.

The function uses `option remember`.

Examples

Example 1

The following call produces exact data for the quadrature rule with four nodes:

```
numeric::ncdata(4)
```

$$\left[\left[\frac{1}{8}, \frac{3}{8}, \frac{3}{8}, \frac{1}{8}\right], \left[0, \frac{1}{3}, \frac{2}{3}, 1\right]\right]$$

Parameters

n

The number of nodes: a positive integer

Return Values

List `[b,c]` is returned. The lists `b = [`b_1`, …, `b_n`]` and `c = [`c_1`, …, `c_n`]` are the rational weights and abscissae of the Newton-Cotes quadrature rule, respectively.

Algorithms

The numerical integrator `numeric::quadrature` calls `numeric::ncdata` to provide the data for Newton-Cotes quadrature.

See Also

MuPAD Functions

`numeric::gldata` | `numeric::gtdata` | `numeric::int` | `numeric::quadrature`

numeric::odesolve

Numerical solution of an ordinary differential equation

Syntax

```
numeric::odesolve(f, t0 .. t, Y0, <method>, <RelativeError = rtol>, <AbsoluteError = at
```

```
numeric::odesolve(t0 .. t, f, Y0, <method>, <RelativeError = rtol>, <AbsoluteError = at
```

Description

`numeric::odesolve(f, t0..t, Y0)` returns a numerical approximation of the solution $Y(t)$ of the first order differential equation (dynamical system) $\frac{dY}{dt} = f(t, Y)$, $Y(t_0) = Y_0$ with $t, t_0 \in \mathbb{R}$ and $Y(t), Y_0 \in \mathbb{C}^n$.

`numeric::odesolve` is a general purpose solver able to deal with initial value problems of various kinds of ordinary differential equations. Equations $y^{(p)} = f\left(t, y, y', \ldots, y^{(p-1)}\right)$ of order p can be solved by `numeric::odesolve` after reformulation to dynamical system form. This can always be achieved by writing the equation as a first order system

$$\frac{d}{dt}\begin{pmatrix} Y_1 \\ \cdots \\ Y_{p-1} \\ Y_p \end{pmatrix} = \begin{pmatrix} Y_2 \\ \cdots \\ Y_p \\ f\left(t, Y_1, \ldots, Y_p\right) \end{pmatrix}$$

for the vector $Y = \left[Y_1, \ldots, Y_p\right] = \left[y, y', \ldots, y^{(p-1)}\right]$. See “Example 4” on page 19-199.

The following single-step Runge-Kutta-type methods are implemented:

- `EULER1` (order 1)

- `RKF43` (order 3)
- `xRKF43` (order 3)
- `RKF34` (order 4)
- `xRKF34` (order 4)
- `RK4` (order 4)
- `RKF54a` (order 4)
- `RKF54b` (order 4)
- `DOPRI54` (order 4)
- `xDOPRI54` (order 4)
- `CK54` (order 4)
- `xRKF54a` (order 4)
- `xRKF54b` (order 4)
- `xCK54` (order 4)
- `RKF45a` (order 5)
- `RKF45b` (order 5)
- `DOPRI45` (order 5)
- `CK45` (order 5)
- `xRKF45a` (order 5)
- `xRKF45b` (order 5)
- `xDOPRI45` (order 5)
- `xCK45` (order 5)
- `DOPRI65` (order 5)
- `xDOPRI65` (order 5)
- `DOPRI56` (order 6)
- `xDOPRI56` (order 6)
- `BUTCHER6` (order 6)
- `RKF87` (order 7)
- `xRKF87` (order 7)
- `DOPRI87` (order 7)

- `xDOPRI87` (order 7)
- `RKF78` (order 8)
- `xRKF78` (order 8)
- `DOPRI78` (order 8)
- `xDOPRI78` (order 8)
- `GAUSS(s)` (order 2 *s*)
- `GAUSS` = `s`

For the Gauss methods, `GAUSS(s)` is equivalent to `GAUSS = s`. The positive integer `s` indicates the number of stages. The order of the `s` stage Gauss method is 2 *s*.

The utility function `numeric::ode2vectorfield` may be used to produce the input parameters *f*, t_0, Y_0 from a set of differential expressions representing the ODE. See "Example 1" on page 19-198.

The input data t_0, t and Y_0 must not contain symbolic objects which cannot be converted to floating point values via `float`. Numerical expressions such as e^{π}, $\sqrt{2}$ etc. are accepted.

The vector field *f* defining the dynamical system $Y' = f(t, Y)$ must be represented by a procedure with two input parameters: the scalar time *t* and the vector *Y*. `numeric::odesolve` internally calls this function with real floating-point values *t* and a list *Y* of floating-point values. It has to return the vector *f*(*t*, *Y*) either as a list or as a 1-dimensional array. The output of *f* may contain numerical expressions such as π, e^2 etc. However, all values must be convertible to real or complex floating point numbers by `float`.

Autonomous systems, where *f*(*t*, *Y*) does not depend on *t*, must also be represented by a procedure with 2 arguments `t` and `Y`.

Scalar functions `Y` also must be represented by a list or an array with one element. For instance, the input data for the scalar initial value problem $y' = t\sin(y)$, $y(0) = 1$ may be of the form

```
f := proc(t,Y)                    /* Y is a 1-dimensional vector */
local y;                          /* represented by a list with */
```

```
begin                                   /* one element: Y = [y]. */
   y := Y[1];
   [t*sin(y)]                           /* the output is a list with 1 element */
end_proc:
Y0 := [1]:                              /* the initial value */
```

The numerical precision is controlled by the global variable `DIGITS`: an adaptive control of the step size keeps local relative discretization errors below `rtol=10^-DIGITS`, unless a different tolerance is specified via the option `RelativeError = rtol`. For small values of the solution vector *Y*, the absolute discretization error can be bounded by the threshold `atol` specified via the option `AbsoluteError = atol`.

If `AbsoluteError` is not specified, only relative discretization errors are controlled and kept below `rtol`.

The error control may be switched off by specifying a fixed `Stepsize = h`.

Note: Only local errors are controlled by the adaptive mechanism. No control of the global error is provided!

With `Y := t -> numeric::odesolve(f, t_0..t, Y_0)`, the numerical solution can be repesented by a MuPAD function: the call `Y(t)` will start the numerical integration from t_0 to `t`. A more sophisticated form of this function may be generated via `Y := numeric::odesolve2(f, t_0, Y_0)`.

This equips `Y` with a remember mechanism that uses previously computed values to speed up the computation. See “Example 2” on page 19-198.

For systems of the special form $Y' = f(t, Y)\,Y$ with a matrix valued function $f(t, Y)$, there is a special solver `numeric::odesolveGeometric` which preserves geometric features of the system more faithfully than `numeric::odesolve`.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We compute the numerical solution $y(10)$ of the initial value problem $y' = t \sin(y)$, $y(0) = 2$:

```
f := proc(t, Y) begin [t*sin(Y[1])] end_proc:
numeric::odesolve(f, 0..10, [2])
```

```
[3.141592654]
```

Alternatively, the utility function `numeric::ode2vectorfield` can be used to generate the input parameters in a more intuitive way:

```
[f, t0, Y0] :=
  [numeric::ode2vectorfield({y'(t) = t*sin(y(t)), y(0) = 2}, [y(t)])]
```

```
[proc f(t, Y) ... end, 0, [2]]
```

```
numeric::odesolve(f, t0..10, Y0)
```

```
[3.141592654]
```

```
delete f, t0, Y0:
```

Example 2

We consider $y' = y$, $y(0) = 1$. The numerical solution may be represented by the function

```
Y := t -> numeric::odesolve((t,Y) -> Y, 0..t, [1]):
```

Calling `Y(t)` starts the numerical integration:

```
Y(-5), Y(0), Y(1), Y(PI)
```

$$[0.006737946999], [1.0], [2.718281828], [23.14069263]$$

```
delete Y:
```

Example 3

We compute the numerical solution $Y(\pi) = [x(\pi), y(\pi)]$ of the system

$$x' = x + y,\ y' = x - y,\ x(0) = 1,\ y(0) = \sqrt{-1}$$

.

```
f := (t, Y) -> [Y[1] + Y[2], Y[1] - Y[2]]: Y0 := [1, I]:
numeric::odesolve(f, 0..PI, Y0)
```

$$[72.57057163 + 30.05484302\ i,\ 30.05484302 + 12.46088558\ i]$$

The solution of a linear dynamical system $Y' = A\ Y$ with a constant matrix A is $Y(t) = e^{t\ A}\ Y_0$. The solution of the system above can also be computed by:

```
t := PI: tA := array(1..2, 1..2, [[t, t], [t, -t]]):
numeric::expMatrix(tA, Y0)
```

$$\begin{pmatrix} 72.57057163 + 30.05484303\ i \\ 30.05484303 + 12.46088558\ i \end{pmatrix}$$

```
delete f, Y0, t, tA:
```

Example 4

We compute the numerical solution $y(1)$ of the differential equation $y'' = y^2$ with initial conditions $y(0) = 0$, $y'(0) = 1$. The second order equation is converted to a first order system for the vector $Y = [y, y'] = [y, z]$:

$$y' = z,\ z' = y^2,\ y(0) = 0,\ z(0) = 1$$

```
f := proc(t, Y) begin [Y[2], Y[1]^2] end_proc:
Y0 := [0, 1]:
numeric::odesolve(f, 0..1, Y0)
```

$$[1.087473533, 1.362851121]$$

```
delete f, Y0:
```

Example 5

We demonstrate how numerical data can be obtained on a user defined time mesh `t[i]`. The initial value problem is $y' = \sin(t) - y$, $y(0) = 1$, the sample points are $t_0 = 0.0$, $t_1 = 0.1$, ..., $t_{100} = 10.0$. First, we define the differential equation and the initial condition:

```
f := (t, Y) -> [sin(t) - Y[1]]:
Y[0] := [1]:
```

We define the time mesh:

```
for i from 0 to 100 do t[i] := i/10 end_for:
```

The equation is integrated iteratively from `t[i-1]` to `t[i]` with a working precision of 4 significant decimal places:

```
DIGITS := 4:
for i from 1 to 100 do
  Y[i] := numeric::odesolve(f, t[i-1]..t[i], Y[i-1])
end_for:
```

The following mesh data are produced:

```
[t[i], Y[i]] $ i = 0..100
```

$$\left[[0, [1]], \left[\frac{1}{10}, [0.9097]\right], \left[\frac{1}{5}, [0.8374]\right], \left[\frac{3}{10}, [0.7813]\right], \left[\frac{2}{5}, [0.7397]\right], \ldots, \left[\frac{99}{10}, [0.2159]\right], [10, [0.1476]]\right]$$

$$[0, [1]], \left[\frac{1}{10}, [0.9097]\right], \left[\frac{1}{5}, [0.8374]\right], \left[\frac{3}{10}, [0.7813]\right], \left[\frac{2}{5}, [0.7397]\right], \left[\frac{1}{2}, [0.7107]\right],$$

$$\left[\frac{3}{5}, [0.6929]\right], \left[\frac{7}{10}, [0.6846]\right], \left[\frac{4}{5}, [0.6843]\right], \left[\frac{9}{10}, [0.6907]\right], [1, [0.7024]], \left[\frac{11}{10}, [0.7181]\right],$$

$$\left[\frac{6}{5}, [0.7366]\right], \left[\frac{13}{10}, [0.7568]\right], \left[\frac{7}{5}, [0.7776]\right], \left[\frac{3}{2}, [0.7981]\right], \left[\frac{8}{5}, [0.8172]\right], \left[\frac{17}{10}, [0.8343]\right],$$

$$\left[\frac{9}{5}, [0.8485]\right], \left[\frac{19}{10}, [0.8591]\right], [2, [0.8657]], \left[\frac{21}{10}, [0.8677]\right], \left[\frac{11}{5}, [0.8647]\right], \left[\frac{23}{10}, [0.8564]\right],$$

$$\left[\frac{12}{5}, [0.8425]\right], \left[\frac{5}{2}, [0.8229]\right], \left[\frac{13}{5}, [0.7976]\right], \left[\frac{27}{10}, [0.7665]\right], \left[\frac{14}{5}, [0.7298]\right],$$

$$\left[\frac{29}{10}, [0.6876]\right], [3, [0.6402]], \left[\frac{31}{10}, [0.5879]\right], \left[\frac{16}{5}, [0.5311]\right], \left[\frac{33}{10}, [0.4702]\right], \left[\frac{17}{5}, [0.4057]\right],$$

$$\left[\frac{7}{2}, [0.3381]\right], \left[\frac{18}{5}, [0.2681]\right], \left[\frac{37}{10}, [0.1962]\right], \left[\frac{19}{5}, [0.1231]\right], \left[\frac{39}{10}, [0.04945]\right],$$

$$[4, [-0.02411]], \left[\frac{41}{10}, [-0.09687]\right], \left[\frac{21}{5}, [-0.1682]\right], \left[\frac{43}{10}, [-0.2373]\right], \left[\frac{22}{5}, [-0.3037]\right],$$

$$\left[\frac{9}{2}, [-0.3667]\right], \left[\frac{23}{5}, [-0.4257]\right], \left[\frac{47}{10}, [-0.4801]\right], \left[\frac{24}{5}, [-0.5295]\right], \left[\frac{49}{10}, [-0.5733]\right],$$

$$[5, [-0.6112]], \left[\frac{51}{10}, [-0.6428]\right], \left[\frac{26}{5}, [-0.6677]\right], \left[\frac{53}{10}, [-0.6858]\right], \left[\frac{27}{5}, [-0.697]\right],$$

$$\left[\frac{11}{2}, [-0.701]\right], \left[\frac{28}{5}, [-0.6979]\right], \left[\frac{57}{10}, [-0.6877]\right], \left[\frac{29}{5}, [-0.6705]\right], \left[\frac{59}{10}, [-0.6466]\right],$$

$$[6, [-0.6161]], \left[\frac{61}{10}, [-0.5794]\right], \left[\frac{31}{5}, [-0.5368]\right], \left[\frac{63}{10}, [-0.4888]\right], \left[\frac{32}{5}, [-0.4358]\right],$$

$$\left[\frac{13}{2}, [-0.3785]\right], \left[\frac{33}{5}, [-0.3173]\right], \left[\frac{67}{10}, [-0.2529]\right], \left[\frac{34}{5}, [-0.186]\right], \left[\frac{69}{10}, [-0.1171]\right],$$

$$[7, [-0.04709]], \left[\frac{71}{10}, [0.02345]\right], \left[\frac{36}{5}, [0.09378]\right], \left[\frac{73}{10}, [0.1632]\right], \left[\frac{37}{5}, [0.231]\right],$$

$$\left[\frac{15}{2}, [0.2965]\right], \left[\frac{38}{5}, [0.3591]\right], \left[\frac{77}{10}, [0.4181]\right], \left[\frac{39}{5}, [0.4729]\right], \left[\frac{79}{10}, [0.523]\right], [8, [0.5679]],$$

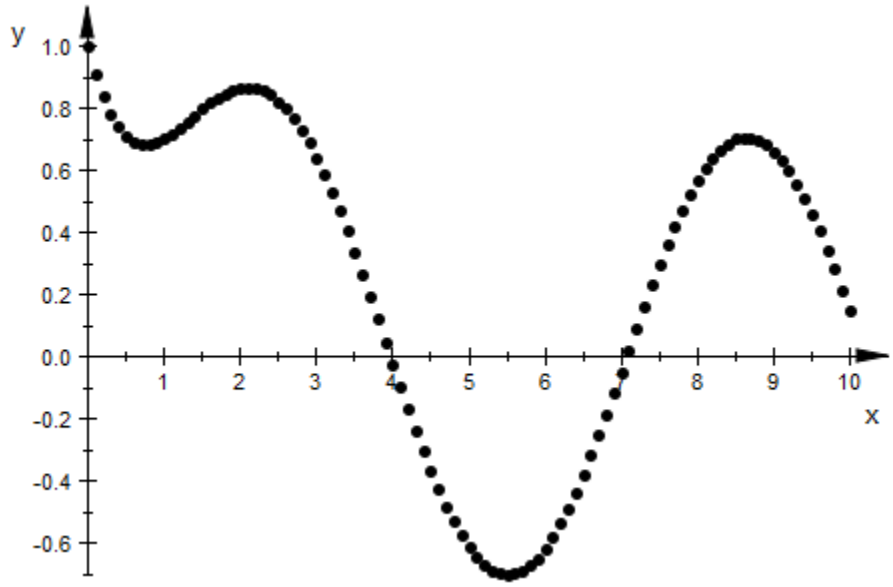
$$\left[\frac{81}{10}, [0.6072]\right], \left[\frac{41}{5}, [0.6404]\right], \left[\frac{83}{10}, [0.6671]\right], \left[\frac{42}{5}, [0.6873]\right], \left[\frac{17}{2}, [0.7006]\right],$$

$$\left[\frac{43}{5}, [0.7068]\right], \left[\frac{87}{10}, [0.7061]\right], \left[\frac{44}{5}, [0.6982]\right], \left[\frac{89}{10}, [0.6834]\right], [9, [0.6618]], \left[\frac{91}{10}, [0.6336]\right],$$

$$\left[\frac{46}{5}, [0.599]\right], \left[\frac{93}{10}, [0.5585]\right], \left[\frac{47}{5}, [0.5124]\right], \left[\frac{19}{2}, [0.4611]\right], \left[\frac{48}{5}, [0.4053]\right],$$

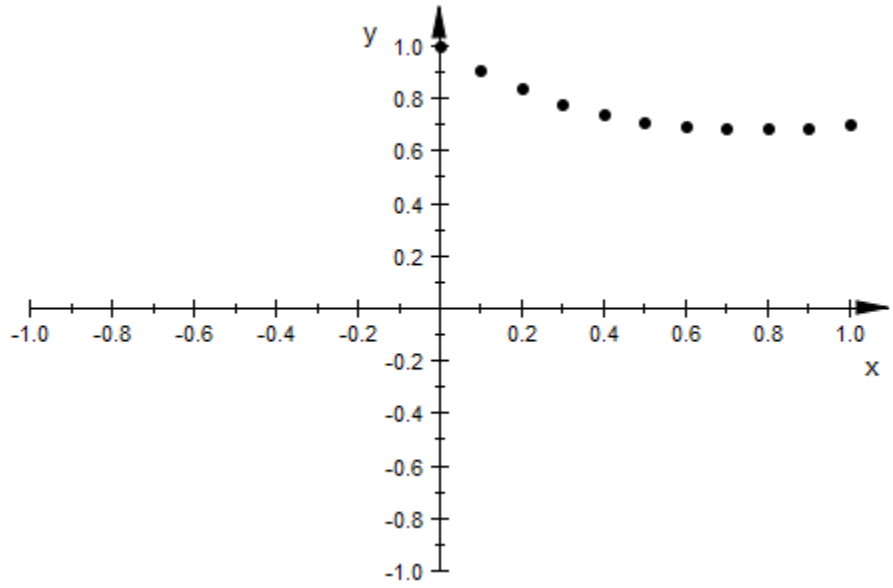
These data can be displayed by a list plot:

```
plotpoints := [[t[i], op(Y[i])] $ i = 0..100]:
plot(plot::PointList2d(plotpoints, PointColor = RGB::Black)):
```



The same plot can be obtained directly via `plot::Ode2d`:

```
plot(plot::Ode2d(
    [t[i] $ i = 0..100], f, Y[0],
    [(t, Y) -> [t, Y[1]], Style = Points, Color = RGB::Black]))
```




```
delete f, t, DIGITS, Y, plotpoints:
```

Example 6

We compute the numerical solution $y(1)$ of $y' = y$, $y(0) = 1$ by the classical 4-th order Runge-Kutta method RK4. By internal local extrapolation, its effective order is 5:

```
f := (t, Y) -> Y:
DIGITS := 13:
numeric::odesolve(f, 0..1, [1], RK4)
```

$$[2.718281828459]$$

Next, we use local extrapolation xRKF78 of the 8-th order submethod of the Runge-Kutta-Fehlberg pair RKF78. This scheme has effective order 9:

```
numeric::odesolve(f, 0..1, [1], xRKF78)
```

$$[2.718281828459]$$

Both methods yield the same result because of the internal adaptive error control. However, due to its higher order, the method `xRKF78` is faster.

```
delete f, DIGITS:
```

Example 7

We consider the stiff ODE $y' = 10^4\,(\cos(t) - y)$, $y(0) = 1$. The default method `DOPRI78` is explicit and not very efficient in solving very stiff problems:

```
f := (t, Y) -> [10^4*(cos(t) - Y[1])]:
t0 := time():
numeric::odesolve(f, 0..1, [1]), (time() - t0)*msec
```

$$[0.5403864476],\ 16065.004\ \text{msec}$$

We use the implicit A-stable method `GAUSS(6)`. For this stiff problem, it is more efficient than the default method `DOPRI78`:

```
t0 := time():
numeric::odesolve(f, 0..1, [1], GAUSS(6)), (time() - t0)*msec
```

$$[0.5403864476],\ 796.05\ \text{msec}$$

```
delete t0:
```

Example 8

We consider the initial value problem $y' = -10^{20}\,y\,(1 - \cos(y))$, $y(0) = 1$. We note that the numerical evaluation of the right hand side of the equation suffers from cancellation effects, when $|y|$ is small.

```
f := (t, Y) -> [-10^20*Y[1]*(1 - cos(Y[1]))]:
Y0 := [1]:
```

We first attempt to compute $y(1)$ with a working precision of 6 digits using the default setting `RelativeError = 10^-DIGITS=10^(-6)`:

```
DIGITS := 6: numeric::odesolve(f, 0..1, Y0)
```

```
[0.000000000329271]
```

Due to numerical round-off in the internal steps, no digit of this result is correct. Next, we use a working precision of 20 significant decimal places to eliminate roundoff effects:

```
DIGITS := 20:
numeric::odesolve(f, 0..1, Y0, RelativeError = 10^(-6))
```

```
[0.00000000010000000007495004558]
```

Since relative local discretization errors are of the magnitude 10^{-6}, not all displayed digits are trustworthy. We finally use a working precision of 20 digits and constrain the local relative discretization errors by the tolerance 10^{-10}:

```
numeric::odesolve(f, 0..1, Y0, RelativeError = 10^(-10))
```

```
[0.00000000010000000000003105061]
```

```
delete f, Y0, DIGITS:
```

Example 9

We compute the numerical solution $y(1)$ of $y' = y$, $y(0) = 1$ with various methods and various constant step sizes. We compare the result with the exact solution $y(1) = e = 2.718281828\ldots$.

```
f := (t, Y) -> Y:
Y0 := [1]:
```

We first use the Euler method of order 1 with two different step sizes:

```
Y := numeric::odesolve(f, 0..1, Y0, EULER1, Stepsize = 0.1):
Y, globalerror = float(exp(1)) - Y[1]
```

```
[2.59374246], globalerror = 0.1245393684
```

Decreasing the step size by a factor of 10 should reduce the global error by a factor of 10. Indeed:

```
Y := numeric::odesolve(f, 0..1, Y0, EULER1, Stepsize = 0.01):
```

```
Y, globalerror = float(exp(1)) - Y[1]
```

$$[2.704813829],\ \text{globalerror} = 0.01346799904$$

Next, we use the classical Runge-Kutta method of order 4 with two different step sizes:

```
Y := numeric::odesolve(f, 0..1, Y0, RK4, Stepsize = 0.1):
Y, globalerror = float(exp(1)) - Y[1]
```

$$[2.718279744],\ \text{globalerror} = 0.00000208432388$$

Decreasing the step size by a factor of 10 in a 4-th order scheme should reduce the global error by a factor of 10^4. Indeed:

```
Y := numeric::odesolve(f, 0..1, Y0, RK4, Stepsize = 0.01):
Y, globalerror = float(exp(1)) - Y[1]
```

$$[2.718281828],\ \text{globalerror} = 0.0000000002246438591$$

```
delete f, Y0, Y:
```

Example 10

We integrate $y' = y^2$, $y(0) = 1$ over the interval $t \in [0, 0.99]$ with a working precision of 4 digits. The exact solution is $y(t) = \frac{1}{1-t}$. Note the singularity at $t = 1$.

```
DIGITS := 4:
f := (t, Y) -> [Y[1]^2]:
Y0 := [1]:
```

The option `Alldata`, equivalent to `Alldata = 1`, yields all mesh data generated during the internal adaptive process:

```
numeric::odesolve(f, 0..0.99, Y0, Alldata)
```

$$[[0.0, [1.0]], [0.4933, [1.974]], [0.74, [3.846]], [0.8633, [7.314]], [0.9249, [13.32]],\ [0.9558, [22.61]], [0.9712, [34.71]], [0.9866, [74.68]], [0.99, [99.97]]]$$

With `Alldata = 2`, only each second point is returned:

```
numeric::odesolve(f, 0..0.99, Y0, Alldata = 2)
```

```
[[0.0, [1.0]], [0.74, [3.846]], [0.9249, [13.32]], [0.9712, [34.71]], [0.99, [99.97]]]
```

One can control the time mesh using the option `Stepsize = h`:

```
numeric::odesolve(f, 0..0.99, Y0, Stepsize=0.1, Alldata = 1)
```

```
[[0.0, [1.0]], [0.1, [1.111]], [0.2, [1.25]], [0.3, [1.429]], [0.4, [1.667]], [0.5, [2.0]], [0.6, [2.5]],
  [0.7, [3.333]], [0.8, [5.0]], [0.9, [10.0]], [0.99, [94.3]]]
```

However, with the option `Stepsize = h`, no automatic error control is provided by `numeric::odesolve`. Note the poor approximation $y(t) = 94.3$ for $t = 0.99$ (the exact value is $y(0.99) = 100.0$). The next computation with smaller step size yields better results:

```
numeric::odesolve(f, 0..0.99, Y0, Stepsize = 0.01, Alldata = 10)
```

```
[[0.0, [1.0]], [0.1, [1.111]], [0.2, [1.25]], [0.3, [1.429]], [0.4, [1.667]], [0.5, [2.0]], [0.6, [2.5]],
  [0.7, [3.333]], [0.8, [5.0]], [0.9, [10.0]], [0.99, [100.0]]]
```

"Example 5" on page 19-200 demonstrates how accurate numerical data on a user defined time mesh can be generated using the automatic error control of `numeric::odesolve`.

```
delete DIGITS, f, Y0:
```

Example 11

The second order equation $y'' + \sin(y) = 0$ is written as the dynamical system $y' = z$, $z' = -\sin(y)$ for the vector $Y = [y, z]$. A single symbolic step

$$[y(t_0), z(t_0)] \to [y(t_0 + h), z(t_0 + h)]$$

of the Euler method is computed:

```
f := proc(t, Y) begin [Y[2], -sin(Y[1])] end_proc:
numeric::odesolve(f, t0..t0+h, [y0, z0], EULER1, Symbolic)
```

$$[y0 + h\ z0,\ z0 - h\ \sin(y0)]$$

```
delete f:
```

Parameters

f

A procedure representing the vector field

t_0

A numerical real value for the initial time

t

A numerical real value (the “time”)

Y_0

A list or 1-dimensional array of numerical values representing the initial condition

method

One of the Runge-Kutta schemes listed below.

Options

BUTCHER6, CK45, CK54, DOPRI45, DOPRI54, DOPRI56, DOPRI65, DOPRI78, DOPRI87, EULER1, RK4, RKF34, RKF43, RKF45a, RKF45b, RKF54a, RKF54b, RKF78, RKF87, xCK45, xCK54, xDOPRI45, xDOPRI54, xDOPRI56, xDOPRI65, xDOPRI78, xDOPRI87, xRKF34, xRKF43, xRKF45a, xRKF45b, xRKF54a, xRKF54b, xRKF78, xRKF87

Name of the Runge-Kutta scheme. See “Example 6” on page 19-203. For details on these schemes, see the Algorithms section.

GAUSS

Name of the Runge-Kutta scheme specified as `GAUSS(s)` or `GAUSS = s`.

The methods `GAUSS(s)` or, equivalently, `GAUSS = s` are the implicit Gauss methods with `s` stages of order $2s$.

These methods are implicit A-stable schemes. The time steps are rather costly to compute. The Gauss methods are useful for integrating stiff ODEs. For non-stiff ODEs, there is usually no need to change the default method `DOPRI78`. This method is an embedded Runge-Kutta pair of orders 7 and 8.

Further, the Gauss methods are symplectic methods. When used with constant step size (`Stepsize = h`), numerical integration of Hamiltonian systems benefits from this property.

See “Example 7” on page 19-204.

`RelativeError`, `AbsoluteError`

Option specified as `RelativeError = rtol` forces internal numerical Runge-Kutta steps to use step sizes with relative local discretization errors below `rtol`. This tolerance must be a positive numerical real value not smaller than $\frac{1}{10^{\mathrm{DIGITS}}}$. The default tolerance is `RelativeError = 10^(-DIGITS)`.

Option specified as `AbsoluteError = atol` forces internal numerical Runge-Kutta steps to use step sizes with absolute local discretization errors below `atol`. This tolerance must be a nonnegative numerical real value. The default tolerance is `AbsoluteError = 10^(-10*DIGITS)`.

The internal control mechanism estimates the local discretization error of a Runge-Kutta step and adjusts the step size adaptively to keep this error below the specified tolerances `rtol` or `atol`, respectively. The code uses the criterion

$$\|\text{discretization error}\|_\infty = \max\left(\|Y\|_\infty \,\text{rtol},\ \text{atol}\right)$$

For accepting a solution vector Y. Roughly speaking, the relative error is controlled when the solution Y is sufficiently large. For very small solution values Y, absolute discretization errors are kept below the threshold `atol`.

Specify `AbsoluteError` = 0 if only control of the relative discretization errors is desired.

The error control may be switched off by specifying a fixed `Stepsize = h`.

The default setting of $\mathrm{rtol} = \frac{1}{10^{\mathrm{DIGITS}}}$, $\mathrm{atol} = \frac{1}{10^{10\ \mathrm{DIGITS}}}$ ensures that the local discretization errors are of the same order of magnitude as numerical roundoff.

Usually there is no need to use these options to change this setting. However, occasionally the numerical evaluation of the Runge-Kutta steps may be ill-conditioned or step sizes are so small that the time parameter cannot be incremented by the step size within working precision. In such a case these options may be used to bound the local discretization errors and use a higher working precision given by `DIGITS`.

Only positive real numerical values $\mathrm{rtol} \geq \frac{1}{10^{\mathrm{DIGITS}}}$ are accepted.

Note: The global error of the result returned by `numeric::odesolve` is usually larger than the local errors bounded by `rtol`, `atol`, respectively. Although the result is displayed with `DIGITS` decimal places one should not expect that all of them are correct. The relative precision of the final result is `rtol` at best!

See “Example 8” on page 19-204.

`Stepsize`

Option, specified as `Stepsize = h`

Switches off the internal error control and uses a Runge-Kutta iteration with constant step size `h` which must be a positive numerical value.

By default, `numeric::odesolve` uses an adaptive step size control mechanism in the numerical iteration. The option `Stepsize = h` switches off this adaptive mechanism and uses the Runge-Kutta method specified (or the default method `DOPRI78`) with constant step size `h`.

A final step with smaller step size is used to match the end `t` of the integration interval `t_0..t` if $\frac{t-t_0}{h}$ is not an integer.

Note: With this option, there is no automatic error control! Depending on the problem and on the order of the method the result may be a poor numerical approximation of the exact solution.

There is usually no need to invoke this option. However, occasionally the built-in adaptive error control mechanism may fail when integrating close to a singularity. In such a case this option may be used to customize a control mechanism for the global error by using different step sizes and investigating the convergence of the corresponding results.

Cf. “Example 9” on page 19-205.

MaxStepsize

Option, specified as `MaxStepsize = hmax`

Restricts adaptive step sizes to values not larger than `h_max`; `h_max` must be a positive numerical value.

By default, `numeric::odesolve` uses an adaptive step size control mechanism in the numerical iteration. The option `MaxStepsize = h_max` restricts the adaptive step size to values no larger than h_{max}.

If a larger stepsize `h` is requested explicitly by `Stepsize = h`, the option `MaxStepsize = h_max` reduces `h` to h_{max}.

Alldata

Option, specified as `Alldata = n`

Makes `numeric::odesolve` return a list of numerical mesh points generated by the internal Runge-Kutta iteration. The integer `n` controls the size of the output list.

With this option, `numeric::odesolve` returns a list of numerical mesh points $[[t_0, Y_0], [t_1, Y_1], \ldots, [t, Y(t)]]$ generated by the internal Runge-Kutta iteration.

The integer `n` controls the size of the output list. For $n = 1$, all internal mesh points are returned. This case may also be invoked by entering the simplified option `Alldata`, which is equivalent to `Alldata = 1`. For $n > 1$, only each n-th mesh point is stored in the list. The list always contains the initial point $[t_0, Y_0]$ and the final point $[t, Y(t)]$. For $n \leq 0$, only the data $[[t_0, Y_0], [t, Y(t)]]$ are returned.

The output list may be useful to inspect the internal numerical process. Also further graphical processing of the mesh data may be useful.

Cf. “Example 10” on page 19-206.

`Symbolic`

Makes `numeric::odesolve` return a vector of symbolic expressions representing a single symbolic step of the Runge-Kutta iteration.

The call `numeric::odesolve(f, t0..t, Y0, < method >, Symbolic)` returns a vector (list or array) of expressions representing a single step of the numerical scheme with step size t - t_0. In this mode symbolic values for t_0, t and the components of Y_0 are accepted.

This option may be useful if the specified numerical method applied to a given differential equation is to be investigated symbolically.

Cf. “Example 11” on page 19-207.

Return Values

The solution vector $Y(t)$ is returned as a list or as a 1-dimensional array of floating-point values. The type of the result vector coincides with the type of the input vector Y_0.

With the option `Alldata`, a list of mesh data is returned.

Algorithms

All methods presently implemented are adaptive versions of Runge-Kutta type single step schemes.

The methods `RKF43`, `RKF34`, `RKF54a`, `RKF54b`, `RKF45a`, `RKF45b`, `RKF87`, `RKF78`, `DOPRI54`, `DOPRI45`, `DOPRI65`, `DOPRI56`, `DOPRI87`, `DOPRI78`, `CK54`, `CK45` are embedded pairs of Runge-Kutta-Fehlberg, Dormand-Prince and Cash-Karp type, respectively. Estimates of the local discretization error are obtained in the usual way by comparing the results of the two submethods of the pair. The names indicate the orders of the subprocesses. For instance, `RKF34` and `RKF43` denote the same embedded Runge-Kutta-Fehlberg pair with orders 3 and 4. In `RKF34` the result of the fourth order submethod is accepted, whereas `RKF43` advances using the result of the third order submethod. In both cases the discretization error of the lower order subprocess is estimated and controlled.

For the single methods `EULER1` (the first order Euler scheme), `RK4` (the classical fourth order Runge-Kutta scheme) and `BUTCHER6` (a Runge-Kutta scheme of order 6), the

relative local error is controlled by comparing steps with different step sizes. The effective order of these methods is increased by one through local extrapolation.

Local extrapolation is also available for the submethods of the embedded pairs. For instance, the method `xRKF78` uses extrapolation of the 8-th order subprocess of `RKF78`, yielding a method of effective order 9. The 7-th order subprocess is ignored. The cheap error estimate based on the embedded pair is not used implying some loss of efficiency when comparing `RKF78` (order 8) and `xRKF78` (effective order 9).

The call `numeric::butcher(method)` returns the Butcher data of the methods used in `numeric::odesolve`. Here `method` is one of `EULER1`, `RKF43`, `RK4`, `RKF34`, `RKF54a`, `RKF54b`, `DOPRI54`, `CK54`, `RKF45a`, `RKF45b`, `DOPRI45`, `CK45`, `DOPRI65`, `DOPRI56`, `BUTCHER6`, `RKF87`, `DOPRI87`, `RKF78`, `DOPRI78`.

Note: Only local errors are controlled by the adaptive mechanism. No control of the global error is provided!

Note: The run time of the numerical integration with a method of order p grows like $O\left(10^{\frac{\text{DIGITS}}{p+1}}\right)$, when `DIGITS` is increased. Computations with high precision goals are very expensive! High order methods such as the default method `DOPRI78` should be used.

Presently, only single step methods of Runge-Kutta type are implemented. Stiff problems cannot be handled efficiently with explicit methods such as the default method `DOPRI78`. For stiff problems, one may use one of the implicit A-stable methods `GAUSS(s)`. See “Example 7” on page 19-204.

For problems of the special type $Y' = f(t, Y)\,Y$ with a matrix valued function $f(t, Y)$, there is a specialized (“geometric”) integration routine `numeric::odesolveGeometric`. Generally, `numeric::odesolve` is faster than the geometric integrator. However, `odesolveGeometric` preserves certain invariants of the system more faithfully.

References

J.C. Butcher: “The Numerical Analysis of Ordinary Differential Equations”, Wiley, Chichester (1987).

E. Hairer, S.P. Norsett and G. Wanner: “Solving Ordinary Differential Equations I”, Springer, Berlin (1993).

See Also

MuPAD Functions

`numeric::butcher` | `numeric::odesolve2` | `numeric::odesolveGeometric`

MuPAD Graphical Primitives

`plot::Ode2d` | `plot::Ode3d`

More About

- “Solve Equations Numerically”

numeric::odesolve2

Numerical solution of an ordinary differential equation

Syntax

```
numeric::odesolve2(f, t0, Y0, <method>, <RememberLast>, <RelativeError = rtol>, <Absolu
```

Description

`numeric::odesolve2( f, t0, Y0, … )` returns a function representing the numerical solution $Y(t)$ of the first order differential equation (dynamical system) $\frac{dY}{dt} = f(t, Y)$, $Y(t_0) = Y_0$ with $t, t_0 \in \mathbb{R}$ and $Y(t), Y_0 \in \mathbb{C}^n$.

The utility function `numeric::ode2vectorfield` may be used to produce the input parameters `f, t0, Y0` from a set of differential expressions representing the ODE. Cf. "Example 1" on page 19-216.

The function generated by `Y := numeric::odesolve2(f, t0, Y0)` is essentially

`Y := t -> numeric::odesolve(t_0..t, f, Y_0)`.

Numerical integration is launched, when `Y` is called with a real numerical argument. The call `Y(t)` returns the solution vector in a format corresponding to the type of the initial condition Y_0 with which `Y` was defined: `Y(t)` either yields a list or a 1-dimensional array.

If `t` is not a real numerical value, then `Y(t)` returns a symbolic function call.

See the help page of `numeric::odesolve` for details on the parameters and the options.

The options `Alldata = n` and `Symbolic` accepted by `numeric::odesolve` have no effect: `numeric::odesolve2` ignores these options.

Note: Without `RememberLast`, the function Y remembers all values it has computed. When calling $Y(T)$, it searches its remember table for the time $t0 < T < t$ closest to t, and integrates from T to t using the previously computed $Y(T)$ as initial value. Here,

t0 is the time for which the initial value of the ODE is given. This reduces the costs of a call considerably, if `Y` must be evaluated many times, for example, when plotting the ODE solution. The best approach is to call *Y* only with a monotonically increasing (or decreasing) sequence of values *t* starting from *t0*. Further, the function must be re-initialized whenever `DIGITS` is increased. See "Example 3" on page 19-218.

Environment Interactions

The function returned by `numeric::odesolve2` is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Without `RememberLast`, the function returned by `numeric::odesolve2` uses `option remember`.

Examples

Example 1

The numerical solution of the initial value problem $y' = t\sin(y)$, $y(0) = 2$ is represented by the following function $Y = [y]$:

```
f := (t, Y) -> [t*sin(Y[1])]:
```

Alternatively, the utility function `numeric::ode2vectorfield` can be used to generate the input parameters in a more intuitive way:

```
[f, t0, Y0] :=
  [numeric::ode2vectorfield({y'(t) = t*sin(y(t)), y(0) = 2}, [y(t)])]
```

```
     [proc f(t, Y) ... end, 0, [2]]
```

```
Y := numeric::odesolve2(f, t0, Y0)
```

```
     proc Y(t) ... end
```

The procedure `Y` starts the numerical integration when called with a numerical argument:

```
Y(-2), Y(0), Y(0.1), Y(PI + sqrt(2))
```

$$[2.968232567], [2.0], [2.004541745], [3.141552691]$$

Calling `Y` with a symbolic argument yields a symbolic call:

```
Y(t), Y(t + 5), Y(t^2 - 4)
```

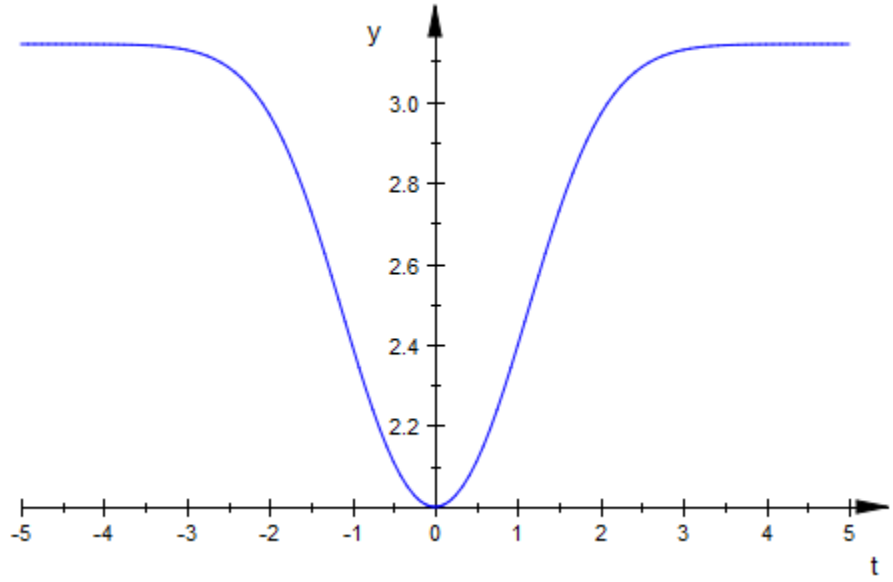
$$Y(t), Y(t+5), Y(t^2-4)$$

```
eval(subs(%, t = PI))
```

$$[3.132357009], [3.141592654], [3.141592611]$$

The numerical solution can be plotted. Note that `Y(t)` returns a list, so we plot the list element `Y(t)[1]`:

```
plotfunc2d(Y(t)[1], t = -5..5):
```



```
delete f, t0, Y0, Y:
```

Example 2

We consider the differential equation $y'' = y^2$ with initial conditions $y(0) = 0$, $y'(0) = 1$. The second order equation is converted to a first order system for $Y = [Y_1, Y_2] = [y, y']$:

$$Y_1' = Y_2,\ Y_2' = Y_1^2,\ Y_1(0) = 0,\ Y_2(0) = 1$$

```
f := (t, Y) -> [Y[2], Y[1]^2]:
t0 := 0: Y0 := [0, 1]:
Y := numeric::odesolve2(f, t0, Y0):
Y(1), Y(PI)
```

$$[1.087473533, 1.362851121], [1274.867469, 37166.52262]$$

```
delete f, t0, Y0, Y:
```

Example 3

We consider the system

$$x' = x + y,\ y' = x - y,\ x(0) = 1,\ y(0) = \sqrt{-1}:$$

```
f := (t, Y) -> [Y[1] + Y[2], Y[1] - Y[2]]:
Y := numeric::odesolve2(f, 0, [1, I]):
DIGITS := 5:
Y(1)
```

$$[3.5465 + 1.3683\ i,\ 1.3683 + 0.80988\ i]$$

Increasing `DIGITS` does not lead to a more accurate result because of the remember mechanism:

```
DIGITS := 15: Y(1)
```

$$[3.54648112716477 + 1.36829878277279\ i,\ 1.36829878277279 + 0.809883561619181\ i]$$

This is the previous value computed with 5 digits, printed with 15 digits. Indeed, only 5 digits are correct. For getting a result that is accurate to full precision, one has to erase the remember table via `Y:=subsop(Y,5=NIL)`. Alternatively, one may create a new numerical solution with a fresh (empty) remember table:

```
Y := numeric::odesolve2(f, 0, [1, I]):
Y(1)
```

$$[3.54648242861716 + 1.36829887200859\,i,\ 1.36829887200859 + 0.80988468459998\,i]$$

```
delete f, Y, DIGITS:
```

Example 4

We demonstrate the effect of the option `RememberLast`. We consider the ODE

$$y'(t) = -y + \sin(t),\ y(0) = 1$$:

```
f := (t, Y) -> [-Y[1] + sin(t)]:
Y := numeric::odesolve2(f, 0, [1]):
Z := numeric::odesolve2(f, 0, [1], RememberLast):
```

After many calls of `Y`, its remember table has grown large. In each call, searching the remember table for input parameters close to the present time value becomes expensive. Created with `RememberLast`, the procedure `Z` does not remember all its previously computed values apart from the last one. Consequently, it becomes faster than `Y`:

```
time(for i from 1 to 1000 do Y(i/100) end)*msec,
time(for i from 1 to 1000 do Z(i/100) end)*msec
```

11592.724 msec, 4352.272 msec

Apart from the efficiency, the values returned by `Y` and `Z` coincide:

```
Y(10.5), Z(10.5)
```

$$[-0.2020381113],\ [-0.2020381113]$$

```
delete f, Y, Z, i:
```

Parameters

f

A procedure representing the vector field of the dynamical system

t_0

A numerical real value for the initial time

Y_0

A list or 1-dimensional array of numerical values representing the initial value

method

One of the Runge-Kutta schemes listed below.

Options

BUTCHER6, CK45, CK54, DOPRI45, DOPRI54, DOPRI56, DOPRI65, DOPRI78, DOPRI87, EULER1, GAUSS, RK4, RKF34, RKF43, RKF45a, RKF45b, RKF54a, RKF54b, RKF78, RKF87, xCK45, xCK54, xDOPRI45, xDOPRI54, xDOPRI56, xDOPRI65, xDOPRI78, xDOPRI87, xRKF34, xRKF43, xRKF45a, xRKF45b, xRKF54a, xRKF54b, xRKF78, xRKF87

Option, specified as `GAUSS = s`

Name of the Runge-Kutta scheme. For details, see the documentation of `numeric::odesolve`.

RememberLast

Modifies the internal remember mechanism: the procedure returned by `numeric::odesolve2` does not remember the results of all previous calls, but only the result of the last call.

Without this option, the procedure returned by `numeric::odesolve2` employs `option remember` to remember the results of all preceding calls. If the function is

called very often (hundreds or thousands of times), the remember table grows large and searching this table for entries close to the current time value may become costly. With `RememberLast`, the procedure returned by `numeric::odesolve2` does not use `option remember` to remember *all* previous results but implements a very simple and inexpensive mechanism to remember only the result of the very last call.

This option is highly recommended when the numerical procedure returned by `numeric::odesolve2` is to be called often (hundreds of thousands of times) with monotonically increasing or decreasing time values. Cf. “Example 4” on page 19-219.

`RelativeError`

Option, specified as `RelativeError = rtol`

Forces internal numerical Runge-Kutta steps to use step sizes with relative local discretization errors below `rtol`. This tolerance must be a positive numerical real value not smaller than $\frac{1}{10^{DIGITS}}$. The default tolerance is `RelativeError = 10^(-DIGITS)`. See the help page of `numeric::odesolve` for further details.

`AbsoluteError`

Option, specified as `AbsoluteError = atol`

Forces internal numerical Runge-Kutta steps to use step sizes with absolute local discretization errors below `atol`. This tolerance must be a non-negative numerical real value. The default tolerance is `AbsoluteError = 10^(-10*DIGITS)`. See the help page of `numeric::odesolve` for further details.

`Stepsize`

Option, specified as `Stepsize = h`

Switches off the internal error control and uses a Runge-Kutta iteration with constant step size `h`. `h` must be a positive real value. See the help page of `numeric::odesolve` for further details.

`MaxStepsize`

Option, specified as `MaxStepsize = hmax`

Restricts adaptive step sizes to values not larger than h_{max}; h_{max} must be a positive numerical value. See the help page of `numeric::odesolve` for further details.

Return Values

Procedure.

See Also

MuPAD Functions

`numeric::ode2vectorfield` | `numeric::odesolve` | `numeric::odesolveGeometric`

MuPAD Graphical Primitives

`plot::Ode2d` | `plot::Ode3d`

More About

- "Solve Equations Numerically"

numeric::odesolveGeometric

Numerical solution of an ordinary differential equation on a homogeneous manifold

Syntax

```
numeric::odesolveGeometric(f, t0 .. t, Y0, <LieGroupAction = LAMBDA>, <method>, <Relati
numeric::odesolveGeometric(t0 .. t, f, Y0, <LieGroupAction = LAMBDA>, <method>, <Relati
```

Description

`numeric::odesolveGeometric(f, t_0..t, Y_0 )` approximates the solution of $\frac{dY(t)}{dt} = f(t, Y(t))\, Y(t)$, where $f(t, Y(t))$ returns $n \times n$ matrices and $Y \in \mathbb{C}^{n \times m}$.

`numeric::odesolveGeometric` is a "geometrical integrator" for ordinary differential equations on homogeneous manifolds embedded in the space of $n \times m$ matrices.

The call `numeric::odesolveGeometric(f, t_0..t, Y_0 )` returns a numerical approximation of the solution $Y(t)$ of the first order differential equation (dynamical system)

$$\frac{dY(t)}{dt} = f(t, Y(t))\, Y(t),\ Y(t_0) = Y_0$$

with $t, t_0 \in \mathbb{R}$. Here, $Y(t)$ is a curve of $n \times m$ matrices (or vectors in $\mathbb{R}^n$ or $\mathbb{C}^n$). The function f must produce $n \times n$ matrices as return values.

The following geometrical feature is preserved by the numerical solution: If the matrices produced by f lie in some Lie subalgebra g of the $n \times n$ matrices, then, within the numerical working precision, the approximation produced by `numeric::odesolveGeometric` stays on the homogeneous manifold $\{m\, Y_0 \mid m \in G\}$, where G is the matrix Lie group of g.

As an introductory example, consider the ODE $\frac{dY}{dt} = f(t, Y)\, Y$, where Y is a vector in $\mathbb{R}^n$ and `f` produces skew symmetric matrices. The solution lies on the orbit of the orthogonal

group $SO(n)$ generated by the skew symmetric matrices through the initial point $Y_0 \in \mathbb{R}^n$. Here, $SO(n)$ acts on $\mathbb{R}^n$ by standard matrix multiplication. The homogeneous manifold given by the orbit of $SO(n)$ through Y_0 is the sphere

$$\{G\,Y_0 \mid G \in \mathrm{SO}(n)\} = \left\{Y \in \mathbb{R}^n \mid \langle Y, Y\rangle = \langle Y_0, Y_0\rangle\right\}.$$

Using standard numerical schemes, the numerical solution drifts away from this manifold in the course of the integration. The geometrical "Lie group" integrator `numeric::odesolveGeometric`, however, produces a numerical solution that stays on this manifold, preserving the invariants of the group action. In this case, the invariant $\langle Y, Y\rangle$ is preserved numerically. See "Example 1" on page 19-227.

With $Y(t) = G(t)\,Y_0$, the matrix ODE

$$\frac{dG(t)}{dt} = f\left(t, G(t)\,Y_0\right)\,G(t),\ G\left(t_0\right) = 1_{n,n}$$

is solved on the space $\mathbb{C}^{n\times n}$ of the complex $n\times n$ matrices ($1_{n,n}$ is the identity matrix). Following Munthe-Kaas [1], the ansatz $G(t) = e^{u(t)}$ reduces a time step for the ode above to the solution of the matrix ODE

$$\frac{du}{dt} = \mathrm{dexpinv}(u, f) = f - \frac{[u, f]}{2} + \frac{[u, [u, f]]}{12} + \cdots,\ u\left(t_0\right) = 0,$$

where $f = f\left(t, e^{u(t)}\,Y_0\right)$ and $[u, f] = u\,f - f\,u$ is the commutator on the Lie algebra of $n\times n$ matrices. In each step, the ODE for u is solved numerically in a classical way by the Runge-Kutta scheme specified by the parameter `method`. Finally, `numeric::odesolveGeometric` performs the time step $t_0 \to t_0 + h$ by computing $Y\left(t_0 + h\right) = G\left(t_0 + h\right)\,Y_0 = e^{u(h)}\,Y_0$.

If the matrices produced by $f(t, Y)$ lie in a Lie subalgebra g of the $n\times n$ matrices, then the numerical solution u also lies in g. The matrix $G = e^u$ is an element of the corresponding Lie group, and $Y = G\,Y_0$ lies on the orbit of the Lie group through the initial value Y_0. Thus, the geometrical invariants of the homogeneous manifold are preserved in the course of the numerical integration.

The input data t_0 and t must not contain symbolic objects which cannot be converted to floating point values via `float`. Numerical expressions such as e^{π}, $\sqrt{2}$ etc. are accepted.

The initial condition Y_0 defines the space in which the homogeneous manifold containing the solution is embedded.

If Y_0 is a list with n entries or a 1-dimensional array `array(1..n)`, then the solution $Y(t)$ consists of vectors from a submanifold of $\mathbb{R}^n$ or $\mathbb{C}^n$, respectively.

If Y_0 is specified as a 2-dimensional `array(1..n, 1..m)` or as a matrix of the corresponding dimension generated by the function `matrix`, then the solution $Y(t)$ consists of matrices from a submanifold of the space $\mathbb{C}^{n\times m}$ of $n\times m$ matrices.

Internally, 2-dimensional $n\times m$ arrays are used to represent the points on the manifold where $m = 1$ for vectors in $\mathbb{R}^n$ or $\mathbb{C}^n$. It is recommended to specify Y_0 in the form `array(1..n, 1..m)` in order to avoid the overhead of internal conversions.

The "vector field" `f` defining the dynamical system $\frac{dY}{dt} = f(t, Y)\,Y$ must be represented by a procedure with two input parameters: the scalar time t and the matrix or vector Y. Internally, f is called with real floating-point values t and matrices/vectors Y of the same domain type as the initial condition Y_0.

The procedure `f` has to return an $n\times n$ matrix either as an `array(1..n, 1..n)` or as a corresponding matrix object of category `Cat::Matrix` (generated by the function `matrix`).

It is recommended that the procedure returns an array of the type `array(1..n, 1..n)`. This avoids the overhead of internal conversions.

The return value of `f` may contain numerical expressions such as π, $\sqrt{2}$ etc. However, all values must be convertible to real or complex floating point numbers by `float`.

Autonomous systems, where $f(t, Y)$ does not depend on t, must be represented by a procedure with two arguments `t` and `Y`, too.

The optional arguments `method`, `RelativeError = tol`, and `Stepsize = h` determine how the ODE $\frac{du}{dt} = \mathrm{dexpinv}(u, f)$ is solved. They correspond to the methods of the classical ODE solver `numeric::odesolve`.

The numerical precision is controlled by the global variable `DIGITS`: an adaptive control of the step size keeps local relative discretization errors below $\mathrm{tol} = \frac{1}{10^{\mathrm{DIGITS}}}$, unless a different tolerance is specified via the option `RelativeError = tol`. The error control may be switched off by specifying a fixed `Stepsize = h`.

Note: Only local errors are controlled by the adaptive mechanism. No control of the global error is provided!

With `Y := t -> numeric::odesolveGeometric(f, t_0..t, Y_0)`, the numerical solution can be repesented by a MuPAD function: the call `Y(t)` will start the numerical integration from t_0 to t.

Classical integration preserves the geometrical invariants up to the relative precision of the solution whereas the geometrical integrator preserves the invariants independent of *tol* up to the working precision set by `DIGITS`: departure from the homogeneous manifold is a pure roundoff effect.

`numeric::odesolveGeometric` is useful when a tolerance *tol* much largerthan $\frac{1}{10^{\mathrm{DIGITS}}}$ is specified by `RelativeError = tol`. For small tolerances, you may consider to use the classical solver `numeric::odesolve` instead.

Since classical integration is significantly faster, larger values of `DIGITS` and smaller tolerances for the discretization error may be used in `numeric::odesolve`. Depending on the concrete problem, this may lead to better results than produced by `numeric::odesolveGeometric`.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We consider the initial value problem

$$\frac{dY}{dt} = \begin{pmatrix} (J_2 - J_3)\, Y_2\, Y_3 \\ (J_3 - J_1)\, Y_3\, Y_1 \\ (J_1 - J_2)\, Y_1\, Y_2 \end{pmatrix}, \quad Y(0) = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

for $Y = [Y_1, Y_2, Y_3] \in \mathbb{R}^3$ with fixed parameters $J_1 = \frac{1}{2}$, $J_2 = 1$, $J_3 = 2$. Writing this ODE as

$$\frac{dY}{dt} = \begin{pmatrix} 0 & -J_3\, Y_3 & J_2\, Y_2 \\ J_3\, Y_3 & 0 & -J_1\, Y_1 \\ -J_2\, Y_2 & J_1\, Y_1 & 0 \end{pmatrix} \begin{pmatrix} Y_1 \\ Y_2 \\ Y_3 \end{pmatrix} = f_1(t, Y)\, Y \,,$$

it is clear that the solution is restricted to the orbit of the orthogonal group $SO(3)$ through the initial point (f_1 produces skew symmetric matrices). The invariant of this action is $H_1(Y) = \langle Y, Y \rangle = {Y_1}^2 + {Y_2}^2 + {Y_3}^2$, i.e., the solution is restricted to a sphere. Writing the ODE as

$$\frac{dY}{dt} = \begin{pmatrix} 0 & J_2\, Y_3 & -J_3\, Y_2 \\ -J_1\, Y_3 & 0 & J_3\, Y_1 \\ J_1\, Y_2 & -J_2\, Y_1 & 0 \end{pmatrix} \begin{pmatrix} Y_1 \\ Y_2 \\ Y_3 \end{pmatrix} = f_2(t, Y)\, Y$$

$$= \begin{pmatrix} 0 & Y_3 & -Y_2 \\ -Y_3 & 0 & Y_1 \\ Y_2 & -Y_1 & 0 \end{pmatrix} \begin{pmatrix} J_1 & 0 & 0 \\ 0 & J_2 & 0 \\ 0 & 0 & J_3 \end{pmatrix} \begin{pmatrix} Y_1 \\ Y_2 \\ Y_3 \end{pmatrix},$$

it is clear that the solution is also restricted to the orbit of the "J-orthogonal" group $SO(J, 3)$ through the initial point. This group consists of matrices G satisfying $G^T J G = J$, where $J = diag(J_1, J_2, J_3)$. The invariant of this group action is $H_2(Y) = \langle Y, J Y \rangle = J_1 Y_1^2 + J_2 Y_2^2 + J_3 Y_3^2$, i.e., the solution is restricted to an ellipsoid.

We consider the first representation and compute a numerical solution that is restricted to a sphere:

```
f1 := proc(t, Y) begin
      array(1..3, 1..3, [ [   0    , -J3*Y[3],  J2*Y[2]],
                          [ J3*Y[3],    0    , -J1*Y[1]],
                          [-J2*Y[2],  J1*Y[1],    0    ]])
end_proc:
J1 := 1/2: J2 := 1: J3 := 2:
tol := 10^(-2):
Gsolve:= (f, t0_t, Y0) ->
  numeric::odesolveGeometric(f, t0_t, Y0, RelativeError = tol):

Y(0) := [1.0, 1.0, 1.0];
Y(1) := Gsolve(f1, 0..1, Y(0));
Y(2) := Gsolve(f1, 1..2, Y(1));
Y(3) := Gsolve(f1, 2..3, Y(2));
Y(4) := Gsolve(f1, 3..4, Y(3));
Y(5) := Gsolve(f1, 4..5, Y(4))
```

```
[1.0, 1.0, 1.0]

[-0.1234874253, 1.573899124, 0.7124551935]

[-1.188837997, 0.6164504304, 1.098477621]

[-0.7811791671, -1.258922776, 0.8971468953]
```

$$[0.3530511101, -1.520952843, 0.7497048516]$$

$$[1.281782142, -0.1892246302, 1.149447075]$$

The invariant H_1 is preserved numerically up to the working precision set by `DIGITS`:

```
H1 := Y -> Y[1]^2 + Y[2]^2 + Y[3]^2:
H1(Y(i)) - H1(Y(0)) $ i = 1..5
```

$$3.045480534\,10^{-14}, -2.691596945\,10^{-14}, 4.011374566\,10^{-14}, 3.491859579\,10^{-13},$$

$$2.851746617\,10^{-13}$$

The invariant H_2 is only preserved within the relative precision of the solution set by the option `RelativeError = tol`:

```
H2 := Y -> J1*Y[1]^2 + J2*Y[2]^2 + J3*Y[3]^2:
H2(Y(i)) - H2(Y(0)) $ i = 1..5
```

$$-0.00003216929769, -0.00001480619998, -0.0002478937317, -0.0002651787298,$$
$$-0.0002541517488$$

Now, we solve the ODE using the second representation:

```
f2 := proc(t, Y) begin
      array(1..3, 1..3, [ [   0    ,  J2*Y[3], -J3*Y[2]],
                          [-J1*Y[3],    0    ,  J3*Y[1]],
                          [ J1*Y[2], -J2*Y[1],    0    ]])
end_proc:

Y(0) := [1.0, 1.0, 1.0];
Y(1) := Gsolve(f2, 0..1, Y(0));
Y(2) := Gsolve(f2, 1..2, Y(1));
Y(3) := Gsolve(f2, 2..3, Y(2));
Y(4) := Gsolve(f2, 3..4, Y(3));
Y(5) := Gsolve(f2, 4..5, Y(4))
```

$$[1.0, 1.0, 1.0]$$

$$[-0.1234661822, 1.573904531, 0.7124614294]$$

$$[-1.188842584,\ 0.6165012417,\ 1.098465492]$$

$$[-0.7810696492,\ -1.258800682,\ 0.8973254548]$$

$$[0.3534089311,\ -1.520647212,\ 0.7500610376]$$

$$[1.28178089,\ -0.1880404787,\ 1.149599855]$$

Now, the invariant H_2 is preserved to working precision, whilst H_1 is only preserved to the tolerance specified by `RelativeError = tol`:

```
H2(Y(i)) - H2(Y(0)) $ i = 1..5
```

$$2.70894418\ 10^{-14},\ 5.287437155\ 10^{-15},\ 2.071884331\ 10^{-13},\ 1.704608676\ 10^{-13},\ 2.268046861\ 10^{-13}$$

```
H1(Y(i)) - H1(Y(0)) $ i = 1..5
```

$$0.0000206606336,\ 0.00004690748333,\ -0.0001580733145,\ -0.0001426238818,\ -0.0000987012679$$

```
delete J1, J2, J3, Gsolve, f1, f2, Y, H1, H2:
```

Example 2

We consider the "Toda lattice equations"

$$\frac{da_i}{dt} = a_i\left(b_i - b_{i+1}\right),\ i \in \{1, \ldots, n-1\},$$

$$\frac{db_i}{dt} = 2\left({a_{i-1}}^2 - a_i^2\right),\ i \in \{1, \ldots, n\}$$

with $a_0 = a_n = 0$. Introducing the tridiagonal $n \times n$ matrices

$$Y = \begin{pmatrix} b_1 & a_1 & & \\ a_1 & \cdots & \cdots & \\ & \cdots & \cdots & a_{n-1} \\ & & a_{n-1} & b_n \end{pmatrix}, f(Y) = \begin{pmatrix} 0 & -a_1 & & \\ a_1 & \cdots & \cdots & \\ & \cdots & \cdots & -a_{n-1} \\ & & a_{n-1} & 0 \end{pmatrix},$$

these equations can be encoded by the matrix ODE $\frac{dY}{dt} = f(Y)\,Y - Y\,f(Y)$. The solution $Y(t)$ is known to be "isospectral", i.e., the eigenvalues of $Y(t)$ do not depend on the time parameter t. As mentioned in the description of the option `LieGroupAction`, the solution of this type of matrix ODE is given by the group action $Y(t) = G(t)\,Y(0)\,G(t)^{-1} = G(t)\,Y(0)\,G(t)^T$, where $G(t)$ are orthogonal matrices (note that $f(Y)$ is skew symmetric). The eigenvalues of the matrices $Y(t)$ are invariants of the group action.

The exact dynamics also preserves the tridiagonal form of the matrices. The numerical dynamics, however, fills in further elements. The following vector field f ignores alle elements outside the central bands:

```
f := proc(t, Y)
local i, r;
begin
  r := array(1..n, 1..n, [[0 $ n] $ n]);
  for i from 1 to n - 1 do
    r[i + 1, i] :=  Y[i, i + 1];
    r[i, i + 1] := -Y[i, i + 1];
  end_for;
  return(r)
end_proc:
```

In the following, the initial value $Y(0)$ is specified by a matrix generated by the function `matrix`. Consequently, both arguments G and Y are passed to the Lie group action `LAMBDA` as corresponding matrices. They can be multiplied by the multiplication operator `*`:

```
LAMBDA:= proc(G, Y)
begin
  G*Y*(G::dom::transpose(G))
end_proc:
```

We define the initial value:

```
n := 3:
Y(0) := matrix(n, n, [1, 1, 1], Banded)
```

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

Now, the dynamics is integrated from $t = 0$ to $t = 1$:

```
tol := 10^(-4):
Y(1) := numeric::odesolveGeometric(f, 0..1, Y(0),
        LieGroupAction = LAMBDA, RelativeError = tol)
```

$$\begin{pmatrix} -0.256354686 & 0.4591148565 & -1.716329601\ 10^{-14} \\ 0.4591148565 & 1.0 & 0.4591148565 \\ -1.716329601\ 10^{-14} & 0.4591148565 & 2.256354686 \end{pmatrix}$$

The invariants of the dynamics are the eigenvalues of the matrices $Y(t)$. They are preserved numerically:

```
numeric::eigenvalues(Y(0)) = numeric::eigenvalues(Y(1))
```

$$[2.414213562,\ 1.0,\ -0.4142135624] = [2.414213562,\ 1.0,\ -0.4142135624]$$

For comparison, we also solve the Toda lattice equations by classical numerics using `numeric::odesolve`. The system is encoded by a vector $Y = [b_1, \ldots, b_n, a_1, \ldots, a_{n-1}]$ in $\mathbb{R}^{2n-1}$:

```
f := proc(t, Y)
local a, b, i;
begin
  b := [Y[i] $ i = 1..n];
  a := [Y[n + i] $ i = 1..n-1];
  [-2*a[1]^2,                          // = d/dt b[1]
    2*(a[i-1]^2 - a[i]^2) $ i = 2..n-1, // = d/dt b[i]
    2*a[n-1]^2,                        // = d/dt b[n]
    a[i]*(b[i] - b[i+1]) $ i = 1..n-1   // = d/dt a[i]
  ]
end_proc:
```

```
solution := numeric::odesolve(f, 0..1, [1 $ 2*n - 1],
                              RelativeError = tol);
```

$$[-0.2563650696,\ 1.0,\ 2.25636507,\ 0.459099923,\ 0.459099923]$$

The invariants are only preserved up to the precision of the solution determined by the tolerance set via `RelativeError = tol`:

```
Y(1) := array(1..n, 1..n, [[0 $ n] $ n]):
for i from 1 to n do
  Y(1)[i, i] := solution[i];
end_for:
for i from 1 to n-1 do
  Y(1)[i, i + 1] := solution[n + i];
  Y(1)[i + 1, i] := solution[n + i];
end_for:

Y(1)
```

$$\begin{pmatrix} -0.2563650696 & 0.459099923 & 0 \\ 0.459099923 & 1.0 & 0.459099923 \\ 0 & 0.459099923 & 2.25636507 \end{pmatrix}$$

```
numeric::eigenvalues(Y(1))
```

$$[2.414213091,\ 1.0,\ -0.4142130909]$$

Comparing these data with the previously computed eigenvalues of the initial condition $Y(0)$, one sees that the invariants are not preserved numerically to the working precision determined by `DIGITS`.

```
delete f, LAMBDA, n, Y, tol, solution, i:
```

Parameters

`f`

A procedure accepting two parameters `(t, Y)`

t_0

A numerical real value for the initial time

t

A numerical real value (the “time”)

Y_0

The initial condition: a list, a 1-dimensional `array(1..n)`, a 2-dimensional `array(1..n, 1..m)`, or an $n \times m$ `matrix` of category `Cat::Matrix` with numerical entries

Options

LieGroupAction

Option, specified as `LieGroupAction = LAMBDA`

The procedure `LAMBDA = proc(G, Y) ... end_proc` defines the action of the group element `G` (an $n \times n$ matrix) on the point Y on the homogeneous manifold (an $n \times m$ matrix or an n dimensional vector). This procedure must return a corresponding point (a matrix or a vector).

The default action is the usual matrix multiplication $(G, Y) \to G\,Y$.

With this option, the default group action `LAMBDA`: $(G, Y) \to G\,Y$ of the $n \times n$ matrices G acting on the $n \times m$ matrices or n dimensional vectors `Y` by left multiplication may be replaced by other group actions.

As a group action, the procedure `LAMBDA` must satisfy $LAMBDA(1_{n,n}, Y) = Y$ and

$$\mathrm{LAMBDA}(G_2, \mathrm{LAMBDA}(G_1, Y)) = \mathrm{LAMBDA}(G_2\,G_1, Y)$$

`numeric::odesolveGeometric` computes the solution of the matrix ODE

$$\frac{dG(t)}{dt} = f(t, \mathrm{LAMBDA}(G(t), Y_0))\,G(t),\ G(t_0) = 1_{n,n}$$

On the space $\mathbb{C}^{n \times n}$ of the $n \times n$ matrices and returns $Y(t) = LAMBDA(G(t), Y_0)$.

For the standard group action *LAMBDA*(*G*, *Y*) = *G Y*, this is the solution of the ODE $\frac{dY}{dt} = f(t, Y)\, Y$.

For homogeneous manifolds embedded in the *n*×*n* matrices, the group action *LAMBDA*(*G*, *Y*) = $G\, Y\, G^{-1}$ may be considered. For this action, the curve *Y*(*t*) = *LAMBDA*(*G*(*t*), Y_0) returned by `numeric::odesolveGeometric` is the solution of the ODE $\frac{dY}{dt} = f(t, Y)\, Y - Y\, f(t, Y)$. Cf. "Example 2" on page 19-230.

`LAMBDA(G, Y)` is called with *n*×*m* matrices or *n* dimensional vectors `Y` of the same domain type as the initial condition Y_0. If Y_0 is a matrix generated by the function `matrix`, then also the *n*×*n* matrix *G* is passed to `LAMBDA` as such a matrix object. In all other cases, *G* is passed as a 2-dimensional `array(1..n, 1..n)`.

The procedure `LAMBDA` should return a 2-dimensional `array(1..n, 1..m)` or a corresponding matrix of category `Cat::Matrix`.

If the initial condition Y_0 is specified by a list or a 1-dimensional `array(1..n)`, `LAMBDA` may also return a corresponding list or array.

Internally, the return value of `LAMBDA` is converted to a 2-dimensional `array(1..n, 1..m)` where *m* = 1 if a list or a 1-dimensional array is returned.

It is recommended that `LAMBDA` returns a 2-dimensional `array(1..n, 1..m)` in order to avoid the overhead of internal conversions.

`RelativeError`

Option, specified as `RelativeError = tol`

Forces internal numerical Runge-Kutta steps to use step sizes with local discretization errors below `tol`. This tolerance must be a numerical real value $\geq \frac{1}{10^{\mathrm{DIGITS}}}$. The default tolerance is $\frac{1}{10^{\mathrm{DIGITS}}}$.

The internal control mechanism estimates the local relative discretization error of a Runge-Kutta step and adjusts the step size adaptively to keep this error below `tol`.

The default setting of $\mathrm{tol} = \frac{1}{10^{\mathrm{DIGITS}}}$ ensures that the local discretization errors are of the same order of magnitude as numerical roundoff.

Usually there is no need to use this option to change this setting. However, occasionally the numerical evaluation of the Runge-Kutta steps may be ill-conditioned or step sizes are so small that the time parameter cannot be incremented by the step size within working precision. In such a case, this option may be used to bound the local discretization error by *tol* and use a higher working precision given by `DIGITS`.

Only real numerical values $\text{tol} \geq \frac{1}{10^{\text{DIGITS}}}$ are accepted.

Note: Usually, the global error of the numeric approximation returned by `numeric::odesolveGeometric` is larger than the local errors bounded by `tol`. Although the result is displayed with `DIGITS` decimal places, one should not expect that all of them are correct. The relative precision of the final result is `tol` at best!

`Stepsize`

Option, specified as `Stepsize = h`

Switches off the internal error control and uses a Runge-Kutta iteration with constant step size `h`. `h` must be a numerical real value.

By default, `numeric::odesolveGeometric` uses an adaptive step size control mechanism in the numerical iteration. The option `Stepsize = h` switches off this adaptive mechanism and uses the Runge-Kutta method specified by `method` (or the default method `DOPRI78`) with constant step size `h`.

A final step with smaller step size is used to match the end t of the integration interval $t_0 .. t$, if $\frac{t-t_0}{h}$ is not an integer.

Note: With this option, there is no automatic error control! Depending on the problem and on the order of the method, the result may be a poor numerical approximation of the exact solution.

There is usually no need to invoke this option. However, occasionally the builtin adaptive error control mechanism may fail when integrating close to a singularity. In such a case, this option may be used to customize a control mechanism for the global error by using different step sizes and investigating the convergence of the corresponding results.

Alldata

Option, specified as `Alldata = n`

With this option, `numeric::odesolveGeometric` returns a list of numerical mesh points $[[t_0, Y_0], [t_1, Y_1], \ldots, [t, Y(t)]]$ generated by the internal Runge-Kutta iteration.

The integer n controls the size of the output list. For $n = 1$, all internal mesh points are returned. This case may also be invoked by entering the simplified option `Alldata`, which is equivalent to `Alldata = 1`. For $n > 1$, only each n-th mesh point is stored in the list. The list always contains the initial point $[t_0, Y_0]$ and the final point $[t, Y(t)]$. For $n \leq 0$, only the data $[[t_0, Y_0], [t, Y(t)]]$ are returned.

The output list may be useful to inspect the internal numerical process. Also further graphical processing of the mesh data may be useful.

Return Values

The solution $Y(t)$ is returned as a list or as an array of floating-point values. The type of the result matrix/vector coincides with the type of the input matrix/vector Y_0.

With the option `Alldata`, a list of mesh data is returned.

References

[1] H. Munthe-Kaas and A. Zanna: “Numerical integration of differential equations on homogeneous manifolds”, in F. Cucker (ed.), *Foundations of Computational Mathematics*, Springer (1997), pp. 305-315.

See Also

MuPAD Functions

`numeric::butcher` | `numeric::odesolve` | `numeric::odesolve2`

MuPAD Graphical Primitives

`plot::Ode2d` | `plot::Ode3d`

numeric::ode2vectorfield

Convert an ode system to vectorfield notation

Syntax

```
numeric::ode2vectorfield(IVP, fields)
```

Description

`numeric::ode2vectorfield` converts a system of ordinary differential equations of arbitrary order to a vector field representation suitable for the numerical ODE solver `numeric::odesolve2`.

`numeric::ode2vectorfield` and `numeric::odeToVectorField` are equivalent.

`numeric::ode2vectorfield` is a utility function to generate input parameters for the numerical ODE solver `numeric::odesolve2`. This solver requires a procedure representing the vectorfield $f(t, Y)$ of a first order system of differential equations (dynamic system) $\frac{dY}{dt} = f(t, Y)$ and initial data $Y_0 = Y(t_0)$. Given an initial value problem `IVP` consisting of (possibly higher order) differential expressions together with initial conditions, `numeric::ode2vectorfield` converts the higher order equations to an equivalent system of first order ODEs and returns the input parameters for `numeric::odesolve2`.

Higher-order differential equations can always be represented as an equivalent dynamic system $\frac{dY}{dt} = f(t, Y)$ with some vector Y. E.g., the n-th order equation

$$y^{(n)} = g\left(t, y, y', \ldots, y^{(n-1)}\right)$$

may be written as the first order system

$$\frac{dY}{dt} = \left[y', \ldots, y^{(n-1)}, g\left(t, y, \ldots, y^{(n-1)}\right)\right]$$

for the vector $Y = \left[y, y', \ldots, y^{(n-1)}\right]$.

The input list `fields` correponds to the vector *Y*. It must be a *complete* specification of all functions and their derivatives through but *not including the highest derivates* of the unknown functions. E.g., for the second order differential equation $y''(t) = y(t)$, the appropriate list of unknown fields is $[y(t), y'(t)]$. The differential equations $y''(t) = z(t)$, $z'(t) = y(t)$ are of second order in *y* and of first order in *z*. Hence, the appropriate list of unknown fields is `[y(t), y'(t), z(t)]`.

The ordering of the fields in `list` determines the ordering of the components of the list that the numerical solver produces as the solution vector. Cf. “Example 2” on page 19-240.

The differential equations must be linear in the highest derivatives of the unknown functions involved. E.g., the ODE $y'(t)^2 = y(t)$ is not admitted. However, equations such as $y_2(t)\, y_1'(t) = y_1(t)$, $y_1(t)\, y_2'(t) = y_2(t)$ are accepted and converted to $y_1'(t) = \frac{y_1(t)}{y_2(t)}$, $y_2'(t) = \frac{y_2(t)}{y_1(t)}$.

A *complete* specification of initial conditions must be contained in `IVP`: for each component in `list`, an initial value must be provided. The initial conditions may be specified by linear equations which will be solved for the initial values of the unknown fields automatically. E.g., for `fields = [y(t), y'(t), z(t)]`, initial conditions may be specified explicitly by $y(t_0) = 1$, `y'(t_0) = 2`, $z(t_0) = 3$, say, or via linear equations such as `y(t_0) + z(t_0) = y'(t_0)`, $y(t_0) = z(t_0)$, $z(t_0) = 2\, y(t_0)$. Cf. “Example 3” on page 19-241.

The differential equations, the initial ‘time’ t_0, and the initial conditions may involve symbolic parameters. However, such parameters must evaluate to numerical objects, when the sequence returned by `numeric::ode2vectorfield` is passed to the numerical solver.

The vectorfield procedure `f` and the initial values Y_0 returned by `numeric::ode2vectorfield` can also be used by the functions `numeric::odesolve`, `plot::Ode2d`, `plot::Ode3d`. Cf. “Example 3” on page 19-241.

Examples

Example 1

We consider the initial value problem

$$y'(t) = t\sin(y(t)),\ y(0) = 1$$

.

The solver `numeric::odesolve2` requires the procedure `f:`$(t, Y) \to t\sin(Y_1)$ for the 1-dimensional vector $Y = [y(t)]$ specified by `fields`. The utility `numeric::ode2vectorfield` accepts a more convenient representation via arithmetical expressions:

```
IVP := {y'(t) = t*sin(y(t)), y(t0) = y0}:
fields := [y(t)]:
IVP := numeric::ode2vectorfield(IVP, fields)
```

```
proc(t, Y) ... end, t0, [y0]
```

This sequence may be passed to the numerical solver which returns a procedure representing the numerical solution:

```
t0 := 0: y0 := 1:
Y := numeric::odesolve2(IVP)
```

```
proc Y(t) ... end
```

Calling the numerical solution `Y` invokes the numerical integration from the intial 'time' $t_0 = 0$ to the 'time' specified in the call to `Y`:

```
Y(0), Y(1), Y(2), Y(3)
```

```
[1.0], [1.466404006], [2.655911348], [3.100928494]
```

```
delete IVP, fields, Y:
```

Example 2

We consider the second order initial value problem

$$y''(t) = t\sin(y(t)),\ y(0) = 1,\ y'(0) = 0$$

The corresponding vectorfield representation involves the vector $Y = [y(t), y'(t)]$ specified by `fields`:

```
IVP := {y''(t) = t*sin(y(t)), y(0) = 1, y'(0) = 0}:
fields := [y(t), y'(t)]:
numeric::ode2vectorfield(IVP, fields)
```

$$\text{proc(t, Y) ... end, 0, [1, 0]}$$

This sequence is accepted by `numeric::odesolve2`. The numerical solution `Y` returns lists representing the components of the vector specified by `fields`:

```
Y := numeric::odesolve2(%):
Y(5)
```

$$[1.916536393,\ -0.8765542819]$$

With a reordering of the unknown fields, the numerical solver returns the solution vector with rearranged components:

```
fields := [y'(t), y(t)]:
Y := numeric::odesolve2(numeric::ode2vectorfield(IVP, fields)):
Y(5)
```

$$[-0.8765542819,\ 1.916536393]$$

```
delete IVP, fields, Y:
```

Example 3

The following IVP involves the unknown fields $u(t)$, $v(t)$, $w(t)$. Since it is of second order in u, of first order in v and of third order in w, the list of unknowns `[y(t), y'(t), v(t), w(t), w'(t), w''(t)]` is appropriate:

```
IVP := {u''(t) - u(t)*v'(t) = exp(-t)*v'(t), v'(t) = w''(t),
        u'(t)*w'''(t) = t + u''(t),
```

```
        u(PI) = 3, u'(PI) = 1,
        v(PI) = 0,
        w(PI) = w'(PI), w'(PI) = 2 - w(PI), w''(PI) = 3*w(PI)}:
fields := [u(t), u'(t), v(t), w(t), w'(t), w''(t)]:
ivp := numeric::ode2vectorfield(IVP, fields)
```

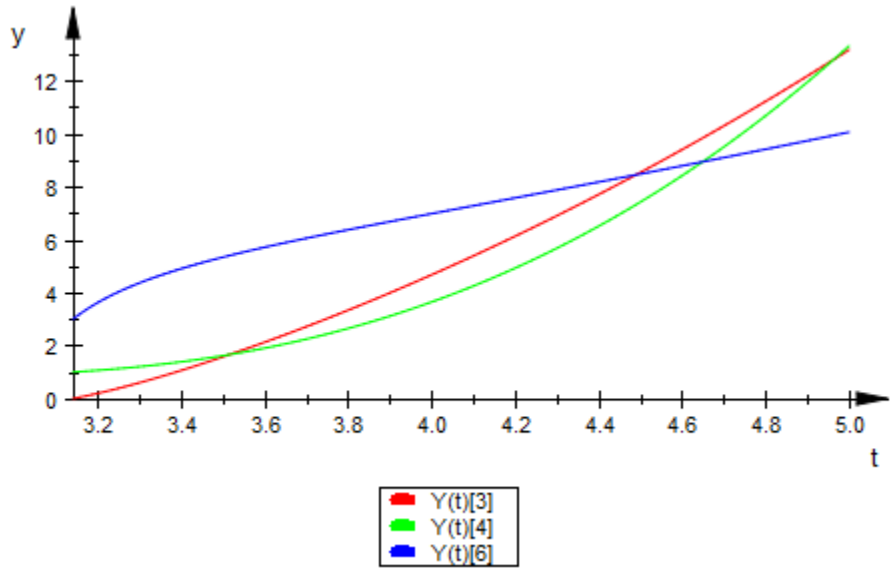
$$\text{proc(t, Y) ... end}, \pi, [3, 1, 0, 1, 1, 3]$$

```
Y := numeric::odesolve2(ivp):
Y(5)
```

$$[195.9501263, 604.3872242, 13.15053015, 13.29454726, 14.15053015, 10.04763196]$$

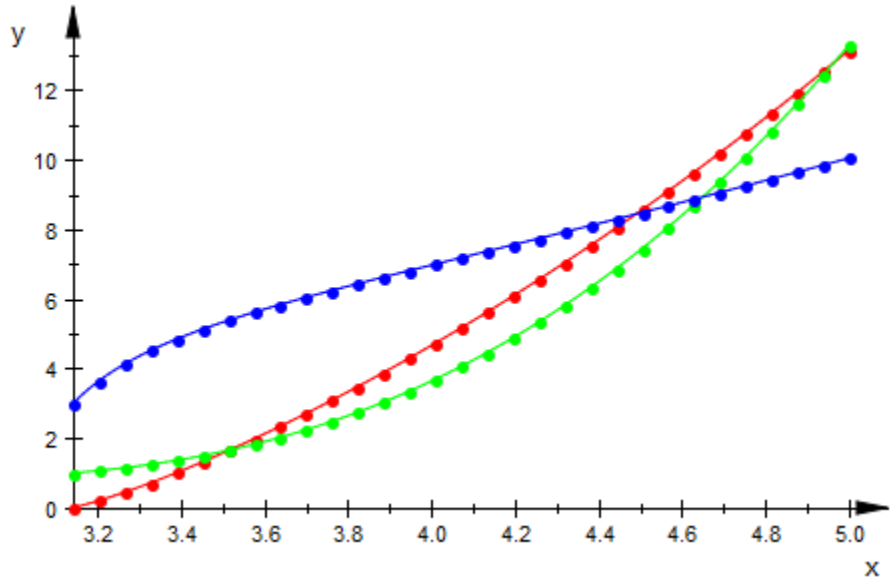
We plot the components v, w, and w'' of the solution vector:

```
plotfunc2d(Y(t)[3], Y(t)[4], Y(t)[6], t = PI .. 5,
           Colors = [RGB::Red, RGB::Green, RGB::Blue]):
```



Alternatively, we use the vectorfield procedure `ivp[1]` and the initial conditions `ivp[3]` as input parameters for `plot::Ode2d`:

```
plot(plot::Ode2d([PI + i*(5 - PI)/30 $ i = 0..30],
                 ivp[1], ivp[3],
                 [(t, Y) -> [t, Y[3]], Color = RGB::Red],
                 [(t, Y) -> [t, Y[4]], Color = RGB::Green],
                 [(t, Y) -> [t, Y[6]], Color = RGB::Blue])):
```




```
delete IVP, fields, ivp, Y:
```

Parameters

`IVP`

The initial value problem: a list or a set of equations involving univariate function calls $y_1(t)$, $y_2(t)$ etc. and derivates $y_1'(t)$, $y_1''(t)$, ..., $y_2'(t)$, $y_2''(t)$ etc. The differential equations must be quasi-linear: the highest derivative of each of the dependent functions $y_1(t)$, $y_2(t)$ etc. must enter the equations linearly. `IVP` must also contain corresponding initial conditions specified by linear equations in the expressions $y_1(t_0)$, $y_1'(t_0)$, ..., $y_2(t_0)$, $y_2'(t_0)$ etc. Alternatively, arithmetical expressions may be specified which are interpreted as equations with vanishing right hand side.

`fields`

The vector of the dynamical system equivalent to `IVP`: a list of symbolic function calls such as `[y_1(t), y_1'(t), dots, y_2(t), y_2'(t), dots]` representing the unknown fields to be solved for.

Return Values

Sequence `f, t0, Y0`. These data represent the dynamical system $\frac{dY}{dt} = f(t, Y)$ with the initial condition $Y(t_0) = Y_0$ equivalent to `IVP`. The vectorfield `f`: $(t, Y) \to f(t, Y)$ is a procedure, `t0` is a numerical expression representing the initial 'time', and `Y0` is a list of numerical expressions representing the components of the initial vector Y_0.

See Also

MuPAD Functions

`numeric::odesolve` | `numeric::odesolve2` | `numeric::odesolveGeometric` | `numeric::odeToVectorField`

MuPAD Graphical Primitives

`plot::Ode2d` | `plot::Ode3d`

More About

- "Solve Equations Numerically"

numeric::odeToVectorField

Convert an ode system to vectorfield notation

Syntax

```
numeric::odeToVectorField(IVP, fields)
```

Description

`numeric::odeToVectorField` converts a system of ordinary differential equations of arbitrary order to a vector field representation suitable for the numerical ODE solver `numeric::odesolve2`.

`numeric::odeToVectorField` and `numeric::ode2vectorfield` are equivalent.

`numeric::odeToVectorField` is a utility function to generate input parameters for the numerical ODE solver `numeric::odesolve2`. This solver requires a procedure representing the vectorfield $f(t, Y)$ of a first order system of differential equations (dynamic system) $\frac{dY}{dt} = f(t, Y)$ and initial data $Y_0 = Y(t_0)$. Given an initial value problem `IVP` consisting of (possibly higher order) differential expressions together with initial conditions, `numeric::odeToVectorField` converts the higher order equations to an equivalent system of first order ODEs and returns the input parameters for `numeric::odesolve2`.

Higher-order differential equations can always be represented as an equivalent dynamic system $\frac{dY}{dt} = f(t, Y)$ with some vector Y. E.g., the n-th order equation

$$y^{(n)} = g\left(t, y, y', \ldots, y^{(n-1)}\right)$$

may be written as the first order system

$$\frac{dY}{dt} = \left[y', \ldots, y^{(n-1)}, g\left(t, y, \ldots, y^{(n-1)}\right)\right]$$

for the vector $Y = \left[y, y', \ldots, y^{(n-1)}\right]$.

The input list `fields` correponds to the vector *Y*. It must be a *complete* specification of all functions and their derivatives through but *not including the highest derivates* of the unknown functions. E.g., for the second order differential equation $y''(t) = y(t)$, the appropriate list of unknown fields is $[y(t), y'(t)]$. The differential equations $y''(t) = z(t)$, $z'(t) = y(t)$ are of second order in *y* and of first order in *z*. Hence, the appropriate list of unknown fields is `[y(t), y'(t), z(t)]`.

The ordering of the fields in `list` determines the ordering of the components of the list that the numerical solver produces as the solution vector. Cf. “Example 2” on page 19-247.

The differential equations must be linear in the highest derivatives of the unknown functions involved. E.g., the ODE $y'(t)^2 = y(t)$ is not admitted. However, equations such as $y_2(t)\, y_1'(t) = y_1(t)$, $y_1(t)\, y_2'(t) = y_2(t)$ are accepted and converted to $y_1'(t) = \frac{y_1(t)}{y_2(t)}$, $y_2'(t) = \frac{y_2(t)}{y_1(t)}$.

A *complete* specification of initial conditions must be contained in `IVP`: for each component in `list`, an initial value must be provided. The initial conditions may be specified by linear equations which will be solved for the initial values of the unknown fields automatically. E.g., for `fields = [y(t), y'(t), z(t)]`, initial conditions may be specified explicitly by $y(t_0) = 1$, `y'(t_0) = 2`, $z(t_0) = 3$, say, or via linear equations such as `y(t_0) + z(t_0) = y'(t_0)`, $y(t_0) = z(t_0)$, $z(t_0) = 2\, y(t_0)$. Cf. “Example 3” on page 19-248.

The differential equations, the initial ‘time’ t_0, and the initial conditions may involve symbolic parameters. However, such parameters must evaluate to numerical objects, when the sequence returned by `numeric::odeToVectorField` is passed to the numerical solver.

The vectorfield procedure `f` and the initial values Y_0 returned by `numeric::odeToVectorField` can also be used by the functions `numeric::odesolve`, `plot::Ode2d`, `plot::Ode3d`. Cf. “Example 3” on page 19-248.

Examples

Example 1

We consider the initial value problem

$$y'(t) = t\sin(y(t)),\ y(0) = 1$$

.

The solver `numeric::odesolve2` requires the procedure `f:`$(t, Y) \to t\sin(Y_1)$ for the 1-dimensional vector $Y = [y(t)]$ specified by `fields`. The utility `numeric::ode2vectorfield` accepts a more convenient representation via arithmetical expressions:

```
IVP := {y'(t) = t*sin(y(t)), y(t0) = y0}:
fields := [y(t)]:
IVP := numeric::ode2vectorfield(IVP, fields)
```

```
proc(t, Y) ... end, t0, [y0]
```

This sequence may be passed to the numerical solver which returns a procedure representing the numerical solution:

```
t0 := 0: y0 := 1:
Y := numeric::odesolve2(IVP)
```

```
proc Y(t) ... end
```

Calling the numerical solution `Y` invokes the numerical integration from the intial 'time' $t_0 = 0$ to the 'time' specified in the call to `Y`:

```
Y(0), Y(1), Y(2), Y(3)
```

```
[1.0], [1.466404006], [2.655911348], [3.100928494]
```

```
delete IVP, fields, Y:
```

Example 2

We consider the second order initial value problem

$$y''(t) = t\sin(y(t)),\ y(0) = 1,\ y'(0) = 0$$

The corresponding vectorfield representation involves the vector $Y = [y(t), y'(t)]$ specified by `fields`:

```
IVP := {y''(t) = t*sin(y(t)), y(0) = 1, y'(0) = 0}:
fields := [y(t), y'(t)]:
numeric::ode2vectorfield(IVP, fields)
```

```
proc(t, Y) ... end, 0, [1, 0]
```

This sequence is accepted by `numeric::odesolve2`. The numerical solution `Y` returns lists representing the components of the vector specified by `fields`:

```
Y := numeric::odesolve2(%):
Y(5)
```

```
[1.916536393, -0.8765542819]
```

With a reordering of the unknown fields, the numerical solver returns the solution vector with rearranged components:

```
fields := [y'(t), y(t)]:
Y := numeric::odesolve2(numeric::ode2vectorfield(IVP, fields)):
Y(5)
```

```
[-0.8765542819, 1.916536393]
```

```
delete IVP, fields, Y:
```

Example 3

The following IVP involves the unknown fields $u(t)$, $v(t)$, $w(t)$. Since it is of second order in u, of first order in v and of third order in w, the list of unknowns `[y(t), y'(t), v(t), w(t), w'(t), w''(t)]` is appropriate:

```
IVP := {u''(t) - u(t)*v'(t) = exp(-t)*v'(t), v'(t) = w''(t),
        u'(t)*w'''(t) = t + u''(t),
```

```
        u(PI) = 3, u'(PI) = 1,
        v(PI) = 0,
        w(PI) = w'(PI), w'(PI) = 2 - w(PI), w''(PI) = 3*w(PI)}:
fields := [u(t), u'(t), v(t), w(t), w'(t), w''(t)]:
ivp := numeric::ode2vectorfield(IVP, fields)
```

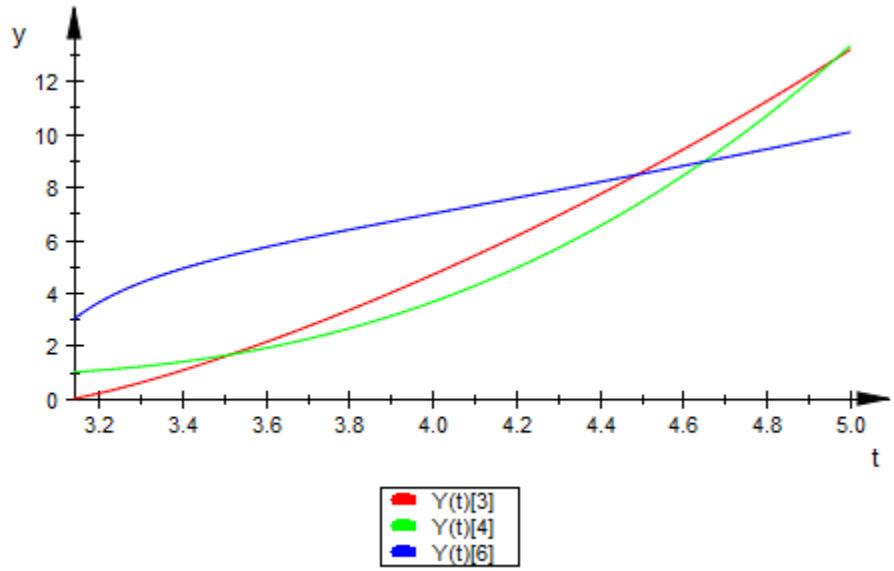
$$\text{proc(t, Y) ... end}, \pi, [3, 1, 0, 1, 1, 3]$$

```
Y := numeric::odesolve2(ivp):
Y(5)
```

$$[195.9501263, 604.3872242, 13.15053015, 13.29454726, 14.15053015, 10.04763196]$$

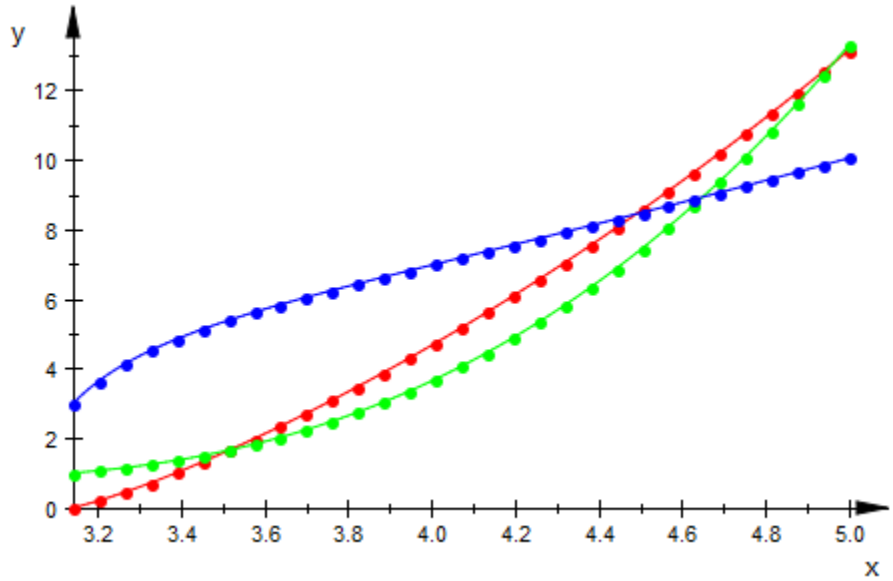
We plot the components v, w, and w''' of the solution vector:

```
plotfunc2d(Y(t)[3], Y(t)[4], Y(t)[6], t = PI .. 5,
           Colors = [RGB::Red, RGB::Green, RGB::Blue]):
```



Alternatively, we use the vectorfield procedure `ivp[1]` and the initial conditions `ivp[3]` as input parameters for `plot::Ode2d`:

```
plot(plot::Ode2d([PI + i*(5 - PI)/30 $ i = 0..30],
                 ivp[1], ivp[3],
                 [(t, Y) -> [t, Y[3]], Color = RGB::Red],
                 [(t, Y) -> [t, Y[4]], Color = RGB::Green],
                 [(t, Y) -> [t, Y[6]], Color = RGB::Blue])):
```




```
delete IVP, fields, ivp, Y:
```

Parameters

IVP

The initial value problem: a list or a set of equations involving univariate function calls $y_1(t)$, $y_2(t)$ etc. and derivates $y_1'(t)$, $y_1''(t)$, ..., $y_2'(t)$, $y_2''(t)$ etc. The differential equations must be quasi-linear: the highest derivative of each of the dependent functions $y_1(t)$, $y_2(t)$ etc. must enter the equations linearly. `IVP` must also contain corresponding initial conditions specified by linear equations in the expressions $y_1(t_0)$, $y_1'(t_0)$, ..., $y_2(t_0)$, $y_2'(t_0)$ etc. Alternatively, arithmetical expressions may be specified which are interpreted as equations with vanishing right hand side.

`fields`

The vector of the dynamical system equivalent to `IVP`: a list of symbolic function calls such as `[y_1(t), y_1'(t), dots, y_2(t), y_2'(t), dots]` representing the unknown fields to be solved for.

Return Values

Sequence `f, t0, Y0`. These data represent the dynamical system $\frac{dY}{dt} = f(t, Y)$ with the initial condition $Y(t_0) = Y_0$ equivalent to `IVP`. The vectorfield `f`: $(t, Y) \to f(t, Y)$ is a procedure, `t0` is a numerical expression representing the initial 'time', and `Y0` is a list of numerical expressions representing the components of the initial vector Y_0.

See Also

MuPAD Functions

`numeric::ode2vectorfield` | `numeric::odesolve` | `numeric::odesolve2` | `numeric::odesolveGeometric`

MuPAD Graphical Primitives

`plot::Ode2d` | `plot::Ode3d`

More About

- "Solve Equations Numerically"

numeric::polyrootbound

Bound for the roots of a univariate polynomial

Syntax

```
numeric::polyrootbound(p)
```

Description

`numeric::polyrootbound(p)` returns a bound b, such that all real and complex roots z of the univariate polynomial p satisfy $|z| \le b$.

The coefficients of `p` may be real or complex numbers. Also exact numerical coefficients such as π, $\sqrt{2}$ etc. are accepted if they can be converted to floats.

For non-zero constant polynomials, `numeric::polyrootbound` returns `infinity`.

For monomials $p(x) = c_n x^n$ with $n > 0$, `numeric::polyrootbound` returns `0.0`.

Consider the polynomial $p(z) = z^n + c_{n-1} z^{n-1} + \cdots + c_0$. If $max(|c_{n-1}|, \ldots, |c_0|) > 0$, the polynomial

$$z^n - |c_{n-1}| z^{n-1} - \cdots - |c_0|$$

has a single real root $b > 0$ which is an upper bound for the absolute values of all real and complex roots of p. The bound returned by `numeric::polyrootbound(p)` approximates b to about 3 leading decimal digits.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

Both polynomial expressions as well as DOM_POLY objects may be used to specify the polynomial:

```
p := x^3 + PI*x - sqrt(2): numeric::polyrootbound(p)
```

$$1.966316026$$

```
p := poly(p, [x]): numeric::polyrootbound(p)
```

$$1.966316026$$

The absolute values of all real and complex roots of p are bounded by this number:

```
numeric::polyroots(p)
```

$$[0.4256164232,\ -0.2128082116 + 1.810374176\,i,\ -0.2128082116 - 1.810374176\,i]$$

```
max(abs(z) $ z in %)
```

$$1.822838993$$

```
delete p:
```

Parameters

p

A univariate polynomial expression or a univariate polynomial of domain type DOM_POLY.

Return Values

Nonnegative real floating-point number or `infinity`.

See Also

MuPAD Functions

`numeric::fsolve` | `numeric::polyroots` | `numeric::polysysroots` | `numeric::realroot` | `numeric::realroots` | `polylib::realroots` | `RootOf` | `solve`

numeric::polyroots

Numerical roots of a univariate polynomial

Syntax

```
numeric::polyroots(eqs, <FixedPrecision>, <SquareFree>, <Factor>, <NoWarning>)
```

Description

`numeric::polyroots(eqs)` returns numerical approximations of all real and complex roots of the univariate polynomials `eqs`.

The coefficients may be real or complex numbers. Also symbolic coefficients are accepted if they can be converted to floats.

The trivial polynomial *eqs* = 0 results in an error message. The empty list is returned for constant polynomials *eqs* ≠ 0.

Multiple roots are listed according to their multiplicities, i.e., the length of the root list coincides with the degree of *eqs*.

The root list is sorted by `numeric::sort`.

Up to roundoff effects, the numerical roots should be accurate to `DIGITS` significant digits, unless the option `FixedPrecision` is used.

All floating-point entries in `eqs` are internally approximated by rational numbers: `numeric::polyroots(eqs)` computes the roots of `numeric::rationalize(eqs, Minimize)`.

For polynomial expressions in factored form, the numerical search is applied to each factor separately.

It is recommended to use `numeric::realroots` or `polylib::realroots` if `eqs` is a real polynomial and only real roots are of interest.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

Both polynomial expressions as well as `DOM_POLY` objects may be used to specify the polynomial:

```
numeric::polyroots(x^3 - 3*x - sqrt(2))
```

$$[1.931851653,\ -0.5176380902,\ -1.414213562]$$

```
numeric::polyroots(PI*z^4 + I*z + 0.1)
```

$$[0.5936837297 - 0.3729248918\ i,\ 0.6455316068\ i,\ 0.1003181767\ i,\ -0.5936837297 - 0.3729248918\ i]$$

```
numeric::polyroots(poly(x^5 - x^2, [x]))
```

$$[1.0,\ 0.0,\ 0.0,\ -0.5 + 0.8660254038\ i,\ -0.5 - 0.8660254038\ i]$$

Example 2

The following polynomial has exact coefficients:

```
p := poly((x - 1)*(x - PI)^3, [x]):
numeric::polyroots(p)
```

$$[3.141592654,\ 3.141592654,\ 3.141592654,\ 1.0]$$

Note that roundoff errors in the coefficients of `eqs` have a dramatic effect on multiple roots:

```
p := poly((x - 1.0)*(x - float(PI))^3, [x]):
numeric::polyroots(p)
```

$$[3.144422386,\ 3.140177788 - 0.00244957836\,i,\ 3.140177788 + 0.00244957836\,i,\ 0.9999999995]$$

These are the roots of the following rationalized polynomial:

```
numeric::rationalize(p, Minimize)
```

$$\mathrm{poly}\left(x^4 - \frac{461286\,x^3}{44249} + \frac{176627\,x^2}{4525} - \frac{2515405\,x}{41498} + \frac{1837649}{59267},\ [x]\right)$$

```
delete p:
```

Example 3

The multiple root $\frac{i}{3}$ of the following polynomial can only be computed with restricted precision by fixed precision arithmetic:

```
p := poly((x^2 - 6*x +8)*(x - I/3)^5, [x]):
numeric::polyroots(p, FixedPrecision)
```

$$[4.0,\ 2.0,\ 0.006617926628 + 0.330782186\,i,\ 0.004459639998 + 0.3388505458\,i,\ -0.0003838250804 + 0.3262671926\,i,\ -0.003866386513 + 0.3392634968\,i,\ -0.006827355033 + 0.3315032456\,i]$$

Without the option `FixedPrecision`, the working precision is increased internally to compute better approximations:

```
numeric::polyroots(p)
```

$$[4.0,\ 2.0,\ 0.3333333333\,i,\ 0.3333333333\,i,\ 0.3333333333\,i,\ 0.3333333333\,i,\ 0.3333333333\,i]$$

```
delete p:
```

Example 4

The following polynomial has badly separated roots. `numeric::polyroots` does not manage to separate them properly:

```
p := poly(_mult((x - 1 - i/10^9) $ i=0..5), [x]):
numeric::polyroots(p)
```

```
[1.000000003, 1.000000003, 1.000000003, 1.000000003, 1.000000003, 0.9999999987]
```

One can preprocess the polynomial by a symbolic factorization:

```
numeric::polyroots(p, Factor)
```

```
[1.000000005, 1.000000004, 1.000000003, 1.000000002, 1.000000001, 1.0]
```

Alternatively, one can increase the working precision to separate the roots:

```
DIGITS := 20:
numeric::polyroots(p)
```

```
[1.000000005, 1.000000004, 1.000000003, 1.000000002, 1.000000001, 1.0]
```

```
delete p, DIGITS:
```

Parameters

eqs

A univariate polynomial expression or a univariate polynomial of domain type `DOM_POLY`. The function also accepts a list, set, `array`, or `matrix` (`Cat::Matrix`) of polynomial expressions.

Options

FixedPrecision

This option provides the fastest way to obtain approximations of the roots by a numerical search with a fixed internal precision of 2 *DIGITS* decimal places.

Note that badly isolated roots or multiple roots will usually not be approximated to `DIGITS` decimals when using this option. The problem of finding such roots is

numerically ill-conditioned, i.e., such roots cannot be found to full precision with fixed precision arithmetic. Typically, a q-fold root will be approximated only to about $\frac{2\,\mathrm{DIGITS}}{q}$ decimal places. Cf. “Example 3” on page 19-257.

Without this option, `numeric::polyroots` internally increases the working precision until all roots are found to `DIGITS` decimal places.

`SquareFree`

With this option, a symbolic square free factorization is computed via `polylib::sqrfree(eqs)`. The numerical root finding algorithm is then applied to each square free factor.

This option is recommended, when `p` is known to have multiple roots. Such roots force `numeric::polyroots` to increase the working precision internally increasing the costs of the numerical search. A square free factorization reduces the multiplicity of each root to one, speeding up the final numerical search.

For polynomials with real rational coefficients, a square free factorization is always used, i.e., this option does not have any effect for such polynomials. For all other types of coefficients, a square free factorization may be costly and must be requested by this option.

Multiple roots of `eqs` can be successfully dealt with by this option. However, for badly separated distinct roots the square free factorization will not improve the performance of the numerical search.

`Factor`

With this option, a symbolic factorizations of `eqs` via `factor` are computed. The numerical root finding algorithm is then applied to each factor.

This option is useful, when `eqs` can be successfully factorized (e.g., when each expression from `eqs` has multiple roots). The numerical search on the factors is much more efficient than the search on the original polynomial. On the other hand, symbolic factorization of `eqs` may be costly.

`NoWarning`

Suppresses warnings

Return Values

List of numerical roots.

Algorithms

The numerical root finding algorithm implemented by `numeric::polyroots` is Laguerre's method: W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling: *Numerical Recipes in C*, Cambridge University Press, 1988.

See Also

MuPAD Functions

`numeric::fsolve` | `numeric::polysysroots` | `numeric::realroot` | `numeric::realroots` | `polylib::realroots` | `RootOf` | `solve`

numeric::polysysroots

Numerical roots of a system of polynomial equations

Syntax

```
numeric::polysysroots(eqs, <NoWarning>)
```

```
numeric::polysysroots(eqs, vars, <NoWarning>)
```

Description

`numeric::polysysroots(eqs, ...)` returns numerical approximations of all real and complex roots of the polynomial system of equations `eqs`.

The coefficients of the polynomials may contain symbolic parameters.

If no unknowns are specified by `vars`, then`numeric::indets(eqs)` is used in place of `vars`.

In most cases, the solution is returned as a set of lists of solved equations of the form

$$[x_1 = \text{value}_1, x_2 = \text{value}_2, \ldots],$$

where x_1, x_2, … are the unknowns. These simplified equations should be regarded as constraints on the unknowns. E.g., if an unknown x_1, say, does not turn up in the form x_1 = … in the solution, then there is no constraint on this unknown and it is an arbitrary parameter. This holds true in general for all unknowns that do not turn up on the left hand side of the solved equations. Cf. “Example 2” on page 19-263.

If no explicit solutions can be computed, expressions of the form $\begin{pmatrix} x_1 \\ x_2 \\ \ldots \end{pmatrix} \in S$ may be returned, where S is the solution set.

The ordering of the unknowns in `vars` determines the ordering of the solved equations. If a *set*`vars` is used, then an internal ordering is used.

Note: If the solution set of `eqs` is not finite, then `numeric::polysysroots` may return solutions with some of the unknowns remaining as free parameters. In this case the representation of the solution depends on the ordering of the unknowns! Cf. "Example 3" on page 19-263. Further, if higher degree polynomials are involved, then `numeric::polysysroots` may fail to compute roots. Cf. "Example 5" on page 19-264. The same may happen, when `eqs` contains symbolic parameters.

You may try `numeric::fsolve` to compute a single numerical root, if `numeric::polysysroots` cannot compute all roots of the system. Note, however, that `numeric::fsolve` does not accept symbolic parameters in the equations.

We recommend to use `numeric::polyroots` to compute all roots of a single univariate polynomial with numerical coefficients.

`numeric::polysysroots` is a hybrid routine: it calls the symbolic solver `solve(eqs, vars, BackSubstitution = FALSE)` and processes its symbolic result numerically. Cf. "Example 4" on page 19-264.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

Equations, expressions as well as `DOM_POLY` objects may be used to specify the polynomials:

```
numeric::polysysroots(x^2 = PI^2, x)
```

$$\{[x = 3.141592654], [x = -3.141592654]\}$$

```
numeric::polysysroots({x^2 + y^2 - 1, x^2 - y^2 = 1/2}, [x, y])
```

$$\{[x = 0.8660254038,\ y = 0.5],\ [x = 0.8660254038,\ y = -0.5],\ [x = -0.8660254038,\ y = -0.5],\ [x = -0.8660254038,\ y = 0.5]\}$$

```
numeric::polysysroots({poly(x^2 + y + 1), y^2 + x = 1}, [x, y])
```

$$\{[x = -0.4533976515,\ y = -1.20556943],\ [x = 0.2266988258 + 1.467711509\,i,\ y = 1.102784715 - 0.6654569512\,i],\ [x = 0,\ y = -1.0],\ [x = 0.2266988258 - 1.467711509\,i,\ y = 1.102784715 + 0.6654569512\,i]\}$$

Symbolic parameters are accepted:

```
numeric::polysysroots(x^2 + y + exp(z), [x, y])
```

$$\left\{\left[x = \sqrt{-1.0\,y - 1.0\,e^z}\right],\ \left[x = -1.0\,\sqrt{-1.0\,y - 1.0\,e^z}\right]\right\}$$

Example 2

The returned solutions may contain some of the unknowns remaining as free parameters:

```
numeric::polysysroots({x^2 + y^2 = z}, [x, y, z])
```

$$\left\{\left[x = \sqrt{z - 1.0\,y^2}\right],\ \left[x = -1.0\,\sqrt{z - 1.0\,y^2}\right]\right\}$$

Example 3

The ordering of the unknowns determines the representation of the solution, if the solution set is not finite. First, the following equation is solved for `x` leaving `y` as a free parameter:

```
numeric::polysysroots({x^3 = y^2}, [x, y])
```

$$\left\{\left[x = (y^2)^{1/3}\right],\ \left[x = (y^2)^{1/3}\,(-0.5 + 0.8660254038\,i)\right],\ \left[x = (y^2)^{1/3}\,(-0.5 - 0.8660254038\,i)\right]\right\}$$

Reordering the unknowns leads to a representation with `x` as a free parameter:

```
numeric::polysysroots({x^3 = y^2}, [y, x])
```

$$\{[y = \sqrt{x^3}],\ [y = -1.0\sqrt{x^3}]\}$$

Example 4

The symbolic solver produces a `RootOf` solution of the following system:

```
eqs := {y^2 - y = x, x^3 = y^3 + x}:
solve(eqs, BackSubstitution = FALSE)
```

$$\begin{pmatrix} x \\ y \end{pmatrix} \in \left\{\begin{pmatrix} 0 \\ 0 \end{pmatrix}\right\} \cup \left\{\begin{pmatrix} y^2 - y \\ z1 \end{pmatrix} \;\middle|\; z1 \in \mathrm{RootOf}(z^5 - 3z^4 + 3z^3 - 2z^2 - z + 1,\ z)\right\}$$

Internally, `numeric::polysysroots` calls `solve` and processes this result numerically:

```
numeric::polysysroots(eqs, [x, y])
```

$$\{[x = 2.237302267,\ y = 2.077118343],\ [x = -1.445049623 - 0.1279930535\,i,\ y = 0.441542078 + 1.094745154\,i],\ [x = 0.8911259621,\ y = -0.5682349751],\ [x = -0.2383289841,\ y = 0.6080324763],\ [x = -1.445049623 + 0.1279930535\,i,\ y = 0.441542078 - 1.094745154\,i],\ [x = 0,\ y = 0]\}$$

```
delete eqs:
```

Example 5

The following equation is solved for the first of the specified unknowns:

```
eqs := y^5 - PI*y = x:
solve(eqs, [x, y])
```

$$\{[x = z^5 - \pi z,\ y = z]\}$$

`numeric::polysysroots` processes this output numerically:

```
numeric::polysysroots(eqs, [x, y])
```

$$\{[x = y^5 - 3.141592654\,y]\}$$

The equation is solved for `y` when the unknowns are reordered. However, no simple representation of the solution exists, so a `RootOf` object is returned:

```
solve(eqs, [y, x])
```

$$\begin{pmatrix} y \\ x \end{pmatrix} \in \bigcup_{z2 \in \mathbb{C}} \left\{ \begin{pmatrix} z1 \\ z2 \end{pmatrix} \;\middle|\; z1 \in \mathrm{RootOf}(z^5 - \pi\, z - z2,\, z) \right\}$$

The roots represented by the `RootOf` expression cannot be computed numerically, because the symbolic parameter `x` is involved:

```
numeric::polysysroots(eqs, [y, x])
```

$$\begin{pmatrix} y \\ x \end{pmatrix} \in \bigcup_{z2 \in \mathbb{C}} \left\{ \begin{pmatrix} z1 \\ z2 \end{pmatrix} \;\middle|\; z1 \in \mathrm{RootOf}(z^5 - \pi\, z - z2,\, z) \right\}$$

```
delete eqs:
```

Parameters

`eqs`

A polynomial equation or a list, set, `array`, or `matrix` (`Cat::Matrix`) of such equations. Also expressions and polynomials of domain type `DOM_POLY` are accepted wherever an equation is expected. They are interpreted as homogeneous equations.

`vars`

An unknown or a list or set of unknowns. Unknowns may be identifiers or indexed identifiers.

Options

`NoWarning`

By default, the roots are double-checked, automatically. Warnings are issued if a solution seems to be marred by some numerical instability. With this option, this check is suppressed and no warnings will be issued.

Return Values

a set of lists of equations or an expression of the form $\begin{pmatrix} x_1 \\ x_2 \\ \dots \end{pmatrix} \in S$, where x_1, x_2, … are the unknowns and S is the solution set.

The set {[]} containing an empty list is returned, if no solutions can be computed.

See Also

MuPAD Functions

`linsolve` | `numeric::fsolve` | `numeric::linsolve` | `numeric::polyroots` | `numeric::realroot` | `numeric::realroots` | `polylib::realroots` | `solve`

numeric::product

Numerical approximation of products

Syntax

```
numeric::product(f, i = a .. b)
numeric::product(f, i in RootOf(p, <x>))
numeric::product(f, i = RootOf(p, <x>))
numeric::product(f, i in {x1, x2, …})
numeric::product(f, i = {x1, x2, …})
```

Description

`numeric::product(f, i = a..b)` computes a numerical approximation of $\prod_{i=a}^{b} f(i)$.

`numeric::product (f, i = RootOf(p,x))` computes a numerical approximation of the product of `f` over the roots of the polynomial `p`.

`numeric::product(f, i in { x1, x2, …})` computes a numerical approximation of $\prod_{i \in \{x_1, x_2, \ldots\}} f(i)$.

The call `numeric::product(...)` is equivalent to calling the `float` attribute of `product` via `float ( hold( product )(...))`, `float ( freeze( product )(...))` or `product::float(...)`.

If there are other symbolic parameters in *f*, apart from the variable *i*, a symbolic call to `numeric::product` is returned. Numerical expressions such as e^{π}, $\sqrt{2}$ etc. are accepted and converted to floating-point numbers.

Note: For finite products, `numeric::product` just returns `_mult ( float(f) $ i=a..b)`. Cf. “Example 3” on page 19-269.

The call `numeric::product(f, i = { x1, x2, …})` computes numerical approximations of `x[1]`, `x[2]` etc., substitutes these values into *f*(*i*) and multiplies the results.

The calls `numeric::product(f, i in { x1, x2, …})` and `numeric::product(f, i = { x1, x2, …})` are equivalent.

The call `numeric::product (f, i in RootOf(p, x))` computes numerical approximations of all roots of *p*, substitutes these values into *f*(*i*) and multiplies the results. Cf. “Example 2” on page 19-269.

The calls `numeric::product(f, i in RootOf(p, x))` and `numeric::product(f, i = RootOf(p, x))` are equivalent.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision and influences the maximum number of steps used in the computation.

Examples

Example 1

We demonstrate some equivalent calls for numerical products:

```
numeric::product(1+1/k^2, k = 1..infinity),
float(hold(product)(1+1/k^2, k = 1..infinity)),
float(freeze(product)(1+1/k^2, k = 1..infinity)),
product::float(1+1/k^2, k = 1..infinity);
```

```
3.67607791, 3.67607791, 3.67607791, 3.67607791
```

`product` fails to find a closed form for the following product:

```
product(1 - 1/4^k, k = 1..infinity);
```

$$\prod_{k=1}^{\infty} \frac{2^{2k}-1}{2^{2k}}$$

`float` implicitly uses `numeric::product` to compute a numerical approximation:

```
float(%);
```

$$0.6885375371$$

The exact value of the following infinite product is $e^{(1-\sqrt{2})\,\zeta\left(\frac{1}{2}\right)}$:

```
numeric::product(exp((-1)^(k+1)*k^(-1/2)), k = 1..infinity)
 = float(exp((1-sqrt(2))*zeta(1/2)))
```

$$1.831066609 = 1.831066609$$

Example 2

We calculate an approximation of the product over the roots of a polynomial:

```
numeric::product(sin(r), r = RootOf(x^2 - PI^2/4, x))
```

$$-1.0$$

If the polynomial expression contains additional indeterminates, a symbolic call to `numeric::product` is returned:

```
numeric::product(r+PI, r = RootOf(x^8 + c*x - PI^2/4, x))
```

$$\text{numeric::product}(r + 3.141592654,\ r = \text{RootOf}(x^8 + c\,x - 2.4674011,\ x))$$

Example 3

`numeric::product` can also be used to compute finite products:

```
numeric::product(1-1/k^2, k = 2..10^n) $ n in { 2, 3, 4 }
```

$$0.505,\ 0.5005,\ 0.50005$$

However, since `numeric::product` uses `_mult` internally anyway, it is more efficient to call `_mult` directly:

```
_mult(float(1-1/k^2) $ k = 2..10^n) $ n in { 2, 3, 4 }
```

$$0.505,\ 0.5005,\ 0.50005$$

Example 4

The following product is returned symbolically because it contains the additional indeterminate `k`:

```
numeric::product(1-1/n^k, n = 2..infinity)
```

$$\text{numeric::product}\left(1.0 - \frac{1.0}{n^{1.0\,k}},\ n = 2..\infty\right)$$

Parameters

`f`

An arithmetical expression depending on `i`

`i`

The product index: an identifier or indexed indentifier

`a, b`

integers or $\pm\infty$ satisfying $a \le b$

`p`

A univariate polynomial expression in `x`

x

Indeterminate

x_1, x_2, …

numerical expressions

Return Values

floating-point number or a symbolic expression of type `numeric::product`.

Algorithms

Infinite products are calculated by summing the series $\sum_{i=a}^{\infty} \ln(f)$ via `numeric::sum`.

`numeric::product` uses `numeric::polyroots` to calculate numerical approximations to the roots of a polynomials.

See Also

MuPAD Functions

`_mult` | `numeric::polyroots` | `numeric::sum` | `product`

numeric::quadrature

Numerical integration (Quadrature)

Syntax

```
numeric::quadrature(f(x), x = a .. b, <GaussLegendre = n | GaussTschebyscheff = n | Nev
```

Description

`numeric::quadrature(f(x), x = a..b)` computes a numerical approximation of $\int_a^b f(x)\,\mathrm{d}x$.

`numeric::quadrature` returns itself symbolically if the integrand `f(x)` contains symbolic objects apart from the integration variable `x` that cannot be converted to numerical values via `float`. Symbolic objects such as π or $\sqrt{2}$ etc. are accepted.

The integrand `f(x)` should be integrable in the Riemann sense. In particular, `f(x)` should be bounded on the integration interval `x = a..b`. Certain types of mild singularities are handled, but numerical convergence is not guaranteed and will be slow in most cases. Also discontinuities and singularities of (higher) derivatives of `f(x)` slow down numerical convergence. For integrands with irregular points, it is recommended to split the integration into several parts, using subintervals on which the integrand is smooth. Cf. “Example 4” on page 19-276.

Integrands given by user-defined procedures can be handled. See “Example 4” on page 19-276 and “Example 5” on page 19-277.

`numeric::quadrature` returns itself symbolically if the boundaries `a,b` contain symbolic objects that cannot be converted to numerical values via `float`. Symbolic objects such as π or $\sqrt{2}$ etc. as well as `infinity` and `-infinity` are accepted.

Note: For infinite ranges, the user should make sure that the integral exists! If `f(x)` does not decay as fast as $O\left(\frac{1}{|x|^2}\right)$ at infinity, then convergence may be slow.

Boundaries $a > b$ are accepted, using $\int_a^b f(x)\,\mathrm{d}x = -\int_b^a f(x)\,\mathrm{d}x$.

For complex values of `a,b`, the integration is to be understood as a contour integral along a straight line from `a` to `b`. Complex boundaries must not involve `infinity`.

Multi-dimensional integration such as

```
numeric::quadrature ( numeric::quadrature(f(x,y), y = A(x)..B(x)), x
= a..b)
```

is possible. See “Example 3” on page 19-275 and “Example 5” on page 19-277.

Internally, an adaptive mechanism based on a fixed quadrature rule specified by `method = n` is used. It tries to keep the relative quadrature error of the result below $\frac{1}{10^{\mathrm{DIGITS}}}$.

The last digit(s) of the result may be incorrect due to roundoff effects.

The routine `numeric::quadrature` is purely numerical: no symbolic check for singularities etc. is carried out.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We demonstrate the standard calls for adaptive numerical integration:

```
numeric::quadrature(exp(x^2), x = -1..1)
```

```
     2.925303492
```

```
numeric::quadrature(max(1/10, cos(PI*x)), x = -2..0.0123)
```

```
                0.752102471
```

```
numeric::quadrature(exp(-x^2), x = -2..infinity)
```

```
                1.768308316
```

The precision goal is set by `DIGITS`:

```
DIGITS := 50:
numeric::quadrature(besselJ(0, x), x = 0..PI)
```

```
                1.3475263146739901712314731279612149642205400522774
```

Note that due to the internal adaptive mechanism, the choice of different methods should not have any significant effect on the quadrature result:

```
DIGITS := 10:
numeric::quadrature(sin(x)/x, x = -1..10, GaussLegendre = 5),
numeric::quadrature(sin(x)/x, x = -1..10, GaussLegendre = 160),
numeric::quadrature(sin(x)/x, x = -1..10, NewtonCotes = 8)
```

```
                2.604430665, 2.604430665, 2.604430665
```

Example 2

The user should make sure that the integrand is well defined and sufficiently regular. The following fails, because Newton-Cotes quadrature tries to evaluate the integrand at the boundaries:

```
numeric::quadrature(sin(x)/x, x = 0..1, NewtonCotes = 8)
```

```
Error: Division by zero. [_power]
  Evaluating: Quadsum
```

One may cure this problem be assigning a value to `f(0)`. The integrand is passed to the integrator as `hold(f)` to prevent premature evaluation of `f(x)` to `sin(x)/x`. Internally, `numeric::quadrature` replaces `x` by numerical values and then evaluates the integrand. Note that one has to define `f(0.0) := 1` rather than `f(0) := 1`:

```
f := x -> sin(x)/x:
f(0.0) := 1:
numeric::quadrature(hold(f)(x), x = 0..1, NewtonCotes = 8)
```

$$0.9460830704$$

The default method (Gauss-Legendre quadrature) does not evaluate the integrand at the end points:

```
numeric::quadrature(sin(x)/x, x = 0..1)
```

$$0.9460830704$$

Nevertheless, problems may still arise for improper integrals:

```
numeric::quadrature(ln((1 + x^4)^2 - 1), x = 0 .. 1)
```

```
Warning: Precision goal is not achieved after 10000 function calls. Increase 'MaxCalls
```

$$-3.213735532$$

In this example, the integrand is evaluated close to 0. The expression $(1 + x^4)^2 - 1$ suffers from severe numerical cancellation and is dominated by round-off. A numerically stable representation is $(1 + x^4)^2 - 1 = x^4 (x^4 + 2)$:

```
numeric::quadrature(ln(x^4*(x^4 + 2)), x = 0..1)
```

$$-3.218234378$$

Note that convergence is rather slow, because the integrand is not bounded.

```
delete f:
```

Example 3

We demonstrate multi-dimensional quadrature:

```
Q := numeric::quadrature:
Q(Q(exp(x*y), x = 0..y), y = 0..1)
```

```
0.6589510757
```

Also more complex types of nested calls are possible. We use numerically defined functions

```
b := y -> Q(exp(-t^2-t*y), t = y..infinity):
```

and

```
f := y -> cos(y^2) +  Q(exp(x*y), x = 0..b(y)):
```

for the following quadrature:

```
Q(f(y), y = 0..1)
```

```
1.261578952
```

Multi dimensional quadrature is computationally expensive. Note that a call to `numeric::quadrature` evaluates the integrand at least $3\,n$ times, where n is the number of nodes of the internal quadrature rule (by default, $n = 20$ for $DIGITS \leq 10$). The following triple quadrature would call the *exp* function no less than $(3\,20)^3 = 216000$ times!

```
Q(Q(Q(exp(x*y*z), x = 0..y+z), y = 0..z), z = 0..1)
```

For low precision goals, low order quadrature rules suffice. In the following, we reduce the computational costs by using Gauss-Legendre quadrature with 5 nodes. We use the shorthand notation `GL` to specify the `GaussLegendre` method:

```
DIGITS := 4:
Q(Q(Q(exp(x*y*z), x=0..y+z, GL=5), y=0..z, GL=5), z=0..1, GL=5)
```

```
0.665
```

```
delete Q, b, f, DIGITS:
```

Example 4

We demonstrate how integrands given by user-defined procedures should be handled. The following integrand

```
f := proc(x) begin
      if x<1 then sin(x^2) else cos(x^5) end_if
    end_proc:
```

cannot be called with a symbolic argument:

```
f(x)
```

```
Error: Cannot evaluate to Boolean. [_less]
  Evaluating: f
```

Consequently, one must use `hold` to prevent premature evaluation of `f(x)`:

```
numeric::quadrature(hold(f)(x), x = -1..PI/2)
```

```
     0.5354101317
```

Note that the above integrand is discontinuous at $x = 1$, causing slow convergence of the numerical quadrature. It is much more efficient to split the integral into two subquadratures with smooth integrands:

```
numeric::quadrature(sin(x^2), x = -1..1) +
numeric::quadrature(cos(x^5), x = 1..PI/2)
```

```
     0.5354101318
```

See “Example 5” on page 19-277 for multi-dimensional quadrature of user-defined procedures.

```
delete f:
```

Example 5

The following integrand

```
f := proc(x, y) begin
       if x<y then x-y else (x-y) + (x-y)^5 end_if
     end_proc:
```

can only be called with numerical arguments and must be delayed twice for 2-dimensional quadrature:

```
Q := numeric::quadrature:

Q(Q(hold(hold(f))(x, y), x = 0..1), y = 0..1)
```

```
0.02380952381
```

Note that delaying the integrand via `hold` will not work for triple or higher-dimensional quadrature! The user can handle this by making sure that the integrand can also be evaluated for symbolic arguments:

```
f := proc(x, y, z)
     begin
       if not testtype([args()], Type::ListOf(Type::Numeric))
          then return(procname(args()))
       end_if;
       if x^2 + y^2 + z^2 <= 1
          then return(1)
          else return(0)
       end_if
     end_proc:
```

Note that this function is not continuous, implying slow convergence of the numerical quadrature. For this reason, we use a low precision goal of only 2 digits and reduce the costs by using a low order quadrature rule:

```
DIGITS := 2:
Q(Q(Q(f(x, y, z), x=0..1, GL=5), y=0..1, GL=5), z=0..1, GL=5)
```

```
0.52
```

```
delete f, Q, DIGITS:
```

Example 6

The following example uses non-adaptive Gauss-Tschebyscheff quadrature with an increasing number of nodes. The results converge quickly to the exact value:

```
a := exp(x)/sqrt(1 - x^2), x = -1..1:
numeric::quadrature(a, Adaptive = FALSE, GT = n) $ n = 3..7
```

```
3.97732196, 3.977462635, 3.977463259, 3.977463261, 3.977463261
```

```
delete a:
```

Example 7

The improper integral $\int_0^1 \frac{1}{x^{9/10}}\,dx = 10$ exists. Numerical convergence, however, is rather slow because of the singularity at $x = 0$:

```
numeric::quadrature(x^(-9/10), x = 0..1)
```

```
Warning: Precision goal is not achieved after 10000 function calls. Increase 'MaxCalls
```

```
     9.998221195
```

We remove the limit for the number of function calls and let `numeric::quadrature` grind along until a result is found. The time for the computation grows accordingly, the last digit is incorrect due to roundoff effects:

```
numeric::quadrature(x^(-9/10), x = 0..1, MaxCalls = infinity)
```

```
     9.999999993
```

Example 8

The following integral does not exist in the Riemann sense, because the integrand is not bounded:

```
numeric::quadrature(1/x, x = -1..2)
```

```
Warning: Precision goal is not achieved after 10000 function calls. Increase 'MaxCalls
```

```
     116.9391252
```

We increase `MaxCalls`. There is no convergence of the numerical algorithm, because the integral does not exist. Consequently, some internal problem must arise: `numeric::quadrature` reaches its maximal recursive depth:

```
numeric::quadrature(1/x, x = -1..2, MaxCalls = infinity)
```

```
Warning: Precision goal is not achieved after 'MAXDEPTH=500' recursive calls. There mig

    3.262987586
```

Parameters

`f(x)`

An arithmetical expression in x

`x`

An identifier or an indexed identifier

`a, b`

Real or complex numerical values or $\pm\infty$

Options

`GaussLegendre, GaussTschebyscheff, NewtonCotes`

Options, specified as `GaussLegendre = n`, `GaussTschebyscheff = n`, `NewtonCotes = n`

The name of the underlying quadrature scheme. Each quadrature rule can be used with an arbitrary number of nodes specified by the positive integer n.

Usually there is no need to use this option to change the default method `GaussLegendre = n` with `n = 20,40,80` or `160`, depending on the precision goal determined by the environment variable `DIGITS`. Due to the corresponding high quadrature orders 40, 80, 160 or 320, respectively, the default settings are suitable for high precision computations.

With `GaussLegendre = n`, an adaptive version of Gauss-Legendre quadrature with n nodes is used.

For $DIGITS \leq 200$, the weights and abscissae of Gaussian quadrature with n = 20, 40, 80, and 160 nodes are available and the integration starts immediately.

For *DIGITS* > 200, the routine `numeric::gldata` is called to compute the Gaussian data before the actual integration starts. This will be noted by some delay in the first call of `numeric::quadrature`.

For `DIGITS` much larger than 200, it is recommended not to use the default setting but to use `GaussLegendre = n` with sufficiently high `n` instead. A reasonable choice is $n \approx$ *DIGITS*.

With `GaussTschebyscheff = n`, non-adaptive Gauss-Tschebyscheff quadrature with n nodes is used. This method may only be used in conjunction with `Adaptive = FALSE`.

Note: With `NewtonCotes = n`, an adaptive version of Newton-Cotes quadrature with n nodes is used. Note that these quadrature rules become numerically unstable for large n (n much larger than 10).

Following alternative names for the methods are accepted:

`GaussLegendre`, `Gauss`, `GL`,

`GaussTschebyscheff`, `GT`, `GaussChebyshev`, `GC`,

`NewtonCotes`, `NC`.

`Adaptive`

Option, specified as `Adaptive = v`

`v` may be `TRUE` or `FALSE`. With `Adaptive = FALSE`, the internal error control is switched off.

The default setting is `Adaptive = TRUE`. An adaptive quadrature scheme with automatic control of the quadrature error is used.

The call `numeric::quadrature(f(x), x = a..b, method = n, Adaptive = FALSE)` returns the quadrature sum $(b-a)\left(\sum_{i=1}^{n} B_i f\left(a + C_i (b-a)\right)\right)$ approximating $\int_a^b f(x)\,\mathrm{d}x$ without any control of the quadrature error. The weights B_i and abscissae C_i are determined by the quadrature rule given by `method = n`. For the methods `GaussLegendre`, `GaussTschebyscheff` and `NewtonCotes`, these data are available via `numeric::gldata`, `numeric::gtdata` and `numeric::ncdata`, respectively.

`Adaptive = FALSE` may only be used in conjunction with `method = n`.

Usually there is no need to switch off the internal adaptive quadrature via `Adaptive = FALSE`. This option is meant to give easy access to standard non-adaptive quadrature rules of Gauss-Legendre, Gauss-Tschebyscheff and Newton-Cotes type, respectively. The user may want to build his own adaptive quadrature based on these non-adaptive rules. Cf. “Example 6” on page 19-278.

`MaxCalls`

Option, specified as `MaxCalls = m`

`m` must be a (large) positive integer or `infinity`. It is the maximal number of evaluations of the integrand, before `numeric::quadrature` gives up.

When called interactively, `numeric::quadrature` returns the approximation it has computed so far and issues a warning. See “Example 7” on page 19-279. When called from inside a procedure, `numeric::quadrature` returns `FAIL`.

The default value is `m = MAXDEPTH*n`. The environment variable `MAXDEPTH` (default value 500) represents the maximal recursive depth of a function. `n` is the number of nodes of the basic quadrature rule specified by the optional argument `method = n`. If no such method is specified by the user, then Gauss-Legendre quadrature is used with n = 20 for $DIGITS \le 10$, n = 40 for $10 < DIGITS \le 50$, n = 80 for $50 < DIGITS \le 100$, n = 160 for $100 < DIGITS$. Thus, for $DIGITS = 10$, the default setting is `MaxCalls = 10000`.

The default value of `m` ensures that the `MaxCalls` limit is reached before `numeric::quadrature` reaches its maximal internal recursive depth. Specifying `MaxCalls = infinity` removes this restriction and `numeric::quadrature` computes until it obtains an approximation with about `DIGITS` correct digits or until it runs into an internal error. The typical error that may occur is the evaluation of the integrand at a singularity. There also is the danger of `numeric::quadrature` reaching its maximal internal recursive depth. When called interactively, `numeric::quadrature` returns the approximation it has computed so far and issues a warning. See “Example 8” on page 19-279. When called from inside a procedure, `numeric::quadrature` returns `FAIL`.

Return Values

Floating point number is returned, unless non-numerical symbolic objects in the integrand `f(x)` or in the boundaries `a,b` prevent numerical evaluation. In this case,

`numeric::quadrature` returns itself symbolically. If numerical problems occur, then `FAIL` is returned.

See Also

MuPAD Functions

`int` | `numeric::gldata` | `numeric::gtdata` | `numeric::int` | `numeric::ncdata`

numeric::rank

Numerical estimate of the rank of a matrix

Syntax

```
numeric::rank(A, <eps>, options)
```

Description

`numeric::rank(A)` returns an integer indicating the rank of the matrix *A*.

All entries of the input matrix must be numerical, i.e., they must be floating-point numbers or expressions that can be converted to floating-point numbers.

The rank of a matrix coincides with the number of non-zero singular values.

A numerical estimate of the rank is computed by counting all singular values that are larger than $eps\, s_max$, where s_max is the largest singular value. (All smaller singular values are regarded as round-off artifacts and treated as zero.)

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We consider a quadratic matrix of rank 2:

```
A := matrix([[1, 1, I],
             [1, 2, 3],
             [2, 4, 6]]):
```

```
numeric::rank(A)
```

$$2$$

Hilbert matrices have full rank. However, they are extremely ill-conditioned and it is difficult to compute their rank numerically. The 10×10 Hilbert matrix has rank 10. Numerically, however, some of the singular values are so small that they may be regarded as zero resulting in a smaller numerical rank. In particular, with the default value eps = $\frac{1}{10^{\mathrm{DIGITS}}}$, two singular values are smaller than $eps\, s_max$ where s_{max} = 1.7519... is the maximal singular value:

```
A := linalg::hilbert(10):
numeric::singularvalues(A)
```

$$\left[1.75191967,\ 0.3429295485,\ 0.03574181627,\ 0.002530890769,\ 0.0001287496143,\ 0.000004729689293,\ 0.0000001228967739,\ 0.00000000214743881 8,\ 2.26674678\ 10^{-11},\ 1.093197787\ 10^{-13}\right]$$

```
numeric::rank(A)
```

$$8$$

We specify a second argument `eps = 10^-14` to allow smaller singular values to be regarded as non-zero. Now, the numerical rank is 10:

```
numeric::rank(A, 10^(-14))
```

$$10$$

```
delete A:
```

Example 2

We consider a non-square matrix of rank 1:

```
A := matrix([[0, 0],
             [I, 1],
             [I, 1]]):
numeric::rank(A)
```

```
    1
```

```
delete A:
```

Example 3

We demonstrate the difference between hardware floats and software floats:

```
A := linalg::hilbert(15):
numeric::rank(A, 10^(-20), SoftwareFloats),
numeric::rank(A, 10^(-20), HardwareFloats)
```

```
    14, 15
```

```
delete A:
```

Parameters

A

An $m \times n$ matrix of domain type `DOM_ARRAY`, `DOM_HFARRAY`, or of category `Cat::Matrix`

eps

Relative tolerance: regard all singular values s of A as zero if they satisfy $s \leq eps\, s_max$, where s_max is the largest singular value of A. The default value of eps is $\frac{1}{10^{\mathrm{DIGITS}}}$.

Options

Hard, HardwareFloats, Soft, SoftwareFloats

With `Hard` (or `HardwareFloats`), computations are done using fast hardware float arithmetic from within a MuPAD session. `Hard` and `HardwareFloats` are equivalent.

With this option, the input data are converted to hardware floats and processed by compiled C code. The result is reconverted to MuPAD floats and returned to the MuPAD session.

With `Soft` (or `SoftwareFloats`) computations are dome using software float arithmetic provided by the MuPAD kernel. `Soft` and `SoftwareFloats` are equivalent. `SoftwareFloats` is used by default if the current value of `DIGITS` is larger than 15 and the input matrix `A` is not of domain type `DOM_HFARRAY`.

Compared to the `SoftwareFloats` used by the MuPAD kernel, the computation with `HardwareFloats` may be many times faster. Note, however, that the precision of hardware arithmetic is limited to about 15 digits. Further, the size of floating-point numbers may not be larger than approximately 10^{308} and not smaller than approximately 10^{-308}.

If no `HardwareFloats` or `SoftwareFloats` are requested explicitly, the following strategy is used: If the current value of `DIGITS` is smaller than 16 or if the matrix `A` is a hardware float array of domain type `DOM_HFARRAY`, then hardware arithmetic is tried. If this is successful, the result is returned.

If the result cannot be computed with hardware floats, software arithmetic by the MuPAD kernel is tried.

If the current value of `DIGITS` is larger than 15 and the input matrix `A` is not of domain type `DOM_HFARRAY`, or if one of the options `Soft`, `SoftwareFloats` or `Symbolic` is specified, MuPAD computes the result with its software arithmetic without trying to use hardware floats first.

There may be several reasons for hardware arithmetic to fail:

- The current value of `DIGITS` is larger than 15.
- The data contains symbolic objects.
- The data contains numbers larger than 10^{308} or smaller than 10^{-308} that cannot be represented by hardware floats.

If neither `HardwareFloats` nor `SoftwareFloats` is specified, the user is not informed whether hardware floats or software floats are used.

If `HardwareFloats` are specified but fail due to one of the reasons above, a warning is issued that the (much slower) software floating-point arithmetic of the MuPAD kernel is used.

Note that `HardwareFloats` can only be used if all input data can be converted to floating-point numbers.

Note: For ill-conditioned matrices, the results returned with `HardwareFloats` and `SoftwareFloats` may differ significantly! See “Example 3” on page 19-286.

Return Values

Positive integer.

See Also

MuPAD Functions
`linalg::rank`

numeric::rationalize

Approximate a floating-point number by a rational number

Syntax

```
numeric::rationalize(object, <Exact | Minimize | Restore>, <digits>)
```

Description

`numeric::rationalize(object)` replaces all floating-point numbers in `object` by rational numbers.

An object of a library domain, characterized by `domtype(extop(object,0))=DOM_DOMAIN` is returned unchanged. For all other objects, `numeric::rationalize` is applied recursively to all operands. Objects of library domains can be rationalized if the domain has an appropriate `map` method. See "Example 5" on page 19-293.

A floating-point number f is approximated by a rational number r satisfying $|f - r| < \varepsilon |f|$.

Note: With the options `Exact` and `Minimize`, the guaranteed precision is $\varepsilon = \frac{1}{10^{\text{digits}}}$.

With `Restore`, the guaranteed precision is only $\varepsilon = \frac{1}{10^{\frac{\text{digits}}{2}}}$.

The default precision is `digits = DIGITS`.

The user defined precision must not be larger than the internal floating-point precision set by `DIGITS`: an error occurs for `digits > DIGITS`.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`.

Examples

Example 1

`numeric::rationalize` is applied to each operand of a composite object:

```
numeric::rationalize(0.2*a+b^(0.7*I))
```

$$\frac{a}{5} + b^{\frac{7\,i}{10}}$$

```
numeric::rationalize([{poly(0.2*x, [x]), sin(7.2*PI) + 1.0*I},
                      exp(3 + ln(2.0*x))])
```

$$\left[\left\{\sin\left(\frac{36\,\pi}{5}\right) + i,\ \mathrm{poly}\left(\frac{x}{5}, [x]\right)\right\},\ e^{\ln(2\,x)+3}\right]$$

Example 2

We demonstrate the default strategy `Exact`:

```
numeric::rationalize(12.3 + 0.5*I),
numeric::rationalize(0.33333),
numeric::rationalize(1/3.0)
```

$$\frac{123}{10} + \frac{i}{2},\ \frac{33333}{100000},\ \frac{33333333333}{100000000000}$$

```
numeric::rationalize(10^12/13.0),
numeric::rationalize(10^(-12)/13.0)
```

$$76923076923,\ \frac{76923076923}{1000000000000000000000000}$$

We reduce the precision of the approximation to 5 digits:

```
numeric::rationalize(10^12/13.0, 5),
```

```
numeric::rationalize(10^(-12)/13.0, 5)
```

$$76923100000,\ \frac{769231}{10000000000000000000}$$

Example 3

We demonstrate the strategy `Minimize` for minimizing the complexity of the resulting rational number:

```
numeric::rationalize(1/13.0, 5),
numeric::rationalize(1/13.0, Minimize, 5),
numeric::rationalize(0.333331, 5),
numeric::rationalize(0.333331, Minimize, 5),
numeric::rationalize(14.285, 5),
numeric::rationalize(14.2857, Minimize, 5),
numeric::rationalize(1234.1/56789.2),
numeric::rationalize(1234.1/56789.2, Minimize)
```

$$\frac{769231}{10000000},\ \frac{1}{13},\ \frac{333331}{1000000},\ \frac{1}{3},\ \frac{2857}{200},\ \frac{100}{7},\ \frac{21731244673}{1000000000000},\ \frac{12341}{567892}$$

We compute rational approximations of π with various precisions:

```
numeric::rationalize(float(PI), Minimize, i) $ i = 1..10
```

$$3,\ \frac{22}{7},\ \frac{22}{7},\ \frac{355}{113},\ \frac{355}{113},\ \frac{355}{113},\ \frac{355}{113},\ \frac{102573}{32650},\ \frac{104348}{33215},\ \frac{208341}{66317}$$

Example 4

We demonstrate the strategy `Restore` for restoring rational numbers after elementary float operations. In many cases, also the `Minimize` strategy restores:

```
numeric::rationalize(1/7.3, Exact),
numeric::rationalize(1/7.3, Minimize),
numeric::rationalize(1/7.3, Restore)
```

$$\frac{13698630137}{100000000000},\ \frac{10}{73},\ \frac{10}{73}$$

However, using `Restore` improves the chances of recovering from roundoff effects:

```
numeric::rationalize(10^9/13.0, Minimize),
numeric::rationalize(10^9/13.0, Restore)
```

$$\frac{923076923}{12}, \frac{1000000000}{13}$$

```
DIGITS:= 11:
numeric::rationalize(1234.56/12345.67, Minimize),
numeric::rationalize(1234.56/12345.67, Restore)
```

$$\frac{88183}{881835}, \frac{123456}{1234567}$$

In some cases, `Restore` manages to recover from roundoff error propagation in composite arithmetical operations:

```
DIGITS:= 10:
x:= float(122393/75025):
y:= float(121393/75025):
z := (x^2 - y^2)/(x + y)
```

$$0.01332889037$$

```
numeric::rationalize(z, Restore)
```

$$\frac{40}{3001}$$

The result with `Restore` corresponds to exact arithmetic:

```
rx := numeric::rationalize(x, Restore):
ry := numeric::rationalize(y, Restore):
rx, ry, (rx^2 - ry^2)/(rx + ry)
```

$$\frac{122393}{75025}, \frac{121393}{75025}, \frac{40}{3001}$$

Note that an approximation with `Restore` may have a reduced precision of only `digits/2`:

```
x := 1.0 + 1/10^6:
numeric::rationalize(x, Exact),
numeric::rationalize(x, Restore)
```

$$\frac{1000001}{1000000},\ 1$$

```
delete x, y, z, rx, ry:
```

Example 5

The floats inside objects of library domains are not rationalized directly. However, for most domains the corresponding `map` method can forward `numeric::rationalize` to the operands:

```
Dom::Multiset(0.2, 0.2, 1/5, 0.3)
```

$$\left\{[0.2,\ 2],\ \left[\frac{1}{5},\ 1\right],\ [0.3,\ 1]\right\}$$

```
numeric::rationalize(%), map(%, numeric::rationalize, Restore)
```

$$\left\{[0.2,\ 2],\ \left[\frac{1}{5},\ 1\right],\ [0.3,\ 1]\right\},\ \left\{\left[\frac{1}{5},\ 3\right],\ \left[\frac{3}{10},\ 1\right]\right\}$$

Parameters

`object`

An arbitrary MuPAD object

`digits`

A positive integer (the number of decimal digits) not bigger than the environment variable `DIGITS`. It determines the precision of the rational approximation.

Options

`Exact`

Specifies the strategy for approximating floating-point numbers by rational numbers. This is the default strategy, so there is no real need to pass `Exact` as a parameter to `numeric::rationalize`.

Any real floating-point number $f \neq 0.0$ has a unique representation

$$f = \text{sign}(f)\ \text{mantissa}\ 10^{\text{exponent}}$$

With integer exponent and $1.0 \leq \textit{mantissa} < 10.0$. With the option `Exact`, the float mantissa is replaced by the rational approximation

$$\frac{\text{round}(\text{mantissa}\ 10^{\text{digits}})}{10^{\text{digits}}}$$

.

This guarantees a relative precision of `digits` significant decimals of the rational approximation.

`Minimize`

Specifies the strategy for approximating floating-point numbers by rational numbers. This strategy tries to minimize the complexity of the rational approximation, i.e., numerators and denominators are to be small.

The guaranteed precision of the rational approximation is `digits`.

See “Example 3” on page 19-291.

`Restore`

Specifies the strategy for approximating floating-point numbers by rational numbers. This strategy tries to restore rational numbers obtained after *elementary* arithmetical operations applied to floating-point numbers. E.g., for rational `r`, the float division `f = 1/float(r)` introduces additional roundoff, which the `Restore` algorithm tries to eliminate: `numeric::rationalize(f, Restore) = 1/r`. This strategy, however, is purely heuristic and will not succeed when significant roundoff is caused by arithmetical float operations!

Note: The guaranteed precision of the rational approximation is only `digits/2`!

See “Example 4” on page 19-291.

Return Values

If the argument is an object of some kernel domain, then it is returned with all floating-point operands replaced by rational numbers. An object of some library domain is returned unchanged.

Overloaded By

object

Algorithms

Continued fraction (CF) expansion is used with the options `Minimize` and `Restore`.

With `Minimize`, the first CF approximation satisfying the precision criterion is returned.

The `Restore` algorithm stops, when large coefficients of the CF expansion are found.

numeric::realroot

Numerical search for a real root of a real univariate function

Syntax

```
numeric::realroot(f(x), x = a .. b, <SearchLevel = s>)
```

Description

`numeric::realroot(f(x), x = a..b)` computes a numerical real root of $f(x)$ in the interval $[a, b]$.

The expression `f(x)` must not contain symbolic objects other than the indeterminate `x` that cannot be converted to numerical values via `float`. Symbolic objects such as π or $\sqrt{2}$ etc. are accepted. The same holds true for the boundaries `a, b` of the search interval.

The function must produce real values. If `float(f(x))` does not yield real floating-point numbers for all real floating-point numbers `x` from the interval $[a, b]$, internal problems may occur. See "Example 5" on page 19-299.

`numeric::realroot` never tries to evaluate `f(x)` outside the search interval. Consequently, singularities outside the interval do not cause any problems. In many cases also singularities inside the interval do not affect the numerical search. However, `numeric::realroot` is not guaranteed to work in such a case. An error may occur, if the internal search accidentally hits a singularity. See "Example 5" on page 19-299.

Up to roundoff effects numerical roots r with $|r| \geq \frac{1}{10^{\mathrm{DIGITS}}}$ are computed to a relative precision of `DIGITS` significant decimal places. Roots of smaller absolute size are computed to an absolute precision of $\frac{1}{10^{2\ \mathrm{DIGITS}}}$. These precision goals are not achieved, if significant roundoff occurs in the numerical evaluation of $f(x)$.

If f takes opposite signs at the endpoints a, b of the search interval and does not have zero-crossing singularities, then `numeric::realroot` is bound to find a root in the interval $[a, b]$.

User defined functions can be handled. See “Example 2” on page 19-297.

Note: `numeric::realroot` approximates a point where `f(x)` changes its sign. This is a root only if the function `f` is continuous. See “Example 3” on page 19-298.

Note that `numeric::realroots` may be used to isolate *all* real roots. However, this function is much slower than `numeric::realroot`, if *f* is not a polynomial.

For univariate polynomials we recommend to use `numeric::realroots` or `polylib::realroots` rather than `numeric::realroot`.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

The following functions assume different signs at the boundaries, so the searches are bound to succeed:

```
numeric::realroot(x^3 - exp(3), x = -PI..10)
```

```
     2.718281828
```

```
numeric::realroot(exp(-x[1]) = x[1], x[1] = 0..1)
```

```
     0.5671432904
```

Example 2

The following function cannot be evaluated for non-numerical `x`. So one has to delay evaluation via `hold`:

```
f := proc(x) begin
      if x<0 then 1 - x else exp(x) - 10*x end_if
     end_proc:
numeric::realroot(hold(f)(x), x = -10..10)
```

```
     0.1118325592
```

```
delete f:
```

Example 3

`numeric::realroot` approximates a point, where $f(x)$ changes its sign. For the following function this happens at the discontinuity $x = 1$:

```
f := proc(x) begin if x<1 then -1 else x end_if end_proc:
numeric::realroot(hold(f)(x), x = 0..3)
```

```
     1.0
```

```
delete f:
```

Example 4

The following function does not have a real root. Consequently, `numeric::realroot` fails:

```
numeric::realroot(x^2 + 1, x = -2..2)
```

```
     FAIL
```

The following function does not have a real root in the search interval:

```
numeric::realroot(x^2 - 1, x =  2..3)
```

```
     FAIL
```

Example 5

The following function is complex valued for $x^2 < 3.5$. An error occurs, when the internal search hits such a point:

```
numeric::realroot(ln(x^2 - 3.5), x = -2..3)
```

```
Error: Cannot evaluate to Boolean. [_less]
  Evaluating: numeric::BrentFindRoot
```

The singularity at $x = 1$ does not cause any problem in the following call:

```
numeric::realroot((x-2)/(x-1), x = -10..10)
```

```
    2.0
```

However, the singularity may be hit accidentally in the internal search:

```
numeric::realroot((x-2)/(x-1), x = 0..3)
```

```
Error: Division by zero. [_power]
  Evaluating: f
```

Example 6

The following function has a root close to 1.0, which is difficult to detect. With the default search level $s = 1$, this root is not found:

```
f := 2 - exp(-100*(x - 2)^2) - 2*exp(-1000*(x - 1)^2):
numeric::realroot(f, x = 0..5)
```

```
    FAIL
```

The root is detected with an increased search level:

```
numeric::realroot(f, x = 0..5, SearchLevel = 3)
```

```
    1.0
```

```
delete f:
```

Parameters

`f(x)`

An arithmetical expression in one unknown `x` or a list, set, `array`, or `matrix` (`Cat::Matrix`) of expressions. Alternatively, equations in the form $f_1(x) = f_2(x)$ equivalent to the expressions $f_1(x) - f_2(x)$.

`x`

An identifier or an indexed identifier

`a, b`

Finite real numerical values

Options

`SearchLevel`

Option, specified as `SearchLevel = s`

The nonnegative integer s controls the internal refinement of the search. The default value is $s = 1$. Increasing s increases the chance of finding roots that are difficult to detect numerically. See “Example 6” on page 19-299.

Note that increasing s by 1 may quadruple the time before `FAIL` is returned, if no real root is found. For this reason we recommend to restrict s to small values ($s \leq 5$, say).

Return Values

Single numerical real root of domain type `DOM_FLOAT`. If no solution is found, then `FAIL` is returned.

Algorithms

A mixture of bisectioning, secant steps and quadratic interpolation is used by `numeric::realroot`.

See Also

MuPAD Functions

`numeric::fsolve` | `numeric::polyroots` | `numeric::realroots` | `polylib::realroots` | `solve`

More About

- "Solve Equations Numerically"

numeric::realroots

Isolate intervals containing real roots of a univariate function

Syntax

```
numeric::realroots(f(x), <x = a .. b>, <eps>, <Merge = c>)
```

Description

`numeric::realroots(f(x), x = a..b)` searches for real roots of $f(x)$ in the interval $[a, b]$. It returns a list of subintervals in which real roots of $f(x)$ may exist. It is guaranteed that there are no real roots in the interval $[a, b]$ lying outside the union of the returned subintervals.

With `Merge = FALSE`, all intervals returned by `numeric::realroots` have length $b_i - a_i < eps$ with a default value $eps = 0.01$. The absolute precision of the root isolation may be redefined using the optional parameter `eps`.

Note: The intervals returned by `numeric::realroots` define a subset of $[a, b]$ that may contain real roots. For polynomial expressions $f(x)$, each of the subintervals of $[a, b]$ returned by `numeric::realroots` is guaranteed to contain exactly one root. For non-polynomial expressions $f(x)$, however, some of the subintervals may contain no root! Cf. "Example 6" on page 19-306.

In any case, the complement $[a, b] \setminus \left(\bigcup_{i \in \{1, \ldots, n\}} [a_i, b_i] \right)$ of the subintervals $[a_i, b_i]$ returned by `numeric::realroots` is guaranteed to contain no real roots. In particular, from the return value [], one may positively conclude that no root exists in the search interval $[a, b]$. Cf. "Example 2" on page 19-304.

Symbolic parameters in $f(x)$ are not allowed: `float(f(x))` must evaluate to a floating point number for all x from the interval $[a, b]$.

Infinite intervals of the form `x = -infinity..b` are not refined if $b \leq -10^5$.

Infinite intervals of the form `x = a..infinity` are not refined if $a \geq 10^5$.

Such intervals are returned directly if `numeric::realroots` thinks that they may contain roots. Cf. “Example 5” on page 19-305.

f(*x*) may contain complex expressions. Only the search parameter *x* is assumed to be real. For complex expressions `f(x)`, the intervals are returned where both the real and the imaginary part of the expression vanish simultaneously.

Note: The expression `f(x)` must be suitable for interval arithmetic. In particular, MuPAD must be able to evaluate `f(a...b)`. Note that not all MuPAD functions support this kind of arithmetic.

Presently, the following special functions support interval arithmetic: `abs`, `arccos`, `arccosh`, `arccoth`, `arccot`, `arccsc`, `arccsch`, `arcsec`, `arcsech`, `arcsin`, `arcsinh`, `arctan`, `arctanh`, `arg`, `beta`, `ceil`, `cos`, `cosh`, `cot`, `coth`, `csc`, `csch`, `dirac`, `exp`, `floor`, `gamma`, `Im`, `ln`, `Re`, `round`, `sec`, `sech`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`, `trunc`. Real roots can be searched for any expression that is built from these functions using the standard arithmetical operations `+`, `-`, `*`, `/`, `^`.

The default value is *eps* = 0.01. User defined precision goals must satisfy $\mathrm{eps} \geq \frac{1}{10^{\mathrm{DIGITS}}}$.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

The following expression has integer zeros. The solutions in the specified interval are approximated to the default precision 0.01:

```
numeric::realroots(sin(PI*x), x = -2..sqrt(2))
```

```
[[-2.0, -1.993331614], [-1.006410506, -0.9997421204],
  [-0.006152626661, 0.000515759203], [0.9941052529, 1.000773639]]
```

The following equation is solved with an absolute precision of 7 digits:

```
numeric::realroots(x*sin(x) = exp(-x), x = -1..1, 10^(-7))
```

```
[[0.7271005511, 0.7271006107]]
```

Example 2

The following expression does not have a real root:

```
numeric::realroots(exp(x) + x^2, x = -100..100)
```

```
[]
```

Example 3

We demonstrate the option `Merge`. If interval arithmetic can not isolate roots to the desired precision *eps* (default 0.01), then adjacent intervals are produced, each of length smaller than *eps*. This happens in the following example:

```
numeric::realroots(ln(x^2 -2*x + 2) = 0, x = -10..10,
                   Merge = FALSE)
```

```
[[0.869140625, 0.87890625], [0.87890625, 0.888671875], [0.888671875, 0.8984375], ...,
  [1.123046875, 1.1328125], [1.1328125, 1.142578125]]
```

[[0.869140625, 0.87890625], [0.87890625, 0.888671875], [0.888671875, 0.8984375], [0.8984375, 0.908203125], [0.908203125, 0.91796875], [0.91796875, 0.927734375], [0.927734375, 0.9375], [0.9375, 0.947265625], [0.947265625, 0.95703125], [0.95703125, 0.966796875], [0.966796875, 0.9765625], [0.9765625, 0.986328125], [0.986328125, 0.99609375], [0.99609375, 1.005859375], [1.005859375, 1.015625], [1.015625, 1.025390625], [1.025390625, 1.03515625], [1.03515625, 1.044921875], [1.044921875, 1.0546875], [1.0546875, 1.064453125], [1.064453125, 1.07421875], [1.07421875,

```
1.083984375], [1.083984375, 1.09375], [1.09375, 1.103515625], [1.103515625,
1.11328125], [1.11328125, 1.123046875], [1.123046875, 1.1328125], [1.1328125,
1.142578125]]
```

With `Merge = TRUE`, these intervals are combined to a single larger interval. Since `Merge = TRUE` is the default setting for non-polynomial functions, it suffices to omit the option `Merge = FALSE`:

```
numeric::realroots(ln(x^2 -2*x + 2) = 0, x = -10..10)
```

```
[[0.869140625, 1.142578125]]
```

Example 4

The following expression has infinitely many solutions $x = \frac{1}{n}$ with $n = 1, 2, \ldots$ in the search interval [0, 1]:

```
numeric::realroots(sin(PI/x), x = 0..1, 0.1, Merge = FALSE)
```

```
[[0.0, 0.0625], [0.0625, 0.125], [0.125, 0.1875], [0.1875, 0.25], [0.25, 0.3125], [0.3125, 0.375],
  [0.4375, 0.5], [0.5, 0.5625], [0.9375, 1.0]]
```

Omitting `Merge = FALSE`, adjacent intevals are merged to larger intervals. The first of the following intervals contains infinitely many roots:

```
numeric::realroots(sin(PI/x), x = 0..1, 0.1)
```

```
[[0.0, 0.375], [0.4375, 0.5625], [0.9375, 1.0]]
```

Example 5

If no search interval is specified, the entire real line is considered:

```
numeric::realroots(x^3 = exp(x))
```

```
[[1.841783524, 1.871585846], [4.523992538, 4.553794861], [100000.0, ∞]]
```

Apart from two finite intervals, `numeric::realroots` tells us that there may be a root close to infinity (but that there is positively no root close to -infinity). Analytically, it is clear that the subinterval $[10000.0, \infty]$ cannot contain a root, since e^x grows much faster than x^3 as x goes to infinity. If a finite upper limit for the search interval is specified, this fact is detected:

```
numeric::realroots(x^3 = exp(x), x = -infinity .. 10^100)
```

$$[[1.842076975, 1.877795331], [4.512024084, 4.556672029]]$$

We isolate the two finite roots more closely by specifying a precision goal of $\frac{1}{10^{\mathrm{DIGITS}}}$:

```
numeric::realroots(x^3 = exp(x), x = -infinity .. 10^100,
                   10^(-DIGITS))
```

$$[[1.85718386, 1.85718386], [4.536403655, 4.536403655]]$$

Example 6

The following equation has no root close to 0. However, interval arithmetic does not produce realistic values of $\frac{\sin(\pi x)}{x}$ for small intervals containing $x = 0$, so an isolating interval around 0 is returned:

```
numeric::realroots(sin(PI*x)/x = 0, x = -1..1.2)
```

$$[[-1.0, -0.99140625], [-0.003125, 0.00546875], [0.99375, 1.00234375]]$$

A similar phenomenon occurs with $x^x = e^{x \ln(x)}$ in a neighbourhood of $x = 0$. An isolating interval around 0 is returned, although no solution exists there:

```
numeric::realroots(x^x*cos(PI*x) = tan(x), x = 0..1)
```

$$[[0.0, 0.0078125], [0.328125, 0.3359375]]$$

This cannot be cured by increasing the precision goal:

```
numeric::realroots(x^x*cos(PI*x) = tan(x), x = 0..1,
                   10^(-DIGITS))
```

$$\left[\left[0.0,\ 5.820766091\ 10^{-11}\right],\ [0.3334737903,\ 0.3334737903]\right]$$

Parameters

`f(x)`

An expression in one indeterminate `x` or a list, set, `array`, or `matrix` (`Cat::Matrix`) of expressions. Alternatively, equations in the form $f_1(x) = f_2(x)$ equivalent to the expressions $f_1(x) - f_2(x)$.

`x`

An identifier or an indexed identifier

`a, b`

Real numbers or numerical expressions satisfying $a < b$. Also `- infinity` and `infinity` may be used.

`eps`

A (small) positive real numerical value defining the precision goal. The default value is 0.01.

Options

`Merge`

Option, specified as `Merge = b`

`b` can be `TRUE` or `FALSE`. With `Merge = FALSE`, `numeric::realroots` returns subintervals of length not larger than `eps`. With `Merge = TRUE`, `numeric::realroots` merges adjacent subintervals to larger intervals, i.e., subintervals of length larger than `eps` may be returned.

The default setting is `Merge = FALSE` for polynomial functions $f(x)$. Otherwise, it is `Merge = TRUE`.

`numeric::realroots` isolates intervals $[a_i, b_i]$ that may contain roots. Internally, all these intervals satisfy $b_i - a_i < eps$ where eps is the precision goal.

With `Merge = FALSE`, these intervals are returned.

With `Merge = TRUE`, adjacent intervals $[a_i, b_i]$, $[a_{i+1}, b_{i+1}]$ with $b_i = a_{i+1}$ are combined to larger intervals $[a_i, b_{i+1}]$. See “Example 3” on page 19-304 and “Example 4” on page 19-305.

The default setting is `Merge = FALSE` for polynomial functions. Otherwise, it is `Merge = TRUE`.

Return Values

List $[[a_1, b_1], [a_2, b_2], \ldots]$ of disjoint floating-point intervals $[a_i, b_i] \subset [a, b]$ which may contain roots of $f(x)$. The empty list is returned if no root exists in the search interval $[a, b]$.

Algorithms

Let X be a subset of the real numbers. Interval arithmetic produces a set $F(X)$ such that the set of image values $\{f(x) \mid x \in X\}$ is contained in $F(X)$. The MuPAD domain `DOM_INTERVAL` facilitates this kind of arithmetic. The routine `numeric::realroots` computes $F := f(a_i \ldots b_i)$ for various subintervals $[a_i, b_i]$ of $[a, b]$. If F does not contain zero, then this subinterval is eliminated from the search interval. Otherwise, the subinterval is returned as a candidate for containing zeros of $f(x)$. However, one cannot conclude that F does indeed contain at least one zero, since F is usually larger than the true image set $\{f(x) \mid x \in [a_i, b_i]\}$ (‘overestimation’).

For polynomials $f(x)$, the routine `polylib::realroots` is called. Its results are intersected with the search interval $[a, b]$. No interval arithmetic is used for

polynomial expressions. For polynomial equations, each isolating interval returned by `numeric::realroots` is guaranteed to contain at least one root if `Merge = TRUE` is specified. With the default setting of `Merge = FALSE` for polynomials, each isolating interval is guaranteed to contain exactly one root.

See Also

MuPAD Domains

`DOM_INTERVAL`

MuPAD Functions

`numeric::fsolve` | `numeric::polyroots` | `numeric::realroot` | `polylib::realroots` | `solve`

More About

- "Solve Equations Numerically"

numeric::rotationMatrix

Orthogonal matrix of the rotation about an axis

Syntax

```
numeric::rotationMatrix(angle, axis, <Symbolic>, <ReturnType = t>)
```

Description

`numeric::rotationMatrix(angle, axis)` returns an orthogonal matrix corresponding to the rotation about the given axis by the given angle.

The rotation by the angle *alpha* about the axis given by the vector [*x*, *y*, *z*] of Euclidean length 1 is given by the rotation matrix

$$e^{\text{alpha}\begin{pmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{pmatrix}} = \begin{pmatrix} t\,x^2+c & t\,x\,y-s\,z & t\,x\,z+s\,y \\ t\,x\,y+s\,z & t\,y^2+c & t\,y\,z-s\,x \\ t\,x\,z-s\,y & t\,y\,z+s\,x & t\,z^2+c \end{pmatrix}$$

with $c = cos(alpha)$, $s = sin(alpha)$, and $t = 1 - c = 2\sin\left(\frac{\text{alpha}}{2}\right)^2$.

The rotation is implemented following the “right hand rule”: Stretch the thumb of your right hand and bend the fingers. When the thumb points into the direction of the rotation axis, your finger tips indicate the direction of the rotation.

Use negative angles to rotate in the opposite direction.

The axis parameter of the routine does not need to be normalized to the Euclidean length 1. However, it must not be of zero length.

If no return type is specified via the option `ReturnType = t`, the domain type of the result depends on the type of the input matrix `axis`:

- If the axis is of domain type array, then the rotation matrix is returned as an array.
- If the axis is of domain type hfarray, then the result is returned as an hfarray.
- If the axis is of domain type `Dom::DenseMatrix()`, then the rotation matrix is returned as a matrix of type `Dom::DenseMatrix()` over the ring of MuPAD expressions.
- If axis is of any different matrix type, the result is a `matrix` of type `Dom::Matrix()` over the ring of MuPAD expressions. This includes input matrices `axis` of type `Dom::Matrix(...)`, `Dom::SquareMatrix(...)`, `Dom::MatrixGroup(...)` etc.
- If axis is a list with 3 elements, the rotation matrix is also returned as an `Dom::Matrix()` over the ring of MuPAD expressions.

Without the option `Symbolic`, all arguments are automatically converted to floating-point arguments (if possible). Use the option `Symbolic` if no such conversion is desired.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

The rotation around the z axis by 45 degrees is given by the following matrix:

```
numeric::rotationMatrix(PI/4, [0, 0, 1])
```

$$\begin{pmatrix} 0.7071067812 & -0.7071067812 & 0 \\ 0.7071067812 & 0.7071067812 & 0 \\ 0 & 0 & 1.0 \end{pmatrix}$$

Symbolic arguments are accepted:

```
numeric::rotationMatrix(a, [1, 2, 3])
```

$$\begin{bmatrix}0.9285714286\cos(a)+0.07142857143, & 0.1428571429-0.8017837257\sin(a)\\ -0.1428571429\cos(a), & 0.5345224838\sin(a)-0.2142857143\cos(a)+0.2142857143\end{bmatrix},$$
$$\begin{bmatrix}0.8017837257\sin(a)-0.1428571429\cos(a)+0.1428571429, & \\ 0.7142857143\cos(a)+0.2857142857, & 0.4285714286-0.2672612419\sin(a)\\ -0.4285714286\cos(a)\end{bmatrix},$$
$$\begin{bmatrix}0.2142857143-0.5345224838\sin(a)-0.2142857143\cos(a), & \\ 0.2672612419\sin(a)-0.4285714286\cos(a)+0.4285714286, & 0.3571428571\cos(a)\\ +0.6428571429\end{bmatrix}$$

With the option `Symbolic`, no automatic conversion to floating-point numbers occurs:

```
numeric::rotationMatrix(a, [1, 2, 3], Symbolic)
```

$$\begin{pmatrix}\frac{13\cos(a)}{14}+\frac{1}{14} & \frac{1}{7}-\sigma_1-\frac{\cos(a)}{7} & \frac{\sigma_4}{7}-\sigma_2+\frac{3}{14}\\ \sigma_1-\frac{\cos(a)}{7}+\frac{1}{7} & \frac{5\cos(a)}{7}+\frac{2}{7} & \frac{3}{7}-\frac{\sigma_4}{14}-\sigma_3\\ \frac{3}{14}-\frac{\sigma_4}{7}-\sigma_2 & \frac{\sigma_4}{14}-\sigma_3+\frac{3}{7} & \frac{5\cos(a)}{14}+\frac{9}{14}\end{pmatrix}$$

where

$$\sigma_1=\frac{3\sqrt{14}\sin(a)}{14}$$

$$\sigma_2=\frac{3\cos(a)}{14}$$

$$\sigma_3=\frac{3\cos(a)}{7}$$

$$\sigma_4=\sqrt{14}\sin(a)$$

Example 2

The return type coincides with the type of the input parameter representing the axis:

```
numeric::rotationMatrix(0.3, matrix([1,2,3]))
```

$$\begin{pmatrix} 0.9585267399 & -0.2305627908 & 0.1675329472 \\ 0.2433237939 & 0.9680974922 & -0.05983959278 \\ -0.1483914426 & 0.0981226021 & 0.9840487461 \end{pmatrix}$$

```
domtype(%)
```

Dom::Matrix()

```
numeric::rotationMatrix(0.3, hfarray(1..3, [1,2,3]))
```

$$\begin{pmatrix} 0.9585267399 & -0.2305627908 & 0.1675329472 \\ 0.2433237939 & 0.9680974922 & -0.05983959278 \\ -0.1483914426 & 0.0981226021 & 0.9840487461 \end{pmatrix}$$

```
domtype(%)
```

DOM_HFARRAY

The option `ReturnType` allows to specify the type of the result:

```
numeric::rotationMatrix(0.3, hfarray(1..3, [1,2,3]),
                                ReturnType = matrix)
```

$$\begin{pmatrix} 0.9585267399 & -0.2305627908 & 0.1675329472 \\ 0.2433237939 & 0.9680974922 & -0.05983959278 \\ -0.1483914426 & 0.0981226021 & 0.9840487461 \end{pmatrix}$$

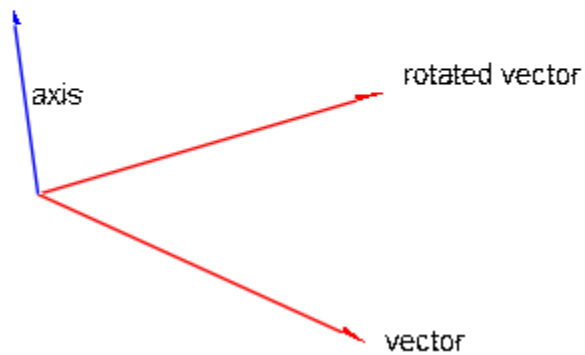
```
domtype(%)
```

Dom::Matrix()

Example 3

The direction of the rotation is given by the “right hand rule”: Stretch the thumb of your right hand and bend the fingers. When the thumb points into the direction of the rotation axis, your finger tips indicate the direction of the rotation:

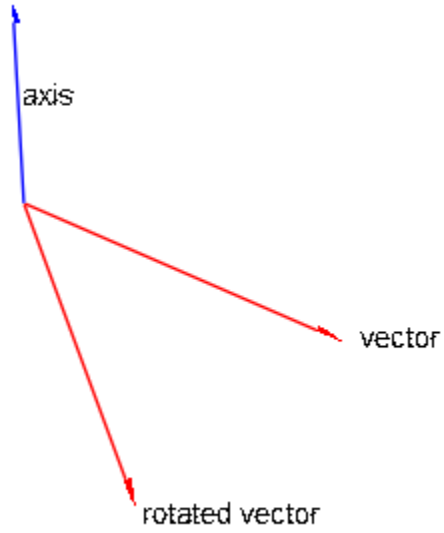
```
axis := matrix([0, 0, 1]):
vector := matrix([1, 0, 0]):
Q := numeric::rotationMatrix(PI/4, axis):
plot(plot::Arrow3d(axis, Color = RGB::Blue),
     plot::Arrow3d(vector, Color = RGB::Red),
     plot::Arrow3d(Q*vector, Color = RGB::Red),
     plot::Text3d("axis", [0.01, 0.01, 0.5]),
     plot::Text3d("vector", [1.05, 0, 0]),
     plot::Text3d("rotated vector", [0.75, 0.75, 0]),
     CameraDirection = [1, -2, 4],
     Scaling = Constrained, Axes = None):
```



Use negative angles to rotate in the opposite direction:

```
axis := matrix([0, 0, 1]):
vector := matrix([1, 0, 0]):
```

```
Q := numeric::rotationMatrix(-PI/4, axis):
plot(plot::Arrow3d(axis, Color = RGB::Blue),
     plot::Arrow3d(vector, Color = RGB::Red),
     plot::Arrow3d(Q*vector, Color = RGB::Red),
     plot::Text3d("axis", [0.01, 0.01, 0.5]),
     plot::Text3d("vector", [1.05, 0, 0]),
     plot::Text3d("rotated vector", [0.75, -0.75, 0]),
     CameraDirection = [1, -2, 4],
     Scaling = Constrained, Axes = None):
```




```
delete axis, vector, Q:
```

Parameters

`angle`

An arithmetical expression

`axis`

A vector represented by a list with 3 entries or by a 3 ×1 matrix of domain type `DOM_ARRAY`, `DOM_HFARRAY`, or of category `Cat::Matrix`

Options

`Symbolic`

Prevents the conversion of the input data to floating-point numbers. Exact arithmetic is used.

`ReturnType`

Option, specified as `ReturnType = t`

Return the result as a matrix of domain type `t`. The following return types are available: `DOM_ARRAY`, `DOM_HFARRAY`, `matrix`, or `densematrix`.

Return Values

Depending on the type of the input matrix `axis`, the 3×3 rotation matrix is returned as a matrix of domain type `DOM_ARRAY`, `DOM_HFARRAY`, `Dom::Matrix()`, or `Dom::DenseMatrix()`.

numeric::singularvalues

Numerical singular values of a matrix

Syntax

```
numeric::singularvalues(A, <Hard | HardwareFloats | Soft | SoftwareFloats>)
```

Description

`numeric::singularvalues(A)` returns numerical singular values of the matrix `A`.

The singular values of an $m \times n$ matrix A are the $p = min(m, n)$ real nonnegative square roots of the eigenvalues of $A^H A$ (for $p = n$) or of $A A^H$ (for $p = m$). The Hermitian transpose A^H is the complex conjugate of the transpose of A.

`numeric::singularvalues` returns a list of real singular values $[d_1, \ldots, d_p]$ sorted by `numeric::sort`, i.e., $d_1 \geq \ldots \geq d_p \geq 0.0$.

All entries of `A` must be numerical. Numerical expressions such as e^{π}, $\sqrt{2}$ etc. are accepted and converted to floats. Non-numerical symbolic entries lead to an error.

`Cat::Matrix` objects, i.e., matrices `A` of a matrix domain such as `Dom::Matrix(…)` or `Dom::SquareMatrix(…)` are internally converted to arrays over expressions via `expr(A)`.

Note: Singular values are approximated with an *absolute* precision of $\frac{1}{10^{\mathrm{DIGITS}}}\, r$ where r is largest singular value of `A`. Consequently, large singular values should be computed correctly to `DIGITS` decimal places. The numerical approximations of small singular values are less accurate.

Singular values may also be computed via `map ( numeric::eigenvalues( A  A^H), sqrt )` or `map ( numeric::eigenvalues( A^H  A ), sqrt )`, respectively. The use of `numeric::singularvalues` avoids the costs of the matrix multiplication. Further,

the eigenvalue routine requires about twice as many DIGITS to compute small singular values with the same precision as numeric::singularvalues. Cf. “Example 2” on page 19-318.

Environment Interactions

The function is sensitive to the environment variable DIGITS, which determines the numerical working precision.

Examples

Example 1

The singular values of A and A^H coincide:

```
A := array(1..3, 1..2, [[1, 2*I], [2, 3],[3, sqrt(2)]]):
numeric::singularvalues(A)
```

```
[4.994802008, 2.012946323]
```

The Hermitian transpose $B = A^H$:

```
B := array(1..2, 1..3, [[1, 2, 3], [-2*I, 3, sqrt(2)]]):
numeric::singularvalues(B)
```

```
[4.994802008, 2.012946323]
```

```
delete A, B:
```

Example 2

We use numeric::eigenvalues to compute singular values:

```
A := matrix([[1/15, 2/15*I],
             [PI, 314159265358980/50000000000000*I],
             [2, 4*I]]):
```

The Hermitian transpose $B = A^H$ can be computed by the methods `conjugate` and `transpose` of the matrix domain:

```
B := A::dom::conjugate(A::dom::transpose(A)):
```

Note that $A^H A$ is positive semi-definite and cannot have negative eigenvalues. However, computing small eigenvalues is numerically ill-conditioned, and a small negative value occurs due to roundoff:

```
numeric::eigenvalues(B*A)
```

$$\left[69.37024423,\ -2.310493271\ 10^{-15}\right]$$

Consequently, an (incorrect) imaginary singular value is computed:

```
map(%, sqrt)
```

$$[8.32888013,\ 0.00000004806759065\ i]$$

We have to increase `DIGITS` in order to compute this value more accurately:

```
DIGITS := 22:
map(numeric::eigenvalues(B*A), sqrt)
```

$$\left[8.328880130465871055353,\ 3.293453726225542754196\ 10^{-15}\right]$$

With `numeric::singularvalues`, the standard precision suffices:

```
DIGITS := 10:
numeric::singularvalues(A, SoftwareFloats)
```

$$\left[8.32888013,\ 3.249076763\ 10^{-15}\right]$$

```
numeric::singularvalues(A, HardwareFloats)
```

$$\left[8.32888013,\ 3.6920369\ 10^{-15}\right]$$

```
delete A, B:
```

Example 3

We demonstrate the use of hardware floats. Hilbert matrices are notoriously ill-conditioned: the computation of the small singular values is subject to severe roundoff effects. In the following results, both with `HardwareFloats` as well as with `SoftwareFloats`, the small singular values are dominated by numerical roundoff. Consequently, the results with `HardwareFloats` differ from those with with `SoftwareFloats`:

```
numeric::singularvalues(linalg::hilbert(13))
```

$$\left[1.813830119,\ 0.396833076,\ \ldots,\ 8.878210699\ 10^{-16},\ 3.222901015\ 10^{-18}\right]$$

[1.813830119, 0.396833076, 0.04902941942, 0.004348755075, 0.0002951777135, 0.0000156237036, 0.0000006466418563, 0.00000002076321421, 0.000000005076551851, 9.141268657e-12, 1.143562234e-13, 8.877867351e-16, 7.878607674e-19]

```
A := linalg::hilbert(15):
numeric::singularvalues(A, HardwareFloats);
numeric::singularvalues(A, SoftwareFloats)
```

$$\left[1.845927746,\ 0.426627957,\ 0.05721209253,\ \ldots,\ 1.351581189\ 10^{-17},\ 5.696479572\ 10^{-18}\right]$$

$$\left[1.845927746,\ 0.426627957,\ 0.05721209253,\ \ldots,\ 9.682265893\ 10^{-19},\ 3.017915363\ 10^{-21}\right]$$

[1.845927746, 0.426627957, 0.05721209253, 0.005639834756, 0.0004364765944, 0.00002710853923, 0.000001361582242, 0.00000005528988481, 0.000000001802959758,

4.657785895e-11, 9.321516341e-13, 1.386205079e-14, 1.463931556e-16, 1.249693852e-17, 6.620874158e-18] [1.845927746, 0.426627957, 0.05721209253, 0.005639834756, 0.0004364765944, 0.00002710853923, 0.000001361582242, 0.00000005528988482, 0.000000001802959751, 4.657786546e-11, 9.321608034e-13, 1.39406179e-14, 1.46659771e-16, 9.68226589e-19, 3.017915362e-21]

```
delete A:
```

Parameters

A

A numerical matrix of domain type DOM_ARRAY, DOM_HFARRAY, or of category Cat::Matrix

Options

Hard, HardwareFloats, Soft, SoftwareFloats

With Hard (or HardwareFloats), computations are done using fast hardware float arithmetic from within a MuPAD session. Hard and HardwareFloats are equivalent. With this option, the input data are converted to hardware floats and processed by compiled C code. The result is reconverted to MuPAD floats and returned to the MuPAD session.

With Soft (or SoftwareFloats) computations are dome using software float arithmetic provided by the MuPAD kernel. Soft and SoftwareFloats are equivalent. SoftwareFloats is used by default if the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY.

Compared to the SoftwareFloats used by the MuPAD kernel, the computation with HardwareFloats may be many times faster. Note, however, that the precision of hardware arithmetic is limited to about 15 digits. Further, the size of floating-point numbers may not be larger than approximately 10^{308} and not smaller than approximately 10^{-308}.

If no HardwareFloats or SoftwareFloats are requested explicitly, the following strategy is used: If the current value of DIGITS is smaller than 16 or if the matrix A is a

hardware float array of domain type DOM_HFARRAY, then hardware arithmetic is tried. If this is successful, the result is returned.

If the result cannot be computed with hardware floats, software arithmetic by the MuPAD kernel is tried.

If the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY, or if one of the options Soft, SoftwareFloats or Symbolic is specified, MuPAD computes the result with its software arithmetic without trying to use hardware floats first.

There may be several reasons for hardware arithmetic to fail:

- The current value of DIGITS is larger than 15.
- The data contains symbolic objects.
- The data contains numbers larger than 10^{308} or smaller than 10^{-308} that cannot be represented by hardware floats.

If neither HardwareFloats nor SoftwareFloats is specified, the user is not informed whether hardware floats or software floats are used.

If HardwareFloats are specified but fail due to one of the reasons above, a warning is issued that the (much slower) software floating-point arithmetic of the MuPAD kernel is used.

Note that HardwareFloats can only be used if all input data can be converted to floating-point numbers.

The trailing digits in floating-point results computed with HardwareFloats and SoftwareFloats may differ.

Note: For ill-conditioned matrices, the result is subject to roundoff errors. The results returned with HardwareFloats and SoftwareFloats may differ! See “Example 3” on page 19-320.

Return Values

Ordered list of real floating-point values.

Algorithms

The code implements standard numerical algorithms from the Handbook of Automatic Computation by Wilkinson and Reinsch.

See Also

MuPAD Functions

`linalg::eigenvalues` | `linalg::eigenvectors` | `numeric::eigenvalues` | `numeric::eigenvectors` | `numeric::singularvectors` | `numeric::spectralradius`

numeric::singularvectors

Numerical singular value decomposition of a matrix

Syntax

```
numeric::singularvectors(A, options)
```

Description

`numeric::singularvectors(A)` and the equivalent call `numeric::svd(A)` return numerical singular values and singular vectors of the matrix `A`.

All entries of `A` must be numerical. Numerical expressions such as e^{π}, $\sqrt{2}$ etc. are accepted and converted to floats. Non-numerical symbolic entries lead to an error.

`Cat::Matrix` objects, i.e., matrices `A` of a matrix domain such as `Dom::Matrix(…)` or `Dom::SquareMatrix(…)` are internally converted to arrays over expressions via `expr(A)`.

The list `[U, d, V, res`$_U$`, res`$_V$`]` returned by `numeric::singularvectors` corresponds to the singular data of an $m \times n$ matrix A as described below.

Let V^H denote the Hermitian transpose of the matrix V, i.e., the complex conjugate of the transpose. The singular value decomposition of an $m \times n$ matrix A is a factorization $A = U\,D\,V^H$. D is an $m \times n$ "diagonal" matrix with real nonnegative entries $D_{ii} = d_i$, $i = 1, \ldots, p$ where $p = min(m, n)$:

$$D = \begin{pmatrix} d_1 & & 0 \\ & \cdots & \\ & & d_p \\ 0 & & 0 \end{pmatrix}$$

or

$$D = \begin{pmatrix} d_1 & & 0 \\ & \dots & \\ 0 & & d_p \; 0 \end{pmatrix},$$

respectively. The list $d = [d_1, \dots, d_p]$ returned by `numeric::singularvectors` are the "singular values" of A. They are sorted by `numeric::sort`, i.e., $d_1 \geq \dots \geq d_p \geq 0.0$.

U is a unitary $m \times m$ matrix. Its i-th column is an eigenvector of $A\,A^H$ associated with the eigenvalue d_i^2 ($d_i = 0$ for $i > p$). These are the "left singular vectors" of A. They are returned by `numeric::singularvectors` as a matrix of floating-point numbers.

V is a unitary $n \times n$ matrix. Its i-th column is an eigenvector of $A^H\,A$ associated with the eigenvalue d_i^2 ($d_i = 0$ for $i > p$). These are the "right singular vectors" of A. They are returned by `numeric::singularvectors` as an array of floating-point numbers. The matrix `V` is normalized such that, in each column, the first entry of absolute size larger than $\frac{1}{10^{\mathrm{DIGITS}}}$ is real and positive.

If no return type is specified via the option `ReturnType = t`, the domain type of the singular vectors U and V depends on the type of the input matrix `A`:

- The singular vectors of an array are returned as arrays.
- The singular vectors of an hfarray are returned as hfarrays.
- The singular vectors of a dense matrix of type `Dom::DenseMatrix()` are returned as dense matrices of type `Dom::DenseMatrix()` over the ring of MuPAD expressions.
- For all other matrices of category `Cat::Matrix`, the singular vectors are returned as matrices of type `Dom::Matrix()` over the ring of MuPAD expressions. This includes input matrices `A` of type `Dom::Matrix(…)`, `Dom::SquareMatrix(…)`, `Dom::MatrixGroup(…)` etc.

$res_U = [resU_1, \dots, resU_m]$ is a list of float residues associated with the left singular vectors:

$$\mathrm{resU}_i = \left\langle A^H\,u_i,\; A^H\,u_i \right\rangle - d_i^2,\; 1 \leq i \leq m.$$

Here, u_i is the (normalized) i-th column of `U`, $\langle \dots, \dots \rangle$ is the usual complex Euclidean scalar product and $d_i = 0$ for $p < i \leq m$.

$res_V = [resV_1, \ldots, resV_n]$ is a list of float residues associated with the right singular vectors:

$$\mathrm{resV}_i = \langle A\,v_p, A\,v_i \rangle - d_i^2,\ 1 \le i \le n$$

.

Here, v_i is the (normalized) i-th column of `V`, $d_i = 0$ for $p < i \le n$.

The residues res_U, res_V vanish for exact singular data U, d, V. Their sizes indicate the quality of the numerical data `U, d, V`.

Note: Singular values are approximated with an *absolute* precision of $\frac{1}{10^{\mathrm{DIGITS}}}\,r$, where r is the largest singular value of `A`. Consequently, large singular values should be computed correctly to `DIGITS` decimal places. The numerical approximations of small singular values are less accurate.

The singular values computed by `numeric::singularvectors` are identical to those computed by `numeric::svd`.

Singular data may also be computed via `[d2, U, resU] := numeric::eigenvectors(A*A^H)` or `[d2, V, resV] := numeric::eigenvectors(A^H*A)`, respectively. The list `d2` is related to the singular values by

$$\mathrm{d2} = \left[d_1^2, d_2^2, \ldots, d_p^2, 0, \ldots, 0\right]$$

.

The use of `numeric::singularvectors` avoids the costs of the matrix multiplication. Further, the eigenvector routine requires about twice as many `DIGITS` to compute the data associated with small singular values with the same precision as `numeric::singularvectors`. Also note that the normalization of `U` and `V` may be different.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

Numerical expressions are converted to floats:

```
DIGITS := 5:
A := array(1..3, 1..2, [[1, PI], [2, 3], [3, exp(sqrt(2))]]):
[U, d, V, resU, resV] := numeric::singularvectors(A):
```

The singular data are:

```
U, d, V
```

$$\begin{pmatrix} 0.45729 & -0.88078 & 0.12293 \\ 0.51483 & 0.14947 & -0.84416 \\ 0.72515 & 0.44932 & 0.5218 \end{pmatrix}, [6.9986, 0.89905], \begin{pmatrix} 0.5233 & 0.85215 \\ 0.85215 & -0.5233 \end{pmatrix}$$

The small residues indicate that these results are not severely affected by roundoff:

```
resU, resV
```

$$\left[2.247\ 10^{-14}, 2.247\ 10^{-14}, 3.898\ 10^{-32}\right], \left[5.6175\ 10^{-15}, 5.6175\ 10^{-15}\right]$$

```
delete DIGITS, A, U, d, V, resU, resV:
```

Example 2

We demonstrate how to reconstruct a matrix from its singular data. With the specified `ReturnType`, the singular vectors are returned as matrices of type `Dom::Matrix()` and can be handled with the overloaded arithmetic:

```
DIGITS := 3:
A := array(1..2, 1..3, [[1.0, I, PI], [2, 3, I]]):
[U, d, V, resU, resV] := numeric::singularvectors(A, NoResidues,
     ReturnType = Dom::Matrix())
```

$$\left[\begin{pmatrix} 0.514 - 0.028\,i & -0.788 + 0.336\,i \\ 0.857 + 0.014\,i & 0.487 - 0.168\,i \end{pmatrix}, [3.9,\ 3.27],\right.$$

$$\left.\begin{pmatrix} 0.571 & 0.0568 & 0.819 \\ 0.652 - 0.121\,i & 0.55 + 0.0871\,i & -0.493 + 0.0785\,i \\ 0.418 - 0.242\,i & -0.81 + 0.174\,i & -0.236 + 0.157\,i \end{pmatrix}, \text{NIL}, \text{NIL}\right]$$

A "diagonal" matrix is built from the singular values:

```
d := matrix(2, 3, d, Diagonal)
```

$$\begin{pmatrix} 3.9 & 0 & 0 \\ 0 & 3.27 & 0 \end{pmatrix}$$

We use the methods `conjugate` and `transpose` of the matrix domain to compute the Hermitian transpose of `V` and reconstruct `A`. Numerical roundoff is eliminated via `numeric::complexRound`:

```
VH := V::dom::conjugate(V::dom::transpose(V)):
map(U*d*VH, numeric::complexRound)
```

$$\begin{pmatrix} 1.0 & 1.0\,i & 3.14 \\ 2.0 & 3.0 & 1.0\,i \end{pmatrix}$$

```
delete DIGITS, A, U, d, V, resU, resV, VH:
```

Example 3

We demonstrate the use of hardware floats. The following matrix A is degenerate: it has rank 1. For the double eigenvalue 0 of the matrix $A^H A$, different base vectors of the corresponding eigenspace are returned with `HardwareFloats` and `SoftwareFloats`, respectively:

```
A := array(1..2, 1..3, [[1, 2, 3], [30, 60, 90]]):
[U1, d1, V1, resU1, resV1] :=
      numeric::singularvectors(A, HardwareFloats):
[U2, d2, V2, resU2, resV2] :=
```

```
      numeric::singularvectors(A, SoftwareFloats):
V1, V2
```

$$\begin{pmatrix} 0.2672612419 & 0.5345224838 & 0.8017837257 \\ 0.5345224838 & -0.7745419206 & 0.3381871191 \\ 0.8017837257 & 0.3381871191 & -0.4927193213 \end{pmatrix},$$

$$\begin{pmatrix} 0.2672612419 & 0.9561828875 & 0.1195228609 \\ 0.5345224838 & -0.04390192219 & -0.8440132318 \\ 0.8017837257 & -0.289459681 & 0.5228345342 \end{pmatrix}$$

```
delete A, U1, d1, V1, resU1, resV1, U2, d2, V2, resU2, resV2:
```

Parameters

A

A numerical matrix of domain type DOM_ARRAY, DOM_HFARRAY, or of category Cat::Matrix.

Options

Hard, HardwareFloats, Soft, SoftwareFloats

With Hard (or HardwareFloats), computations are done using fast hardware float arithmetic from within a MuPAD session. Hard and HardwareFloats are equivalent. With this option, the input data are converted to hardware floats and processed by compiled C code. The result is reconverted to MuPAD floats and returned to the MuPAD session.

With Soft (or SoftwareFloats) computations are dome using software float arithmetic provided by the MuPAD kernel. Soft and SoftwareFloats are equivalent. SoftwareFloats is used by default if the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY.

Compared to the SoftwareFloats used by the MuPAD kernel, the computation with HardwareFloats may be many times faster. Note, however, that the precision of

hardware arithmetic is limited to about 15 digits. Further, the size of floating-point numbers may not be larger than approximately 10^{308} and not smaller than approximately 10^{-308}.

If no `HardwareFloats` or `SoftwareFloats` are requested explicitly, the following strategy is used: If the current value of `DIGITS` is smaller than 16 or if the matrix `A` is a hardware float array of domain type `DOM_HFARRAY`, then hardware arithmetic is tried. If this is successful, the result is returned.

If the result cannot be computed with hardware floats, software arithmetic by the MuPAD kernel is tried.

If the current value of `DIGITS` is larger than 15 and the input matrix `A` is not of domain type `DOM_HFARRAY`, or if one of the options `Soft`, `SoftwareFloats` or `Symbolic` is specified, MuPAD computes the result with its software arithmetic without trying to use hardware floats first.

There may be several reasons for hardware arithmetic to fail:

- The current value of `DIGITS` is larger than 15.
- The data contains symbolic objects.
- The data contains numbers larger than 10^{308} or smaller than 10^{-308} that cannot be represented by hardware floats.

If neither `HardwareFloats` nor `SoftwareFloats` is specified, the user is not informed whether hardware floats or software floats are used.

If `HardwareFloats` are specified but fail due to one of the reasons above, a warning is issued that the (much slower) software floating-point arithmetic of the MuPAD kernel is used.

Note that `HardwareFloats` can only be used if all input data can be converted to floating-point numbers.

The trailing digits in floating-point results computed with `HardwareFloats` and `SoftwareFloats` may differ.

Note: For ill-conditioned matrices, the result is subject to roundoff errors. The results returned with `HardwareFloats` and `SoftwareFloats` may differ! See “Example 3” on page 19-328.

NoLeftVectors

Suppresses the computation of left singular vectors

If only right singular vectors are required, this option may be used to suppress the computation of `U` and the corresponding residues $\texttt{res}_{\texttt{U}}$. The return values for these data are `NIL`.

Depending on the size of `U`, this option may speed up the computation considerably.

NoRightVectors

Suppresses the computation of right singular vectors

If only left singular vectors are required, this option may be used to suppress the computation of `V` and the corresponding residues $\texttt{res}_{\texttt{V}}$. The return values for these data are `NIL`.

Depending on the size of `V`, this option may speed up the computation considerably.

NoResidues

Suppresses the computation of error estimates

If no error estimates are required, this option may be used to suppress the computation of the residues $\texttt{res}_{\texttt{U}}$ and $\texttt{res}_{\texttt{V}}$. The return values for these data are `NIL`.

The alternative option name `NoErrors` used in previous MuPAD versions is still available.

ReturnType

Option, specified as `ReturnType = t`

Return the left and right singular vectors as matrices of domain type `t`. The following return types *t* are available: `DOM_ARRAY`, or `DOM_HFARRAY`, or `Dom::Matrix()`, or `Dom::DenseMatrix()`.

This option determines the domain type of the matrices containing the singular vectors.

NoWarning

Suppresses warnings

Return Values

List `[U, d, V, res_U, res_V]`. `U` is a unitary square float matrix whose columns are left singular vectors. The list `d` contains the singular values. `V` is a unitary square float matrix whose columns are right singular vectors. The lists of float residues $\texttt{res}_U$ and $\texttt{res}_V$ provide error estimates for the numerical data.

See Also

MuPAD Functions

`linalg::eigenvalues` | `linalg::eigenvectors` | `numeric::eigenvalues` | `numeric::eigenvectors` | `numeric::singularvalues` | `numeric::spectralradius` | `numeric::svd`

numeric::svd

Numerical singular value decomposition of a matrix

Syntax

```
numeric::svd(A, options)
```

Description

`numeric::svd(A)` and the equivalent call `numeric::singularvectors(A)` return numerical singular values and singular vectors of the matrix `A`.

All entries of `A` must be numerical. Numerical expressions such as e^{π}, $\sqrt{2}$ etc. are accepted and converted to floats. Non-numerical symbolic entries lead to an error.

`Cat::Matrix` objects, i.e., matrices `A` of a matrix domain such as `Dom::Matrix(…)` or `Dom::SquareMatrix(…)` are internally converted to arrays over expressions via `expr(A)`.

The list `[U, d, V, res`$_U$`, res`$_V$`]` returned by `numeric::svd` corresponds to the singular data of an $m \times n$ matrix A as described below.

Let V^H denote the Hermitian transpose of the matrix V, i.e., the complex conjugate of the transpose. The singular value decomposition of an $m \times n$ matrix A is a factorization $A = U\,D\,V^H$. D is an $m \times n$ "diagonal" matrix with real nonnegative entries $D_{ii} = d_i$, $i = 1, \ldots, p$ where $p = min(m, n)$:

$$D = \begin{pmatrix} d_1 & & 0 \\ & \cdots & \\ & & d_p \\ 0 & & 0 \end{pmatrix}$$

or

$$D = \begin{pmatrix} d_1 & & 0 \\ & \dots & \\ 0 & & d_p \; 0 \end{pmatrix},$$

respectively. The list $d = [d_1, \dots, d_p]$ returned by `numeric::svd` are the "singular values" of A. They are sorted by `numeric::sort`, i.e., $d_1 \geq \dots \geq d_p \geq 0.0$.

U is a unitary $m \times m$ matrix. Its i-th column is an eigenvector of $A\,A^H$ associated with the eigenvalue d_i^2 ($d_i = 0$ for $i > p$). These are the "left singular vectors" of A. They are returned by `numeric::svd` as a matrix of floating-point numbers.

V is a unitary $n \times n$ matrix. Its i-th column is an eigenvector of $A^H\,A$ associated with the eigenvalue d_i^2 ($d_i = 0$ for $i > p$). These are the "right singular vectors" of A. They are returned by `numeric::svd` as an array of floating-point numbers. The matrix `V` is normalized such that, in each column, the first entry of absolute size larger than $\frac{1}{10^{\text{DIGITS}}}$ is real and positive.

If no return type is specified via the option `ReturnType = t`, the domain type of the singular vectors U and V depends on the type of the input matrix `A`:

- The singular vectors of an array are returned as arrays.
- The singular vectors of an hfarray are returned as hfarrays.
- The singular vectors of a dense matrix of type `Dom::DenseMatrix()` are returned as dense matrices of type `Dom::DenseMatrix()` over the ring of MuPAD expressions.
- For all other matrices of category `Cat::Matrix`, the singular vectors are returned as matrices of type `Dom::Matrix()` over the ring of MuPAD expressions. This includes input matrices `A` of type `Dom::Matrix(…)`, `Dom::SquareMatrix(…)`, `Dom::MatrixGroup(…)` etc.

$res_U = [resU_1, \dots, resU_m]$ is a list of float residues associated with the left singular vectors:

$$\text{resU}_i = \left\langle A^H\,u_i,\, A^H\,u_i \right\rangle - d_i^2,\; 1 \leq i \leq m.$$

Here, u_i is the (normalized) i-th column of `U`, $\langle \dots, \dots \rangle$ is the usual complex Euclidean scalar product and $d_i = 0$ for $p < i \leq m$.

$res_V = [resV_1, \ldots, resV_n]$ is a list of float residues associated with the right singular vectors:

$$\mathrm{resV}_i = \langle A\,v_p, A\,v_i\rangle - d_i^2,\ 1 \le i \le n$$

.

Here, v_i is the (normalized) i-th column of `V`, $d_i = 0$ for $p < i \le n$.

The residues res_U, res_V vanish for exact singular data U, d, V. Their sizes indicate the quality of the numerical data `U, d, V`.

Note: Singular values are approximated with an *absolute* precision of $\frac{1}{10^{\mathrm{DIGITS}}}\,r$, where r is the largest singular value of `A`. Consequently, large singular values should be computed correctly to `DIGITS` decimal places. The numerical approximations of small singular values are less accurate.

The singular values computed by `numeric::svd` are identical to those computed by `numeric::singularvalues`.

Singular data may also be computed via `[d2, U, resU] := numeric::eigenvectors(A*A^H)` or `[d2, V, resV] := numeric::eigenvectors(A^H*A)`, respectively. The list `d2` is related to the singular values by

$$\mathrm{d2} = \left[d_1^2, d_2^2, \ldots, d_p^2, 0, \ldots, 0\right]$$

.

The use of `numeric::svd` avoids the costs of the matrix multiplication. Further, the eigenvector routine requires about twice as many `DIGITS` to compute the data associated with small singular values with the same precision as `numeric::singularvectors`. Also note that the normalization of `U` and `V` may be different.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

Numerical expressions are converted to floats:

```
DIGITS := 5:
A := array(1..3, 1..2, [[1, PI], [2, 3], [3, exp(sqrt(2))]]):
[U, d, V, resU, resV] := numeric::singularvectors(A):
```

The singular data are:

```
U, d, V
```

$$\begin{pmatrix} 0.45729 & -0.88078 & 0.12293 \\ 0.51483 & 0.14947 & -0.84416 \\ 0.72515 & 0.44932 & 0.5218 \end{pmatrix}, [6.9986, 0.89905], \begin{pmatrix} 0.5233 & 0.85215 \\ 0.85215 & -0.5233 \end{pmatrix}$$

The small residues indicate that these results are not severely affected by roundoff:

```
resU, resV
```

$$\left[2.247\ 10^{-14}, 2.247\ 10^{-14}, 3.898\ 10^{-32}\right], \left[5.6175\ 10^{-15}, 5.6175\ 10^{-15}\right]$$

```
delete DIGITS, A, U, d, V, resU, resV:
```

Example 2

We demonstrate how to reconstruct a matrix from its singular data. With the specified `ReturnType`, the singular vectors are returned as matrices of type `Dom::Matrix()` and can be handled with the overloaded arithmetic:

```
DIGITS := 3:
A := array(1..2, 1..3, [[1.0, I, PI], [2, 3, I]]):
[U, d, V, resU, resV] := numeric::singularvectors(A, NoResidues,
     ReturnType = Dom::Matrix())
```

$$\left[\begin{pmatrix} 0.514 - 0.028\,i & -0.788 + 0.336\,i \\ 0.857 + 0.014\,i & 0.487 - 0.168\,i \end{pmatrix}, [3.9,\ 3.27],\right.$$

$$\left.\begin{pmatrix} 0.571 & 0.0568 & 0.819 \\ 0.652 - 0.121\,i & 0.55 + 0.0871\,i & -0.493 + 0.0785\,i \\ 0.418 - 0.242\,i & -0.81 + 0.174\,i & -0.236 + 0.157\,i \end{pmatrix}, \text{NIL}, \text{NIL}\right]$$

A "diagonal" matrix is built from the singular values:

```
d := matrix(2, 3, d, Diagonal)
```

$$\begin{pmatrix} 3.9 & 0 & 0 \\ 0 & 3.27 & 0 \end{pmatrix}$$

We use the methods `conjugate` and `transpose` of the matrix domain to compute the Hermitian transpose of `V` and reconstruct `A`. Numerical roundoff is eliminated via `numeric::complexRound`:

```
VH := V::dom::conjugate(V::dom::transpose(V)):
map(U*d*VH, numeric::complexRound)
```

$$\begin{pmatrix} 1.0 & 1.0\,i & 3.14 \\ 2.0 & 3.0 & 1.0\,i \end{pmatrix}$$

```
delete DIGITS, A, U, d, V, resU, resV, VH:
```

Example 3

We demonstrate the use of hardware floats. The following matrix A is degenerate: it has rank 1. For the double eigenvalue 0 of the matrix $A^H A$, different base vectors of the corresponding eigenspace are returned with `HardwareFloats` and `SoftwareFloats`, respectively:

```
A := array(1..2, 1..3, [[1, 2, 3], [30, 60, 90]]):
[U1, d1, V1, resU1, resV1] :=
      numeric::singularvectors(A, HardwareFloats):
[U2, d2, V2, resU2, resV2] :=
```

```
      numeric::singularvectors(A, SoftwareFloats):
V1, V2
```

$$\begin{pmatrix} 0.2672612419 & 0.5345224838 & 0.8017837257 \\ 0.5345224838 & -0.7745419206 & 0.3381871191 \\ 0.8017837257 & 0.3381871191 & -0.4927193213 \end{pmatrix},$$

$$\begin{pmatrix} 0.2672612419 & 0.9561828875 & 0.1195228609 \\ 0.5345224838 & -0.04390192219 & -0.8440132318 \\ 0.8017837257 & -0.289459681 & 0.5228345342 \end{pmatrix}$$

```
delete A, U1, d1, V1, resU1, resV1, U2, d2, V2, resU2, resV2:
```

Parameters

A

A numerical matrix of domain type DOM_ARRAY, DOM_HFARRAY, or of category Cat::Matrix.

Options

Hard, HardwareFloats, Soft, SoftwareFloats

With Hard (or HardwareFloats), computations are done using fast hardware float arithmetic from within a MuPAD session. Hard and HardwareFloats are equivalent. With this option, the input data are converted to hardware floats and processed by compiled C code. The result is reconverted to MuPAD floats and returned to the MuPAD session.

With Soft (or SoftwareFloats) computations are dome using software float arithmetic provided by the MuPAD kernel. Soft and SoftwareFloats are equivalent. SoftwareFloats is used by default if the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY.

Compared to the SoftwareFloats used by the MuPAD kernel, the computation with HardwareFloats may be many times faster. Note, however, that the precision of

hardware arithmetic is limited to about 15 digits. Further, the size of floating-point numbers may not be larger than approximately 10^{308} and not smaller than approximately 10^{-308}.

If no `HardwareFloats` or `SoftwareFloats` are requested explicitly, the following strategy is used: If the current value of `DIGITS` is smaller than 16 or if the matrix `A` is a hardware float array of domain type `DOM_HFARRAY`, then hardware arithmetic is tried. If this is successful, the result is returned.

If the result cannot be computed with hardware floats, software arithmetic by the MuPAD kernel is tried.

If the current value of `DIGITS` is larger than 15 and the input matrix `A` is not of domain type `DOM_HFARRAY`, or if one of the options `Soft`, `SoftwareFloats` or `Symbolic` is specified, MuPAD computes the result with its software arithmetic without trying to use hardware floats first.

There may be several reasons for hardware arithmetic to fail:

- The current value of `DIGITS` is larger than 15.
- The data contains symbolic objects.
- The data contains numbers larger than 10^{308} or smaller than 10^{-308} that cannot be represented by hardware floats.

If neither `HardwareFloats` nor `SoftwareFloats` is specified, the user is not informed whether hardware floats or software floats are used.

If `HardwareFloats` are specified but fail due to one of the reasons above, a warning is issued that the (much slower) software floating-point arithmetic of the MuPAD kernel is used.

Note that `HardwareFloats` can only be used if all input data can be converted to floating-point numbers.

The trailing digits in floating-point results computed with `HardwareFloats` and `SoftwareFloats` may differ.

Note: For ill-conditioned matrices, the result is subject to roundoff errors. The results returned with `HardwareFloats` and `SoftwareFloats` may differ! See “Example 3” on page 19-337.

NoLeftVectors

Suppresses the computation of left singular vectors

If only right singular vectors are required, this option may be used to suppress the computation of `U` and the corresponding residues $\texttt{res}_{\texttt{U}}$. The return values for these data are `NIL`.

Depending on the size of `U`, this option may speed up the computation considerably.

NoRightVectors

Suppresses the computation of right singular vectors

If only left singular vectors are required, this option may be used to suppress the computation of `V` and the corresponding residues $\texttt{res}_{\texttt{V}}$. The return values for these data are `NIL`.

Depending on the size of `V`, this option may speed up the computation considerably.

NoResidues

Suppresses the computation of error estimates

If no error estimates are required, this option may be used to suppress the computation of the residues $\texttt{res}_{\texttt{U}}$ and $\texttt{res}_{\texttt{V}}$. The return values for these data are `NIL`.

The alternative option name `NoErrors` used in previous MuPAD versions is still available.

ReturnType

Option, specified as `ReturnType = t`

Return the left and right singular vectors as matrices of domain type `t`. The following return types *t* are available: `DOM_ARRAY`, or `DOM_HFARRAY`, or `Dom::Matrix()`, or `Dom::DenseMatrix()`.

This option determines the domain type of the matrices containing the singular vectors.

NoWarning

Suppresses warnings

Return Values

List `[U, d, V, res_U, res_V]`. `U` is a unitary square float matrix whose columns are left singular vectors. The list `d` contains the singular values. `V` is a unitary square float matrix whose columns are right singular vectors. The lists of float residues $\texttt{res}_U$ and $\texttt{res}_V$ provide error estimates for the numerical data.

See Also

MuPAD Functions

`linalg::eigenvalues` | `linalg::eigenvectors` | `numeric::eigenvalues` | `numeric::eigenvectors` | `numeric::singularvalues` | `numeric::singularvectors` | `numeric::spectralradius`

numeric::solve

Numerical solution of equations (the float attribute of solve)

Syntax

```
numeric::solve(eqs, <vars>, options)

float(holdsolve(eqs, <vars>, options))

float(freezesolve(eqs, <vars>, options))
```

Description

`numeric::solve` computes numerical solutions of equations. For polynomial equations, *all* solutions are returned. For non-polynomial equations, only *one* solution, if any, is returned unless the option `AllRealRoots` is used.

Note: Note that only for polynomial/rational equations *all* solutions are searched for. For non-polynomial/non-rational equations, only one solution, if any, is returned unless the option `AllRealRoots` is used.

If the equations contain non-polynomial expressions, it is in general not possible to isolate *all* roots numerically. Think of equations such as $\sin\left(\frac{1}{x}\right) = 0$ that have infinitely many real solutions around the origin! If a complete set of *all* real solutions of a single non-polynomial/non-rational equation in one unknown is desired, you may try the opton `AllRealRoots`. With this option, a heuristics tries to isolate all real solutions of the equation. This, however, is purely heuristical: there is no rigor in the algorithm and it is not guaranteed that all solutions are found. Alternatively, you may also use the routine `numeric::realroots` to isolate the intervals in which solutions may exist.

`numeric::solve` is a simple interface function unifying the functionality of the numerical solvers `numeric::fsolve`, `numeric::linsolve`, `numeric::polyroots`, and `numeric::polysysroots`. The return format of these routines is changed to make it consistent with the return values of the symbolic solver `solve`.

You may call the specialized numerical solvers directly. However, note the return types specific to each of these solvers.

`numeric::solve` classifies the equations as follows:

- If `eqs` is a single univariate polynomial equation, then it is directly passed to `numeric::polyroots`. Cf. "Example 2" on page 19-345. The roots are returned as a set or as a `Dom::Multiset` if `Multiple` is used.
- If `eqs` is a multivariate polynomial equation or a list or set of such equations, then the equations and the appropriate optional arguments are passed to either `numeric::linsolve` or `numeric::polysysroots`. Cf. "Example 3" on page 19-345. The roots are returned as a set or as a `Dom::Multiset` if `Multiple` is used.
- A rational equation or a set or list of rational equations is replaced by its/their numerator(s). Such equations are processed like polynomial equations.
- If `eqs` is a non-polynomial/non-rational equation or a set or list containing such an equation, then the equations and the appropriate optional arguments are passed to the numerical solver `numeric::fsolve`.

Note: For non-polynomial equations, only a single numerical root is returned, unless `AllRealRoots` is specified! Cf. "Example 4" on page 19-346.

Note: For non-polynomial equations, there must not be more equations than unknowns!

Using `Multiple` for non-polynomial equations leads to an error, unless the option `AllRealRoots` is specified, too!

Note: For systems of multivariate non-polynomial equations, MuPAD uses a Newton search. It must be able to evaluate the partial derivatives of the equations with respect to the variables to be solved for.

For a single univariate equation, first a bisectioning scheme with quadratic interpolation is used that does not require any differentiation of the equation. If

this is not successful, a Newton search is started that requires the derivative of the functions involved.

For convenience, also polynomials of domain type `DOM_POLY` are accepted, wherever an equation is expected.

Note: In contrast to the symbolic solver `solve`, the numerical solver does not react to properties of identifiers set via `assume`. To use these properties, call `float ( hold( solve )(arguments))` instead.

If the user does not specify indeterminates to be solved for, then the indeterminates are internally chosen by `numeric::indets(eqs)`.

Starting points such as `x = a` or search ranges such as `x = a..b` specified in `vars` are ignored if `eqs` is a polynomial equation or a system of polynomial equations.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

The following three solver calls are equivalent:

```
eqs := {x^2 = sin(y), y^2 = cos(x)}:
numeric::solve(eqs, {x, y}),
float(hold(solve)(eqs, {x, y})),
float(freeze(solve)(eqs, {x,y}))
```

$$\{[x = -0.8517004887,\ y = 0.8116062151]\},\ \{[x = -0.8517004887,\ y = 0.8116062151]\},\ \{[x = -0.8517004887,\ y = 0.8116062151]\}$$

```
delete eqs:
```

Example 2

We demonstrate the root search for univariate polynomials:

```
numeric::solve(x^6 - PI*x^2 = sin(3), x)
```

$$\{-1.339589766,\ 1.339589766,\ 1.322706295\,i,\ 0.2120113223\,i,\ -0.2120113223\,i,\ -1.322706294\,i\}$$

Polynomials of type `DOM_POLY` can be used as input:

```
numeric::solve(poly((x - 1/3)^3, [x]), x)
```

$$\{0.3333333333\}$$

With `Multiple`, a `Dom::Multiset` is returned, indicating the multiplicity of the root:

```
numeric::solve(x^3 - x^2 + x/3 -1/27, x, Multiple)
```

$$\{[0.3333333333,\ 3]\}$$

Example 3

We demonstrate the root search for polynomial systems. Note that the symbolic solver `solve` is involved if the system is nonlinear. Symbolic parameters are accepted:

```
numeric::solve({x^2 + y^2 = 1, x^2 - y^2 = exp(z)}, {x, y})
```

$$\left\{\left[x = -0.7071067812\sqrt{e^z + 1.0},\ y = -0.7071067812\sqrt{1.0 - 1.0\,e^z}\right],\ \ldots,\ \left[x = 0.7071067812\sqrt{e^z + 1.0},\ y = 0.7071067812\sqrt{1.0 - 1.0\,e^z}\right]\right\}$$

{[x = 0.7071067812*sqrt(exp(z) + 1.0), y = -0.7071067812*sqrt(1.0 - 1.0*exp(z))], [x = 0.7071067812*sqrt(exp(z) + 1.0), y = 0.7071067812*sqrt(1.0 - 1.0*exp(z))], [x = -0.7071067812*sqrt(exp(z) + 1.0), y = 0.7071067812*sqrt(1.0 - 1.0*exp(z))], [x = -0.7071067812*sqrt(exp(z) + 1.0), y = -0.7071067812*sqrt(1.0 - 1.0*exp(z))]}

Example 4

We demonstrate the root search for non-polynomial equations. Without the option `AllRealRoots`, only one solution is searched for:

```
eq := exp(-x) - 10*x^2:
numeric::solve(eq, x)
```

$$\{0.2755302947\}$$

Since `numeric::solve` just calls the root finder `numeric::fsolve`, one may also use this routine directly. Note the different output format:

```
numeric::fsolve(eq, x)
```

$$[x = 0.2755302947]$$

The input syntax of `numeric::solve` and `numeric::fsolve` are identical, i.e., starting points, search ranges and options may be used. E.g., another solution of the previous equation is found by a restricted search over the interval $[-1, 0]$:

```
numeric::solve(eq, x = -1..0, RestrictedSearch)
```

$$\{-0.3829657727\}$$

We use the option `AllRealRoots` to isolate all real solutions of the equation:

```
numeric::solve(eq, x, AllRealRoots)
```

$$\{-5.827897796, -0.3829657727, 0.2755302947\}$$

With the following call we restrict the search to the negative semi-axis:

```
numeric::solve(eq, x = -infinity..0, AllRealRoots)
```

$$\{-5.827897796, -0.3829657727\}$$

Example 5

For the following system, `numeric::solve` finds the solution with positive y:

```
eqs := [exp(x) = 2*y^2, sin(y) = y*x^3]:
numeric::solve(eqs, [x, y])
```

$$\{[x = 0.9290711314,\ y = 1.125201325]\}$$

Another solution with negative y is found with an appropriate search range:

```
numeric::solve(eqs, [x = 1, y = -infinity..0])
```

$$\{[x = 0.9290711314,\ y = -1.125201325]\}$$

```
delete eq, eqs:
```

Parameters

eqs

An equation, a list, set, `array`, or `matrix` (`Cat::Matrix`) of equations. Also arithmetical expressions are accepted and interpreted as homogeneous equations.

vars

An unknown, a list of unknowns or a set of unknowns. Unknowns may be identifiers or indexed identifiers. Also equations of the form `x=a` or `x=a..b` are accepted wherever an unknown `x` is expected. This way, starting points and search ranges are specified for the numerical search. They must be numerical; infinite search ranges are accepted.

Options

AllRealRoots

Only to be used if `eqs` is a single equation in one unknown. With this option, a *heuristics* is used to find *all* real solutions of the equation.

Note: Note that there is no guarantee that all real solutions will be found.

Note: Interval arithmetic is used to isolate search intervals for the solutions. The expressions in `eqs` must be suitable for such arithmetic. Internally, the procedure `numeric::realroots` is called. See the help page of `numeric::realroots` for restrictions on the expressions in `eqs`.

Note: The equation must be suitable for evaluation with interval arithmetic. See `numeric::realroots` for restrictions on the expressions in the equation.

With `AllRealRoots`, only the additonal options `Multiple` and `NoWarning` have an effect. All other options such as `UnrestrictedSearch` etc. are ignored.

It is highly recommend to specify a search interval by a call such as `numeric::solve(f(x), x = a..b, AllRealRoots)`. In this case, only the real solutions between `a` and `b` are searched for.

The search for all real solutions may be very time consuming!

`Multiple`

Only to be used if `eqs` is a polynomial equation or a system of polynomial equations or in conjunction with the option `AllRealRoots`. With this option, information on the multiplicity of degenerate polynomial roots is returned.

It changes the return type from `DOM_SET` to `Dom::Multiset`.

`FixedPrecision`

Only to be used if `eqs` is a single univariate polynomial. It launches a quick numerical search with fixed internal precision.

It is passed to `numeric::polyroots`, which uses a numerical search with fixed internal precision. This is fast, but degenerate roots may be returned with a restricted precision. See the help page of `numeric::polyroots` for details.

`SquareFree`

Only to be used if `eqs` is a single univariate polynomial. Symbolic square free factorization is applied, before the numerical search starts.

It is passed to `numeric::polyroots`, which preprocesses the polynomial by a symbolic square free factorization. See the help page of `numeric::polyroots` for details.

`Factor`

Only to be used if `eqs` is a single univariate polynomial. Symbolic factorization is applied, before the numerical search starts.

It is passed to `numeric::polyroots`, which preprocesses the polynomial by a symbolic factorization. See the help page of `numeric::polyroots` for details.

`RestrictedSearch`

The numerical search is restricted to the search ranges specified in `vars`.

This option is passed to `numeric::fsolve`, which uses a corresponding search strategy when looking for roots in the search range specified in `vars`. It must be used only in conjunction with search range and only for non-polynomial equations.

See `numeric::fsolve` for details.

`UnrestrictedSearch`

The numerical search may return results outside the search ranges specified in `vars`.

This option is passed to `numeric::fsolve`, which uses a corresponding search strategy when looking for roots in the search range specified in `vars`. It must be use only in conjunction with search ranges and only for non-polynomial equations.

See `numeric::fsolve` for details.

`MultiSolutions`

Only to be used for non-polynomial equations in conjunction with `RestrictedSearch`. Several roots may be returned.

It is passed to `numeric::fsolve`, which returns a sequence of all roots found in the internal search. See the help page of `numeric::fsolve` for details.

`Random`

Only to be used for non-polynomial equations. With this option, several calls to `numeric::solve` may lead to different solutions of the equation(s).

It is passed to `numeric::fsolve` which switches to a random search strategy. See the help page of `numeric::fsolve` for details.

`NoWarning`

This option only has an effect when it is used for polynomial equations in conjunction with `AllRealRoots`. When you use `AllRealRoots`, warnings are issued if interval arithmetic indicates technical difficulties such as serious overestimation (for example, when encountering multiple roots). With this option, the warnings are suppressed.

Note: This option has an effect if `eqs` is a multivariate polynomial system or a univariate polynomial with a symbolic parameter.

In such a case, this option is passed to `numeric::polysysroots`.

Return Values

Set of numerical solutions. With the option `Multiple`, a set of domain type `Dom::Multiset` is returned.

See Also

MuPAD Functions

`isolate` | `linsolve` | `numeric::fsolve` | `numeric::linsolve` | `numeric::polyroots` | `numeric::polysysroots` | `numeric::realroot` | `numeric::realroots` | `polylib::realroots` | `solve`

More About

- "Solve Equations Numerically"

numeric::sort

Sort a numerical list

Syntax

```
numeric::sort(list)
```

Description

`numeric::sort(list)` sorts the elements in `list`.

The elements of the list are sorted such that their real parts are descending. Elements with the same real part are sorted from large absolute value to small absolute value. In case of a tie (i.e., two elements form a complex conjugate pair), the element with positive imaginary part comes first.

The elements of the list are converted to floating-point numbers via `float`. Elements that cannot be converted lead to an error.

This function is used to sort the return values of `numeric::eigenvalues`, `numeric::eigenvectors`, `numeric::polyroots`, `numeric::singularvalues`, and `numeric::singularvectors`.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`.

Examples

Example 1

The elements in the sorted list have descending real parts:

```
numeric::sort([1, 2.0, I, -3, -I, PI, sqrt(2)])
```

$$[3.141592654, 2.0, 1.414213562, 1.0, 1.0\,i, -1.0\,i, -3.0]$$

In the following example, the sorting criterion does not seem to be satisfied. Elements with the same real part are supposed to be ordered from large absolute values to small absolute values:

```
x := sin(PI/3):
L := numeric::sort([x, sin(float(PI/3)) - I, x + I])
```

$$[0.8660254038 + 1.0\,i, 0.8660254038 - 1.0\,i, 0.8660254038]$$

This is explained by the fact that the floating-point numbers internally have a more accurate representation than shown on the screen. The real part of the last element is indeed a little bit smaller than the other real parts:

```
DIGITS := 20:
L
```

$$[0.86602540378443864469 + 1.0\,i, 0.86602540378443864469 - 1.0\,i, 0.86602540378443864469]$$

```
delete x, L, DIGITS:
```

Parameters

`list`

A list of numbers or numerical expressions

Return Values

Sorted list.

See Also

MuPAD Functions

`sort`

numeric::spectralradius

Spectral radius of a matrix

Syntax

```
numeric::spectralradius(A, <x0>, <n>, <mode>, <ReturnType = t>, <NoWarning>)
```

Description

`numeric::spectralradius(A)` returns data corresponding to the eigenvalue of the matrix `A` that has the largest absolute value.

`numeric::spectralradius` and `numeric::spectralRadius` are equivalent.

The spectral radius of a matrix with eigenvalues λ_i is $max(|\lambda_i|)$.

The return value `lambda` is an approximation of the corresponding eigenvalue: `abs(lambda)` is the spectral radius.

The return value `x` is the corresponding normalized eigenvector: $||x||_2 = 1$.

The return value $\text{residue} = ||A\,x - \text{lambda}\,x||_2$ provides an error estimate for the eigenvalue. For Hermitian matrices this is a rigorous upper bound for the error $|lambda - \lambda_{exact}|$, where λ_{exact} is the exact eigenvalue.

`numeric::spectralradius` implements the power method to compute the eigenvalue and the associated eigenvector defining the spectral radius: the vector iteration $x_i = \frac{A^i x_0}{\|A^i x_0\|_2}$ "converges" towards the eigenspace associated with the spectral radius. The starting vector x_0 is provided by the second argument of `numeric::spectralradius`. If no starting vector is provided by the user, a randomly chosen vector is used.

Note: The iteration does not converge (converges slowly), if the spectral radius is generated by several distinct eigenvalues with the same (similar) absolute value.

Internally, the iteration stops, when the approximation of the eigenvalue becomes stationary within the relative precision given by `DIGITS`. If this does not happen within *n* iterations, then a warning is issued and the present values are returned. Cf. “Example 4” on page 19-357.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We let the routine choose a random starting vector:

```
A := matrix(2, 2, [[10, 1], [1, 20]]):
numeric::spectralradius(A)
```

$$\left[20.09901951,\ \begin{pmatrix} 0.0985376181 \\ 0.9951333267 \end{pmatrix},\ 0.000000001332123191\right]$$

We define a starting vector as a 1-dimensional array and allow a maximum of 1000 internal iterations:

```
A := array(1..2, 1..2, [[1, 2], [5, -10]]):
x0 := array(1..2, [1, 1]):
numeric::spectralradius(A, x0, 1000)
```

$$\left[-10.84428877,\ (\ 0.1665007247\ \ -0.9860413321\ ),\ 7.808576008\ 10^{-11}\right]$$

Next, we use a list to specify a starting vector:

```
A := array(1..2, 1..2, [[I, 3], [3, I]]):
numeric::spectralradius(A, [1, 1], 1000)
```

$$[3.0 + 1.0\,i,\ [0.7071067812,\ 0.7071067812],\ 0.0]$$

```
delete A, x0:
```

Example 2

With the default setting of `DIGITS` = 10, the following result is computed using `HardwareFloats`.

```
A := hfarray(1..2, 1..2, [[10^4, 10^4], [50,  60]]):
x0 := array(1..2, [1, 1]):
numeric::spectralradius(A, x0)
```

```
[10050.0498, ( 0.9999874753  0.00500491737 ), 0.000000004820940092]
```

We request `SoftwareFloats` in the next call. Note the difference in the trailing digits:

```
numeric::spectralradius(A, x0, Soft)
```

```
[10050.0498, ( 0.9999874753  0.00500491737 ), 0.000000004820942201]
```

```
delete DIGITS, A, x0:
```

Example 3

The eigenvector that is returned can have various types. If no starting vector is provided, the type of the matrix determines the type of the eigenvector:

```
A:= array(1..2, 1..2, [[1, 2], [3, 4]]):
[l, x, residue]:= numeric::spectralradius(A);
```

```
[5.372281324, ( 0.4159735579  0.9093767091 ), 0.0000000002881934691]
```

```
domtype(x)
```

```
DOM_ARRAY
```

```
A:= hfarray(1..2, 1..2, [[1, 2], [3, 4]]):
[l, x, residue]:= numeric::spectralradius(A):
domtype(x)
```

```
DOM_HFARRAY
```

```
A:= matrix(2, 2, [[1, 2], [3, 4]]):
[l, x, residue]:= numeric::spectralradius(A):
domtype(x)
```

```
Dom::Matrix()
```

If a starting vector is provided, its type determines the type of the return vector:

```
A:= hfarray(1..2, 1..2, [[1, 2], [3, 4]]):
x0:= [1, 1]:
[l, x, residue]:= numeric::spectralradius(A, x0):
domtype(x)
```

```
DOM_LIST
```

```
x0:= array(1..2, [1, 1]):
[l, x, residue]:= numeric::spectralradius(A, x0):
domtype(x)
```

```
DOM_ARRAY
```

```
x0:= hfarray(1..2, [1, 1]):
[l, x, residue]:= numeric::spectralradius(A, x0):
domtype(x)
```

```
DOM_HFARRAY
```

```
x0:= matrix([1, 1]):
[l, x, residue]:= numeric::spectralradius(A, x0):
domtype(x)
```

```
Dom::Matrix()
```

The return type can be requested explicitly:

```
[l, x, residue] :=
  numeric::spectralradius(A, x0, ReturnType = DOM_LIST):
domtype(x)
```

```
DOM_LIST
```

```
[l, x, residue] :=
  numeric::spectralradius(A, x0, ReturnType = DOM_HFARRAY):
domtype(x)
```

DOM_HFARRAY

```
delete A, x0, l, x, residue:
```

Example 4

The following matrix has two distinct eigenvalues 1 and -1 of the same absolute value. The power method must fail.

```
A := array(1..2, 1..2, [[1, 0], [0, -1]]):
```

We allow a maximum of 1000 internal steps. The call results in a warning. The large residue also indicates that the power method did not converge:

```
numeric::spectralradius(A, [1, 1], 1000)
```

```
Warning: There is no convergence of vector iteration. [numeric::spectralradius]
```

$$[1.0, [0.7071067812, -0.7071067812], 1.414213562]$$

```
delete A:
```

Parameters

A

An $m \times m$ array of domain type `DOM_ARRAY` or `DOM_HFARRAY` or a matrix of category `Cat::Matrix`

x_0

A starting vector: a 1-dimensional array, or an hfarray, or a list of length m. Also 2-dimensional arrays (`array(1..m, 1..1, ...)`, `hfarray(1..m, 1..1, ...)`) and matrices representing vectors are accepted.

n

The maximal number of iterations: a positive integer. The default value is 1000.

mode

One of the flags Hard, HardwareFloats, Soft, or SoftwareFloats

Options

Hard, HardwareFloats, Soft, SoftwareFloats

With Hard (or HardwareFloats), computations are done using fast hardware float arithmetic from within a MuPAD session. Hard and HardwareFloats are equivalent. With this option, the input data are converted to hardware floats and processed by compiled C code. The result is reconverted to MuPAD floats and returned to the MuPAD session.

With Soft (or SoftwareFloats) computations are dome using software float arithmetic provided by the MuPAD kernel. Soft and SoftwareFloats are equivalent. SoftwareFloats is used by default if the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY.

Compared to the SoftwareFloats used by the MuPAD kernel, the computation with HardwareFloats may be many times faster. Note, however, that the precision of hardware arithmetic is limited to about 15 digits. Further, the size of floating-point numbers may not be larger than approximately 10^{308} and not smaller than approximately 10^{-308}.

If no HardwareFloats or SoftwareFloats are requested explicitly, the following strategy is used: If the current value of DIGITS is smaller than 16 or if the matrix A is a hardware float array of domain type DOM_HFARRAY, then hardware arithmetic is tried. If this is successful, the result is returned.

If the result cannot be computed with hardware floats, software arithmetic by the MuPAD kernel is tried.

If the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY, or if one of the options Soft, SoftwareFloats or Symbolic is

specified, MuPAD computes the result with its software arithmetic without trying to use hardware floats first.

There may be several reasons for hardware arithmetic to fail:

- The current value of `DIGITS` is larger than 15.
- The data contains symbolic objects.
- The data contains numbers larger than 10^{308} or smaller than 10^{-308} that cannot be represented by hardware floats.

If neither `HardwareFloats` nor `SoftwareFloats` is specified, the user is not informed whether hardware floats or software floats are used.

If `HardwareFloats` are specified but fail due to one of the reasons above, a warning is issued that the (much slower) software floating-point arithmetic of the MuPAD kernel is used.

Note that `HardwareFloats` can only be used if all input data can be converted to floating-point numbers.

The trailing digits in floating-point results computed with `HardwareFloats` and `SoftwareFloats` may differ.

`ReturnType`

Option, specified as `ReturnType = t`

Return the eigenvector associated with the spectral radius as a vector of domain type `t`. The following return types are available: `DOM_ARRAY`, or `DOM_HFARRAY`, or `DOM_LIST`, or `Dom::Matrix()`, or `Dom::DenseMatrix()`.

`NoWarning`

Suppresses warnings

Return Values

A list `[lambda, x, residue]` is returned. The floating-point number `lambda` is an approximation of the eigenvalue of largest absolute value. The vector `x` is a numerical

eigenvector corresponding to `lambda`. `residue` is a floating-point number indicating the numerical quality of `lambda` and `x`.

If no return type is requested via the `ReturnType` option, the type of the returned vector `x` coincides with the type of the input vector x_0 (i.e., it is a 1-dimensional array of type `DOM_ARRAY` or `DOM_HFARRAY`, respectively, or a list, or a column vector of type `matrix` or `densematrix`. If no starting vector is specified, the type of `x` is determined by the type of `A`.

See Also

MuPAD Functions

`linalg::eigenvalues` | `linalg::eigenvectors` | `numeric::eigenvalues` | `numeric::eigenvectors` | `numeric::singularvalues` | `numeric::singularvectors` | `numeric::spectralRadius` | `numeric::svd`

numeric::spectralRadius

Spectral radius of a matrix

Syntax

```
numeric::spectralRadius(A, <x0>, <n>, <mode>, <ReturnType = t>, <NoWarning>)
```

Description

`numeric::spectralRadius(A)` returns data corresponding to the eigenvalue of the matrix `A` that has the largest absolute value.

`numeric::spectralRadius` and `numeric::spectralradius` are equivalent.

The spectral radius of a matrix with eigenvalues λ_i is $max(|\lambda_i|)$.

The return value `lambda` is an approximation of the corresponding eigenvalue: `abs(lambda)` is the spectral radius.

The return value `x` is the corresponding normalized eigenvector: $||x||_2 = 1$.

The return value $\text{residue} = ||A\,x - \text{lambda}\,x||_2$ provides an error estimate for the eigenvalue. For Hermitian matrices this is a rigorous upper bound for the error $|lambda - \lambda_{exact}|$, where λ_{exact} is the exact eigenvalue.

`numeric::spectralRadius` implements the power method to compute the eigenvalue and the associated eigenvector defining the spectral radius: the vector iteration $x_i = \frac{A^i x_0}{\|A^i x_0\|_2}$ "converges" towards the eigenspace associated with the spectral radius. The starting vector x_0 is provided by the second argument of `numeric::spectralradius`. If no starting vector is provided by the user, a randomly chosen vector is used.

Note: The iteration does not converge (converges slowly), if the spectral radius is generated by several distinct eigenvalues with the same (similar) absolute value.

Internally, the iteration stops, when the approximation of the eigenvalue becomes stationary within the relative precision given by `DIGITS`. If this does not happen within n iterations, then a warning is issued and the present values are returned. Cf. “Example 4” on page 19-365.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We let the routine choose a random starting vector:

```
A := matrix(2, 2, [[10, 1], [1, 20]]):
numeric::spectralradius(A)
```

$$\left[20.09901951,\ \begin{pmatrix} 0.0985376181 \\ 0.9951333267 \end{pmatrix},\ 0.000000001332123191\right]$$

We define a starting vector as a 1-dimensional array and allow a maximum of 1000 internal iterations:

```
A := array(1..2, 1..2, [[1, 2], [5, -10]]):
x0 := array(1..2, [1, 1]):
numeric::spectralradius(A, x0, 1000)
```

$$\left[-10.84428877,\ (\ 0.1665007247\ \ -0.9860413321\ ),\ 7.808576008\ 10^{-11}\right]$$

Next, we use a list to specify a starting vector:

```
A := array(1..2, 1..2, [[I, 3], [3, I]]):
numeric::spectralradius(A, [1, 1], 1000)
```

$$[3.0 + 1.0\,i,\ [0.7071067812,\ 0.7071067812],\ 0.0]$$

```
delete A, x0:
```

Example 2

With the default setting of `DIGITS` = 10, the following result is computed using `HardwareFloats`.

```
A := hfarray(1..2, 1..2, [[10^4, 10^4], [50,  60]]):
x0 := array(1..2, [1, 1]):
numeric::spectralradius(A, x0)
```

```
[10050.0498, ( 0.9999874753  0.00500491737 ), 0.000000004820940092]
```

We request `SoftwareFloats` in the next call. Note the difference in the trailing digits:

```
numeric::spectralradius(A, x0, Soft)
```

```
[10050.0498, ( 0.9999874753  0.00500491737 ), 0.000000004820942201]
```

```
delete DIGITS, A, x0:
```

Example 3

The eigenvector that is returned can have various types. If no starting vector is provided, the type of the matrix determines the type of the eigenvector:

```
A:= array(1..2, 1..2, [[1, 2], [3, 4]]):
[l, x, residue]:= numeric::spectralradius(A);
```

```
[5.372281324, ( 0.4159735579  0.9093767091 ), 0.0000000002881934691]
```

```
domtype(x)
```

```
DOM_ARRAY
```

```
A:= hfarray(1..2, 1..2, [[1, 2], [3, 4]]):
[l, x, residue]:= numeric::spectralradius(A):
domtype(x)
```

```
DOM_HFARRAY
```

```
A:= matrix(2, 2, [[1, 2], [3, 4]]):
[l, x, residue]:= numeric::spectralradius(A):
domtype(x)
```

```
Dom::Matrix()
```

If a starting vector is provided, its type determines the type of the return vector:

```
A:= hfarray(1..2, 1..2, [[1, 2], [3, 4]]):
x0:= [1, 1]:
[l, x, residue]:= numeric::spectralradius(A, x0):
domtype(x)
```

```
DOM_LIST
```

```
x0:= array(1..2, [1, 1]):
[l, x, residue]:= numeric::spectralradius(A, x0):
domtype(x)
```

```
DOM_ARRAY
```

```
x0:= hfarray(1..2, [1, 1]):
[l, x, residue]:= numeric::spectralradius(A, x0):
domtype(x)
```

```
DOM_HFARRAY
```

```
x0:= matrix([1, 1]):
[l, x, residue]:= numeric::spectralradius(A, x0):
domtype(x)
```

```
Dom::Matrix()
```

The return type can be requested explicitly:

```
[l, x, residue] :=
  numeric::spectralradius(A, x0, ReturnType = DOM_LIST):
domtype(x)
```

```
DOM_LIST
```

```
[l, x, residue] :=
  numeric::spectralradius(A, x0, ReturnType = DOM_HFARRAY):
domtype(x)
```

DOM_HFARRAY

```
delete A, x0, l, x, residue:
```

Example 4

The following matrix has two distinct eigenvalues 1 and -1 of the same absolute value. The power method must fail.

```
A := array(1..2, 1..2, [[1, 0], [0, -1]]):
```

We allow a maximum of 1000 internal steps. The call results in a warning. The large residue also indicates that the power method did not converge:

```
numeric::spectralradius(A, [1, 1], 1000)
```

```
Warning: There is no convergence of vector iteration. [numeric::spectralradius]
```

$$[1.0, [0.7071067812, -0.7071067812], 1.414213562]$$

```
delete A:
```

Parameters

A

An $m \times m$ array of domain type `DOM_ARRAY` or `DOM_HFARRAY` or a matrix of category `Cat::Matrix`

x_0

A starting vector: a 1-dimensional array, or an hfarray, or a list of length m. Also 2-dimensional arrays (`array(1..m, 1..1, ...)`, `hfarray(1..m, 1..1, ...)`) and matrices representing vectors are accepted.

n

The maximal number of iterations: a positive integer. The default value is 1000.

mode

One of the flags Hard, HardwareFloats, Soft, or SoftwareFloats

Options

Hard, HardwareFloats, Soft, SoftwareFloats

With Hard (or HardwareFloats), computations are done using fast hardware float arithmetic from within a MuPAD session. Hard and HardwareFloats are equivalent. With this option, the input data are converted to hardware floats and processed by compiled C code. The result is reconverted to MuPAD floats and returned to the MuPAD session.

With Soft (or SoftwareFloats) computations are dome using software float arithmetic provided by the MuPAD kernel. Soft and SoftwareFloats are equivalent. SoftwareFloats is used by default if the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY.

Compared to the SoftwareFloats used by the MuPAD kernel, the computation with HardwareFloats may be many times faster. Note, however, that the precision of hardware arithmetic is limited to about 15 digits. Further, the size of floating-point numbers may not be larger than approximately 10^{308} and not smaller than approximately 10^{-308}.

If no HardwareFloats or SoftwareFloats are requested explicitly, the following strategy is used: If the current value of DIGITS is smaller than 16 or if the matrix A is a hardware float array of domain type DOM_HFARRAY, then hardware arithmetic is tried. If this is successful, the result is returned.

If the result cannot be computed with hardware floats, software arithmetic by the MuPAD kernel is tried.

If the current value of DIGITS is larger than 15 and the input matrix A is not of domain type DOM_HFARRAY, or if one of the options Soft, SoftwareFloats or Symbolic is

specified, MuPAD computes the result with its software arithmetic without trying to use hardware floats first.

There may be several reasons for hardware arithmetic to fail:

- The current value of `DIGITS` is larger than 15.
- The data contains symbolic objects.
- The data contains numbers larger than 10^{308} or smaller than 10^{-308} that cannot be represented by hardware floats.

If neither `HardwareFloats` nor `SoftwareFloats` is specified, the user is not informed whether hardware floats or software floats are used.

If `HardwareFloats` are specified but fail due to one of the reasons above, a warning is issued that the (much slower) software floating-point arithmetic of the MuPAD kernel is used.

Note that `HardwareFloats` can only be used if all input data can be converted to floating-point numbers.

The trailing digits in floating-point results computed with `HardwareFloats` and `SoftwareFloats` may differ.

`ReturnType`

Option, specified as `ReturnType = t`

Return the eigenvector associated with the spectral radius as a vector of domain type `t`. The following return types are available: `DOM_ARRAY`, or `DOM_HFARRAY`, or `DOM_LIST`, or `Dom::Matrix()`, or `Dom::DenseMatrix()`.

`NoWarning`

Suppresses warnings

Return Values

A list `[lambda, x, residue]` is returned. The floating-point number `lambda` is an approximation of the eigenvalue of largest absolute value. The vector `x` is a numerical

eigenvector corresponding to `lambda`. `residue` is a floating-point number indicating the numerical quality of `lambda` and `x`.

If no return type is requested via the `ReturnType` option, the type of the returned vector `x` coincides with the type of the input vector x_0 (i.e., it is a 1-dimensional array of type `DOM_ARRAY` or `DOM_HFARRAY`, respectively, or a list, or a column vector of type `matrix` or `densematrix`. If no starting vector is specified, the type of `x` is determined by the type of `A`.

See Also

MuPAD Functions

`linalg::eigenvalues` | `linalg::eigenvectors` | `numeric::eigenvalues` | `numeric::eigenvectors` | `numeric::singularvalues` | `numeric::singularvectors` | `numeric::spectralradius` | `numeric::svd`

numeric::sum

Numerical approximation of sums (the Float attribute of Sum)

Syntax

```
numeric::sum(f(x), x = a .. b)
numeric::sum(f(x), x in {x1, x2, …})
numeric::sum(f(x), x = {x1, x2, …})
numeric::sum(f(x), x in RootOf(p(X), X))
numeric::sum(f(x), x = RootOf(p(X), X))
float(hold(sum)(f(x), x = a .. b))
float(hold(sum)(f(x), x in {x1, x2, …}))
float(hold(sum)(f(x), x = {x1, x2, …}))
float(hold(sum)(f(x), x in RootOf(p(X), X)))
float(hold(sum)(f(x), x = RootOf(p(X), X)))
float(freeze(sum)(f(x), x = a .. b))
float(freeze(sum)(f(x), x in {x1, x2, …}))
float(freeze(sum)(f(x), x = {x1, x2, …}))
float(freeze(sum)(f(x), x in RootOf(p(X), X)))
float(freeze(sum)(f(x), x = RootOf(p(X), X)))
```

Description

`numeric::sum(f(i), i=a..b)` computes a numerical approximation of $\sum_{i=a}^{b} f(i)$.

`numeric::sum(f(x), x ∈ {x1, x2, …})` computes a numerical approximation of $\sum_{x \in \{x_1, x_2, \ldots\}} f(x)$.

`numeric::sum(f(x), x in RootOf(p(X), X))` computes a numerical approximation of $\sum_{x \in \mathrm{RootOf}(p(X), X)} f(x)$.

The call `numeric::sum(…)` is equivalent to calling the `float` attribute of `sum` via `float ( hold( sum )(…))` or `float ( freeze( sum )(…))`.

If there are other symbolic parameters in $f(x)$, apart from the summation variable x, a symbolic sum is returned. Numerical expressions such as e^{π}, $\sqrt{2}$ etc. are accepted and converted to floating-point numbers.

Note: For infinite sums, the expression $f(i)$ with integer i must have an extension $f(x)$ to all real x in the interval $[a, b]$. Internally, the integral $\int_a^b f(x)\,dx$ is computed numerically and used in the approximation process.

Note: For finite sums, `numeric::sum` just returns `_plus ( float(f(i)$i=a..b))`. Note that numerical cancellation may occur! If `f(i)` does not contain floating-point numbers, cancellation can be avoided summing the symbolic terms by `_plus(f(i)$i=a..b)` instead. Cf. “Example 3” on page 19-372.

Convergence is fast, if $f(x)$ decays rapidly for `x -> infinity` or `| x | -> infinity`, respectively

---

Note: Convergence may be slow for alternating sums containing expressions such as $(-1)^i$. Such sums are also often subject to cancellation problems!

The call `numeric::sum(f(x), x = {x1, x2, …})` computes numerical approximations of `x1, x2` etc., substitutes these values into $f(x)$ and adds up the results. This process may be subject to cancellation problems!

The calls `numeric::sum(f(x), x ∈ {x1, x2, …})` and `numeric::sum(f(x), x = {x1, x2, …})` are equivalent.

The call `numeric::sum(f(x), x in RootOf(p(X), X))` computes numerical approximations of all roots of p, substitutes these values into $f(x)$ and adds up the results. Cf. “Example 4” on page 19-372. This process may be subject to cancellation problems!

The calls `numeric::sum(f(x), x in RootOf(p(X), X))` and `numeric::sum(f(x), x = RootOf(p(X), X))` are equivalent.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We demonstrate some equivalent calls for numerical summation:

```
numeric::sum(1/i!, i = 0..infinity),
float(hold(sum)(1/i!, i = 0..infinity)),
float(freeze(sum)(1/i!, i = 0..infinity))
```

$$2.718281828,\ 2.718281828,\ 2.718281828$$

The MuPAD symbolic summation `sum` does not find a simple representation of the following sum:

```
sum(1/i!/(i^2+1)!, i = 0..infinity)
```

$$\sum_{i=0}^{\infty} \frac{1}{(i^2+1)!\,i!}$$

The following float evaluation calls `numeric::sum`:

```
float(%)
```

$$1.504166713$$

The exact value of the following sum is $\pi\,coth(\pi)$:

```
numeric::sum(1/(1+i^2), i = -infinity..infinity) =
```

```
float(PI*coth(PI))
```

$$3.153348095 = 3.153348095$$

Example 2

The following sum cannot be evaluated numerically because of the symbolic parameter `x`:

```
numeric::sum(1/(x+i^2), i = -infinity..infinity)
```

$$\text{numeric::sum}\left(\frac{1}{i^2+x}, i = -\infty..\infty\right)$$

Example 3

We demonstrate numerical cancellation when summing the Taylor series for e^{-20}:

```
exp(-20.0) <> numeric::sum((-20)^i/i!, i = 0..100)
```

$$0.000000002061153622 \neq 0.000000002148938644$$

Also the infinite sum suffers from cancellation:

```
exp(-20.0) <> numeric::sum((-20)^i/i!, i = 0..infinity)
```

$$0.000000002061153622 \neq 0.000000002205327316$$

Cancellation can be avoided using a finite sum with exact terms:

```
exp(-20.0) = float(_plus((-20)^i/i! $ i = 0..100))
```

$$0.000000002061153622 = 0.000000002061153622$$

Example 4

The following call computes the numerical roots of the polynomial in the `RootOf` expression and sums over all the roots:

```
numeric::sum(exp(x)/x, x in RootOf(X^10 - X - PI, X))
```

```
9.681693381
```

Parameters

`f(x)`

An arithmetical expression in x

`i, x`

Summation variables: identifiers or indexed identifiers

`a, b`

Integers or ±`infinity` satisfying $a \le b$

x_1, x_2, …

Numerical expressions

`p(X)`

A univariate polynomial expression in X

`X`

The indeterminate of p: an identifier or an indexed identifier

Return Values

Floating point number or a symbolic expression of type `numeric::sum`.

Algorithms

Depending on whether the series is alternating or monotone, `numeric::sum` tries a number of strategies to calculate its limit: Levin's u transformation, the Euler-MacLaurin formula or van Wijngaarden's trick.

The Euler-MacLaurin formula is

$$\sum_{i=a}^{b} f(i) = \frac{f(a)+f(b)}{2} + \int_a^b f(x)\,\mathrm{d}x + \left(\sum_{m=1}^{M} \frac{B_{2\,m}}{(2\,m)!}\left(f(b)^{(2\,m-1)} - f(a)^{(2\,m-1)}\right)\right) + \cdots$$

involving the Bernoulli numbers $B_{2\,m}$.

See Also

MuPAD Functions

`_plus` | `int` | `numeric::product` | `numeric::quadrature` | `sum`

numlib – Number Theory

numlib::checkPrimalityCertificate
numlib::contfrac
numlib::contfracPeriodic
numlib::cornacchia
numlib::decimal
numlib::divisors
numlib::factorGaussInt
numlib::fibonacci
numlib::fromAscii
numlib::g_adic
numlib::ichrem
numlib::igcdmult
numlib::invphi
numlib::ispower
numlib::isquadres
numlib::issqr
numlib::jacobi
numlib::Lambda
numlib::lambda
numlib::legendre
numlib::lincongruence
numlib::mersenne
numlib::moebius
numlib::mroots
numlib::msqrts
numlib::numdivisors
numlib::numprimedivisors
numlib::omega
numlib::Omega
numlib::order
numlib::phi
numlib::pi

numlib::proveprime
numlib::primedivisors
numlib::primroot
numlib::reconstructRational
numlib::sigma
numlib::sqrt2cfrac
numlib::sqrtmodp
numlib::sumdivisors
numlib::sumOfDigits
numlib::tau
numlib::toAscii

numlib::checkPrimalityCertificate

Test the primality certificate

Syntax

```
numlib::checkPrimalityCertificate(certificate)
```

Description

`numlib::checkPrimalityCertificate` tests the certificate of primality returned by `numlib::proveprime`. For large prime numbers, the `numlib::proveprime` function generates certificates that provide all data you need for proving primality of a number by the Atkin-Goldwasser-Kilian-Morain algorithm. See “Example 1” on page 20-3.

For small prime numbers, `numlib::proveprime` does not return a certificate of primality. Instead, it returns `TRUE`. For nonprime numbers `numlib::proveprime` returns `FALSE`. In both cases, you do not need to use `numlib::checkPrimalityCertificate`.

Examples

Example 1

Use the `numlib::proveprime` function to check the primality of the number 1299709. The function returns the following sequence of lists. This sequence is the certificate of primality:

```
certificate := numlib::proveprime(1299709)
```

```
[1299709, 15, [2, 2, 2, 2, 2, 107, 379], 700619, 67686, 0, 796444, [107, 379]]
```

The certificate provides all data that you need for proving primality of 1299709 by the Atkin-Goldwasser-Kilian-Morain algorithm. You can substitute the numbers into the algorithm and verify the primality of the number. Alternatively, you can verify the certificate by using the `numlib::checkPrimalityCertificate` function:

```
numlib::checkPrimalityCertificate(certificate)
```

```
    TRUE
```

Parameters

`certificate`

A list or a sequence of lists returned by `numlib::proveprime`

Return Values

TRUE or FALSE

See Also

MuPAD Functions

`ifactor` | `numlib::proveprime`

More About

- "Primes and Factorizations"

numlib::contfrac

Domain of continued fractions

Syntax

```
numlib::contfrac(x, <n>)
```

Description

`numlib::contfrac(x)` creates a continued fraction approximation for the real number `x`.

If `x` is an integer or a rational number and `n` is not specified, a continued fraction is returned that represents `x` exactly. Cf. “Example 1” on page 20-6.

Irrational numerical values `x` such as `1 + sqrt(2)` or `PI/3` are first converted to floating-point numbers. The first `n` significant decimals of floating-point numbers are taken into account. If `n` is not specified, `n = DIGITS` is used. The value of the continued fraction (given by `numlib::contfrac ::rational`) satisfies

$$\left|x - \text{numlib::contfrac::rational}(x)\right| \le |x| \ \frac{1}{10^n}.$$

Integers or rational numbers are also converted to floating point numbers, if a precision `n` is specified.

Objects of type `numlib::contfrac` can be handled by the usual arithmetical operations. They are sensitive to the environment variable `DIGITS` if floating-point numbers or irrational numerical values are involved.

Use `contfrac` to compute continued fraction approximations of expressions involving symbolic parameters.

Examples

Example 1

For rational numbers, exact representations are returned:

```
numlib::contfrac(123/1234)
```

$$\frac{1}{10+\frac{1}{30+\frac{1}{1+\frac{1}{3+\ldots}}}}$$

The rational representation (the second operand of the continued fraction) coincides with the original rational:

```
numlib::contfrac::rational(%), expr(%), op(%, 2)
```

$$\frac{123}{1234}, \frac{123}{1234}, \frac{123}{1234}$$

Restricted continued fraction approximations can be computed by passing a precision as second argument:

```
numlib::contfrac(123/1234, 2),
numlib::contfrac(123/1234, 3),
numlib::contfrac(123/1234, 5)
```

$$\frac{1}{10+\ldots}, \frac{1}{10+\frac{1}{30+\ldots}}, \frac{1}{10+\frac{1}{30+\frac{1}{1+\frac{1}{3+\ldots}}}}$$

$$\frac{1}{10+\ldots}, \frac{1}{10+\frac{1}{30+\ldots}}, \frac{1}{10+\frac{1}{30+\frac{1}{1+\frac{1}{3+\ldots}}}}$$

Example 2

The coefficients are extracted by the method `nthcoeff`:

```
cf := numlib::contfrac(12/123)
```

$$\frac{1}{10+\frac{1}{4+\ldots}}$$

```
nthcoeff(cf, 1), nthcoeff(cf, 2), nthcoeff(cf, 3), nthcoeff(cf, 4)
```

$$0,\ 10,\ 4,\ \text{FAIL}$$

The internal list of coefficients can also be queried via `op`:

```
op(cf, 1)
```

$$[0,\ 10,\ 4]$$

```
delete cf:
```

Example 3

`numlib::contfrac` can also compute continued fraction approximations of irrational numbers:

```
numlib::contfrac(PI, 2),
numlib::contfrac(PI, 4),
numlib::contfrac(PI, 5)
```

$$3+\frac{1}{7+\ldots},\ 3+\frac{1}{7+\frac{1}{15+\ldots}},\ 3+\frac{1}{7+\frac{1}{15+\frac{1}{1+\ldots}}}$$

$$3+\frac{1}{7+\ldots},\ 3+\frac{1}{7+\frac{1}{15+\ldots}},\ 3+\frac{1}{7+\frac{1}{15+\frac{1}{1+\ldots}}}$$

A finite continued fraction approximation may be regarded as an interval of numbers (the symbol … represents a number between 0 and 1):

```
numlib::contfrac::rationalInterval(numlib::contfrac(PI, 2));
```

$$\left[\frac{25}{8}, \frac{22}{7}\right]$$

```
float(%)
```

$$[3.125, 3.142857143]$$

Example 4

All basic arithmetical operations are available for continued fractions:

```
x := numlib::contfrac(PI, 3):
y := numlib::contfrac(1/12):
DIGITS:= 3: 3/x + sqrt(2)*y^(1/3)
```

$$1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{26 + \ldots}}}}}$$

$$1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{26 + \ldots}}}}}$$

```
delete x, y, DIGITS:
```

Example 5

We search for a simple continued fraction in an interval:

```
numlib::contfrac::convert(1/2 - 1/10^8, 1/2 + 1/10^8)
```

$$\cfrac{1}{2 + \ldots}$$

$$\frac{1}{2+\ldots}$$

```
numlib::contfrac::convert(PI, PI + 1/10^10)
```

$$3+\frac{1}{7+\frac{1}{15+\frac{1}{1+\frac{1}{292+\frac{1}{1+\frac{1}{1+\frac{1}{1+\ldots}}}}}}}$$

$$3+\frac{1}{7+\frac{1}{15+\frac{1}{1+\frac{1}{292+\frac{1}{1+\frac{1}{1+\frac{1}{1+\ldots}}}}}}}$$

Parameters

x

A real numerical expression

n

The number of significant digits: a positive integer greater one

Methods

Mathematical Methods

`_plus` — Sum of two continued fractions

```
_plus(x, y)
```

`_mult` — Product of two continued fractions

```
_mult(x, y)
```

`_invert` — Reciprocal of a continued fraction

```
_invert(x)
```

Inverting a continued fraction means a shift of the coefficients by one to the left or to the right.

If `x` is an exact representation of a rational number, the continued fraction expansion of the reciprocal is also an exact representation.

`_power` — Power of a continued fraction

```
_power(x, m)
```

Access Methods

`print` — Print a continued fraction

```
print(cf)
```

See Also

MuPAD Functions

`contfrac` | `numeric::rationalize`

numlib::contfracPeriodic

Periodic continued fraction expansions

Syntax

```
numlib::contfracPeriodic(p, q, n)
```

Description

`numlib::contfracPeriodic(p, q, n)` returns the continued fraction expansion of `p + q*sqrt(n)` as a sequence of two lists: the first one contains the non-periodic part, the second one contains the periodic part of the expansion.

The non-periodic part may be an empty list. No periodic part is returned for rational input, i.e., $q = 0$ or `n` square.

Examples

Example 1

The non-periodic part may start with zero. All other coefficients of a continued fraction expansion are positive:

```
numlib::contfracPeriodic(2/7, 1/7, 2)
```

```
[0, 2, 19], [1, 8, 1, 18]
```

The result agrees with that one of `contfrac`:

```
op(contfrac(2/7 + 1/7 *sqrt(2)), 1)
```

```
[0, 2, 19, 1, 8, 1, 18, 1, 8, 1]
```

Example 2

The golden mean is famous for its simple continued fraction expansion:

```
numlib::contfracPeriodic(1/2, 1/2, 5)
```

$$[], [1]$$

Example 3

Since 81 is a perfect square, there is no periodic part in the continued fraction expansion of its square root:

```
numlib::contfracPeriodic(0, 1, 81)
```

$$[9]$$

Parameters

p

A rational number

q

A rational number

n

A positive integer

Return Values

If $p + q\sqrt{n}$ is a rational number, then `numlib::contfracPeriodic` returns one list, otherwise two lists of integers.

Algorithms

A real number has a periodic continued fraction expansion if and only if it is of the form $p + q\sqrt{n}$.

See Also

MuPAD Functions

`numlib::contfrac` | `numlib::sqrt2cfrac`

numlib::cornacchia

Cornachia's algorithm

Syntax

```
numlib::cornacchia(a, b, m)
```

Description

`numlib::cornacchia(a, b, m)` returns all pairs of positive and relatively prime integers x, y that solve the equation $ax^2 + by^2 = m$.

The arguments `a`, `b`, `m` must be pairwise relatively prime.

Examples

Example 1

We compute the solutions to $3\,x^2 + 5\,y^2 = 74533332452454382449233$:

```
numlib::cornacchia(3, 5, 74533332452454382449233)
```

```
{[22457088474, 120847316879]}
```

Example 2

For non-prime `m`, there may be many solutions:

```
numlib::cornacchia(1, 4, 5*13*17*29*73)
```

```
{[103, 763], [151, 761], [217, 757], [521, 719], [553, 713], [583, 707], [809, 649], [887, 623],
  [1081, 541], [1127, 517], [1159, 499], [1367, 343], [1369, 341], [1463, 223], [1481, 191],
  [1529, 19]}
```

Parameters

a

A positive integer

b

A positive integer

m

A positive integer

Return Values

`numlib::cornacchia` returns a set each element of which is a list of two positive integers.

See Also

MuPAD Functions
`numlib::msqrts`

numlib::decimal

Infinite representation of rational numbers

Syntax

```
numlib::decimal(q)
```

Description

`numlib::decimal(q)` computes the decimal expansion of a rational number `q`.

If `q` is a nonnegative rational number whose decimal expansion is finite, then `numlib::decimal(q)` returns the expression sequence starting with the integral part of `q` and followed by the digits after the decimal point.

If `q` is a nonnegative rational number whose decimal expansion is infinite, then `numlib::decimal(q)` returns the expression sequence starting with the integral part of `q`, followed by the digits of the pre-period and terminated with a list, containing the digits of a minimal period.

Examples

Example 1

Computing the decimal expansion of 1999:

```
numlib::decimal(1999)
```

```
1999
```

Example 2

Computing the (finite) decimal expansion of $\frac{52187}{78125}$:

```
numlib::decimal(52187/78125)
```

$$0, 6, 6, 7, 9, 9, 3, 6$$

Example 3

Computing the (infinite) decimal expansion of $\frac{111}{7}$:

```
numlib::decimal(111/7)
```

$$15, [8, 5, 7, 1, 4, 2]$$

Example 4

Computing the (infinite) decimal expansion of $\frac{37}{28}$:

```
numlib::decimal(37/28)
```

$$1, 3, 2, [1, 4, 2, 8, 5, 7]$$

Parameters

q

Nonnegative rational number

Return Values

`numlib::decimal(q)` returns an expression sequence consisting of nonnegative integers or an expression sequence consisting of nonnegative integers and terminated by a list of nonnegative integers.

numlib::divisors

Divisors of an integer

Syntax

```
numlib::divisors(n)
```

Description

`numlib::divisors(n)` returns the list of positive divisors of `n`.

If `a` is a non-zero integer then `numlib::divisors(a)` returns the sorted list of all positive divisors of `a`.

`numlib::divisors(0)` returns `[0]`.

`numlib::divisors` returns an error if the argument evaluates to a number of wrong type.

Examples

Example 1

We compute the list of all positive divisors of 72:

```
numlib::divisors(72)
```

```
[1, 2, 3, 4, 6, 8, 9, 12, 18, 24, 36, 72]
```

Example 2

`numlib::divisors` returns the positive divisors of negative numbers, too:

```
numlib::divisors(-63)
```

```
[1, 3, 7, 9, 21, 63]
```

Parameters

n

Integer

Return Values

`numlib::divisors` returns a list of nonnegative integers.

Algorithms

Internally, `ifactor` is used for factoring `n`.

See Also

MuPAD Functions

`ifactor` | `numlib::numdivisors` | `numlib::numprimedivisors` | `numlib::primedivisors` | `numlib::tau` | `polylib::divisors`

numlib::factorGaussInt

Factorization of Gaussian integers

Syntax

```
numlib::factorGaussInt(n)
```

Description

`numlib::factorGaussInt(n)` returns the factorization of the Gaussian integer `n` into Gaussian primes. Among associate primes, that one with smallest polar angle is chosen.

Examples

Example 1

In the Gaussian integers, 3 remains prime while 5 does not:

```
numlib::factorGaussInt(3), numlib::factorGaussInt(5)
```

$$[1, 3, 1], [-i, 1+2\,i, 1, 2+i, 1]$$

Example 2

The argument to `numlib::factorGaussInt` may be any Gaussian integer, that is, every complex number of the form $a + b\,\mathrm{i}$ where a and b are integers:

```
numlib::factorGaussInt(2+2*I)
```

$$[-i, 1+i, 3]$$

Parameters

n

An integer, or a complex number whose real and imaginary part are integers

Return Values

`numlib::factorGaussInt` returns a list `[u, p1, a1, ..., pk, ak]` where u is a unit in the Gaussian integers, the p_i are Gaussian primes and the a_i are positive integers, such that $n = u\left(\prod_{i=1}^{k} p_i^{a_i}\right)$.

Algorithms

The function `ifactor` is used to factor the norm; this step takes most of the running time. Hence, the running time of the algorithm mainly depends on the size of the prime factors of the norm of `n`.

See Also

MuPAD Functions
`factor`

numlib::fibonacci

Fibonacci numbers

Syntax

```
numlib::fibonacci(n)
```

Description

`numlib::fibonacci(n)` returns the `n`-th Fibonacci number.

If `n` is a nonnegative integer then `numlib::fibonacci(n)` returns the `n`-th Fibonacci number.

`numlib::fibonacci` returns an error if the argument evaluates to a number of wrong type. `numlib::fibonacci` returns the unevaluated function call if `n` does not evaluate to a number.

Examples

Example 1

We compute the 201st Fibonnacci number:

```
numlib::fibonacci(201)
```

```
453973694165307953197296969697410619233826
```

Parameters

n

A nonnegative integer

Return Values

Nonnegative integer, or the function call with its arguments evaluated.

Algorithms

The n-th Fibonacci number F_n is defined by the recursion formula $F_0 = 0$, $F_1 = 1$, and $F_{n+2} = F_n + F_{n+1}$.

`numlib::fibonacci` uses quadratic recursion formulas.

numlib::fromAscii

Decoding of ASCII codes

Syntax

```
numlib::fromAscii(listOfCodes)
```

Description

If `L` is a list of ASCII codes then `numlib::fromAscii(L)` returns the string coded by `L`.

ASCII codes of non-printable characters, i. e., codes between `0` and `8` and between `11` and `31`, are ignored.

`numlib::fromAscii` returns an error if its argument is not a list of integers between `0` and `127`, i. e., not a list of legal ASCII codes.

Examples

Example 1

Non-printable characters are ignored, but tabulator and newline characters are decoded.

```
L := [0,1,2,3,9,10,31,10,9,32,45,32,101,105,110,32,
            84,101,115,116,32,61,32,97,32,116,101,115,116]:

numlib::fromAscii(L)
```

```
"            - ein Test = a test"
```

Parameters

`listOfCodes`

A list of ASCII codes

Return Values

String

See Also

MuPAD Functions
`numlib::toAscii`

numlib::g_adic

G-adic representation of a nonnegative integer

Syntax

```
numlib::g_adic(number, base)
```

Description

`numlib::g_adic(0, g)` returns `[0]`.

`numlib::g_adic` returns an error if the arguments evaluate to numbers which are not both of the correct type.

If `a` is a natural number and `g` is an integer such that $|g| > 1$, `numlib::g_adic(a, g)` returns the `g`-adic representation of `a` as a list $[a_0, \ldots, a_r]$ such that

$$a = a_0 + a_1\, g + a_2\, g^2 + \ldots + a_r\, g^r$$

and $0 \le a_i < |g|$ für $i = 0, \ldots, r - 1$ and $0 < a_r < |g|$.

Examples

Example 1

Computing the dyadic representation of 1994:

```
numlib::g_adic(1994, 2)
```

```
[0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1]
```

Example 2

Computing the hexadecimal representation of 2001:

```
numlib::g_adic(2001, 16)
```

```
[1, 13, 7]
```

Parameters

`number`

An nonnegative integer

`base`

An integer with absolute value is greater than 1

Return Values

List of nonnegative integers, or the function call with evaluated arguments if one of the arguments is not a number.

See Also

MuPAD Functions

`genpoly` | `int2text` | `text2int`

numlib::ichrem

Chinese remainder theorem for integers

Syntax

```
numlib::ichrem(a, m)
```

Description

`numlib::ichrem(a,m)` returns the least nonnegative integer x such that $x \equiv a_i \pmod{m_i}$ for $i = 1, \ldots, nops(m)$ if such a number exists; otherwise `numlib::ichrem(a,m)` returns `FAIL`.

The entries in `m` need not be pairwise coprime.

`numlib::ichrem(a,m)` returns an error if `a` is not a list of integers or `m` is not a list of natural numbers or `a` and `m` are not lists of the same length.

Examples

Example 1

Here the moduli are pairwise coprime. In this case, a solution always exists:

```
numlib::ichrem([2,3,2],[3,5,7])
```

```
23
```

Example 2

Here the moduli are not pairwise coprime, and a solution does not exist:

```
numlib::ichrem([5,6,8],[20,21,22])
```

```
FAIL
```

Example 3

Also here the moduli are not pairwise coprime, but a solution nevertheless exists:

```
numlib::ichrem([5,6,7],[20,21,22])
```

```
4605
```

Parameters

a

A list of integers

m

A list of natural numbers of the same length as `a`

Return Values

Either a nonnegative integer or `FAIL`.

See Also

MuPAD Functions

`numlib::lincongruence`

numlib::igcdmult

Extended Euclidean algorithm for integers

Syntax

```
numlib::igcdmult(par1, par2, …)
```

Description

`numlib::igcdmult` is an extension of the kernel function `igcdex`.

`numlib::igcdmult` returns an error if the arguments evaluate to numbers which are not all of the correct type.

For integers `a_1,a_2,...,a_n`, `numlib::igcdmult(a_1,a_2,...,a_n)` returns a list $[d, v_1, \ldots, v_n]$ of integers such that `d` is the nonnegative greatest common divisor of `a_1,a_2,...,a_n` and `d = a_1*v_1 + a_2*v_2 + ...+ a_n*v_n`.

For integers `a_1,a_2,...,a_n`, `numlib::igcdmult(a_1,a_2,...,a_n)` returns a list $[d, v_1, \ldots, v_n]$ of integers such that `d` is the nonnegative greatest common divisor of `a_1,a_2,...,a_n` and `d = a_1*v_1 + a_2*v_2 + ...+ a_n*v_n`.

Examples

Example 1

Computing the greatest non-negative common divisor d of 455, 385, 165, 273 and integers v_1, v_2, v_3, v_4 such that $d = 455\,v_1 + 385\,v_2 + 165\,v_3 + 273\,v_4$:

```
numlib::igcdmult(455,385,165,273)
```

$$[1, -7630, 9156, -327, 2]$$

Parameters

`par1`

Integer

`par2, …`

Integers

Return Values

List of integers, or the function call with evaluated arguments if some argument is not a number.

See Also

MuPAD Functions

`igcd` | `igcdex`

numlib::invphi

Inverse of the Euler phi function

Syntax

```
numlib::invphi(n)
```

Description

`numlib::invphi(n)` computes all positive integers `i` with $\varphi(i) = n$.

Examples

Example 1

We compute all numbers `i` with $\varphi(i) = 500$:

```
s := numlib::invphi(500)
```

```
[625, 753, 1004, 1250, 1506]
```

Test for correctness:

```
map(s, numlib::phi)
```

```
[500, 500, 500, 500, 500]
```

Parameters

n

A positive integer

Return Values

List of positive integer numbers.

See Also

MuPAD Functions

`numlib::phi`

numlib::ispower

Test for perfect powers

Syntax

```
numlib::ispower(n)
```

Description

`numlib::ispower(n)` tests whether `n` is of the form a^k for some integers a, k with a, $k \geq 2$.

`numlib::ispower` returns `FALSE` if `n` is not a perfect power.

Among several pairs (a, k) for which $n = a^k$, that one with minimal a is returned.

Examples

Example 1

This number is a perfect power:

```
numlib::ispower(1977326743)
```

```
7, 11
```

This number is not a perfect power:

```
numlib::ispower(1977326744)
```

```
FALSE
```

Parameters

n

An integer

Return Values

`numlib::ispower` returns a sequence of two positive integers greater than 1, or `FALSE` if `n` is not a perfect power.

See Also

MuPAD Functions
`_power` | `ifactor` | `isqrt`

numlib::isquadres

Test for quadratic residues

Syntax

```
numlib::isquadres(a, m)
```

Description

If the integer number `a` is a quadratic residue modulo the natural number `m`numlib::isquadres(a,m) returns `TRUE`, and if `a` is a quadratic non-residue modulo `m`numlib::isquadres(a,m) returns `FALSE`.

If `a` and `m` are not coprime `numlib::isquadres(a,m)` returns an error.

`numlib::isquadres` returns an error if the arguments evaluate to numbers which are not both of the correct type.

`numlib::isquadres` returns the function call with its arguments evaluated if the arguments do not evaluate to numbers.

Examples

Example 1

132132 is a quadratic residue modulo 3231227:

```
numlib::isquadres(132132, 3231227)
```

```
    TRUE
```

Example 2

222222 is a quadratic non-residue modulo 324899:

```
numlib::isquadres(222222,324899)
```

```
    FALSE
```

Example 3

37 is a quadratic residue modulo 48884:

```
numlib::isquadres(37,48884)
```

```
    TRUE
```

Parameters

a

An integer

m

A natural number coprime to a

Return Values

`numlib::isquadres` returns `TRUE`, `FALSE`, or the function call with its arguments evaluated.

See Also

MuPAD Functions

`numlib::jacobi` | `numlib::legendre` | `numlib::msqrts`

numlib::issqr

Test for perfect squares

Syntax

```
numlib::issqr(a)
```

Description

`numlib::issqr(a)` returns `TRUE` if `a` is the square of an integer, and `FALSE` otherwise.

Examples

Example 1

361 is the square of 19:

```
numlib::issqr(361)
```

```
    TRUE
```

Example 2

362 is not a square:

```
numlib::issqr(362)
```

```
    FALSE
```

Example 3

Negative integers are not squares:

```
numlib::issqr(-361)
```

```
FALSE
```

Parameters

a

An integer

Return Values

`numlib::issqr` returns `TRUE`, `FALSE`, or the unevaluated call.

See Also

MuPAD Functions

`isqrt` | `numlib::ispower` | `sqrt`

numlib::jacobi

Jacobi symbol

Syntax

```
numlib::jacobi(a, m)
```

Description

`numlib::jacobi(a,m)` returns the Jacobi symbol `(a | m)`.

`numlib::jacobi` returns an error if one of its arguments evaluates to a number of wrong type.

Examples

Example 1

Computing the Jacobi symbol `(222222 | 304679)`:

```
numlib::jacobi(222222, 304679)
```

```
-1
```

Example 2

Computing the Jacobi-Symbol `(222222 | 324889)`:

```
numlib::jacobi(222222, 324899)
```

```
1
```

Example 3

Computing the Jacobi symbol `(222222 | 333333)`:

```
numlib::jacobi(222222, 333333)
```

```
    0
```

Parameters

a

An integer

m

An odd positive integer

Return Values

`numlib::jacobi(a,m)` returns 0, 1, or -1, or the function call with evaluated arguments if one of the arguments is not a number.

Algorithms

`numlib::jacobi` doesn't use `ifactor`.

If a is an integer and m is an odd integer not coprime to a then by definition the Jacobi Symbol `(a | m)` is zero.

See Also

MuPAD Functions

`numlib::isquadres` | `numlib::legendre`

numlib::Lambda

Von Mangoldt's function

Syntax

```
numlib::Lambda(m)
```

Description

`numlib::Lambda(m)` returns the value of von Mangoldt's function at `m`.

It is an error if `m` is a number but not a natural number.

If `m` is not a number, `numlib::Lambda` returns the unevaluated function call.

Examples

Example 1

`numlib::Lambda` takes on non-zero values only for prime powers:

```
numlib::Lambda(49)
```

$$\ln(7)$$

```
numlib::Lambda(48)
```

$$0$$

Example 2

`numlib::Lambda` returns the function call if its argument is not a number:

```
numlib::Lambda(3+n^4)
```

```
numlib::Lambda(n^4 + 3)
```

Parameters

`m`

Arithmetical expression

Return Values

`numlib::Lambda` returns an arithmetical expression

Algorithms

The function value of `Lambda` at `m` is defined to be $\log p$ if $m = p^n$ for some prime number p and some positive integer n, and to be zero for positive integers that are not prime powers.

See Also

MuPAD Functions

`numlib::ispower`

numlib::lambda

Carmichael function

Syntax

```
numlib::lambda(n)
```

Description

`numlib::lambda(n)` returns the value of the Carmichael function at `n`.

If `m` is a natural number then `numlib::lambda(m)` returns the value of the Carmichael function in `m`, i. e., the maximal order of an element in the group of units modulo `m`.

`numlib::lambda` returns an error if the argument evaluates to a number of wrong type. `numlib::lambda` returns the function call with its argument evaluated if `m` is not a number.

Examples

Example 1

We compute the value of the Carmichael function λ in 97:

```
numlib::lambda(97)
```

```
96
```

Example 2

We compute the value of the Carmichael function λ in 96:

```
numlib::lambda(96)
```

```
8
```

Parameters

n

A natural number

Return Values

`numlib::lambda(n)` returns a natural number, or the function call with its argument evaluated.

Algorithms

Internally, `ifactor` is used for factoring `n`.

See Also

MuPAD Functions

`numlib::order` | `numlib::phi`

numlib::legendre

Legendre symbol

Syntax

```
numlib::legendre(a, p)
```

Description

`numlib::legendre(a, p)` returns the Legendre symbol `(a | p)`.

`numlib::legendre` returns an error if one of its arguments evaluates to a number of wrong type.

`numlib::legendre` returns the function call with evaluated arguments if at least one of its arguments does not evaluate to a number.

Examples

Example 1

Computing the Legendre symbol `(132132 | 3231277)`:

```
numlib::legendre(132132,3231227)
```

```
1
```

Example 2

Computing the Legendre symbol `(132131 | 3231277)`:

```
numlib::legendre(132131,3231227)
```

```
-1
```

Example 3

Computing the Legendre symbol `(-303 | 101)`:

```
numlib::legendre(-303,101)
```

```
    0
```

Parameters

a

An integer

p

An odd prime

Return Values

`numlib::legendre(a,p)` returns `-1`, `0`, `1`, or the function call with evaluated arguments.

Algorithms

If p is an odd prime and if a is an integer divisible by p then by definition the Legendre symbol `(a | p)` is zero.

See Also

MuPAD Functions

`numlib::isquadres` | `numlib::jacobi`

numlib::lincongruence

Linear congruence

Syntax

```
numlib::lincongruence(a, b, m)
```

Description

`numlib::lincongruence(a,b,m)` returns an error if one of the arguments evaluates to a number of wrong type.

For integers `a` and `b` and a non-zero integer `m` `numlib::lincongruence(a,b,m)` returns the sorted list of all solutions $x \in \{0, 1, \ldots, m - 1\}$ of the linear congruence $a\,x \equiv b \pmod{m}$ if this congruence is solvable. Otherwise `FAIL` is returned.

For integers `a` and `b` and a non-zero integer `m` `numlib::lincongruence(a,b,m)` returns the sorted list of all solutions $x \in \{0, 1, \ldots, m - 1\}$ of the linear congruence $a\,x \equiv b \pmod{m}$ if this congruence is solvable. Otherwise `FAIL` is returned.

Examples

Example 1

A linear congruence possessing one solution:

```
numlib::lincongruence(7,19,23)
```

```
[6]
```

Example 2

A linear congruence possessing several solutions:

```
numlib::lincongruence(77,209,253)
```

```
[6, 29, 52, 75, 98, 121, 144, 167, 190, 213, 236]
```

Example 3

A linear congruence possessing no solutions:

```
numlib::lincongruence(77,208,253)
```

```
FAIL
```

Parameters

a

An integer

b

An integer

m

A non-zero integer

Return Values

`numlib::lincongruence(a,b,m)` returns a list of nonnegative integers if the linear congruence is solvable.

`numlib::lincongruence(a,b,m)` returns `FAIL` if the linear congruence is not solvable.

`numlib::lincongruence(a,b,m)` returns the function call with its arguments evaluated if one of the arguments is a symbolic expression.

See Also

MuPAD Functions
`numlib::ichrem` | `numlib::mroots` | `numlib::msqrts`

numlib::mersenne

Mersenne primes

Syntax

```
numlib::mersenne(n)
```

```
numlib::mersenne()
```

Description

`numlib::mersenne()` returns the list of known Mersenne primes `p`. For these numbers, the Mersenne number 2^p - 1 is prime.

`numlib::mersenne(n)` returns the `n`th currently known Mersenne prime. The numbers of the Mersenne primes after the 40th prime can change in the future. More Mersenne primes might be found.

Examples

Example 1

The following primes p are known to have the property that the Mersenne number 2^p - 1 is prime:

```
numlib::mersenne()
```

```
[2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689,
  9941, 11213, 19937, 21701, 23209, 44497, 86243, 110503, 132049, 216091, 756839, 859433,
  1257787, 1398269, 2976221, 3021377, 6972593, 13466917, 20996011, 24036583, 25964951,
  30402457, 32582657, 37156667, 42643801, 43112609, 57885161]
```

Example 2

Display the `10`th Mersenne prime:

```
numlib::mersenne(10)
```

```
    89
```

Return Values

Natural number or a list of natural numbers.

References

See `http://www.mersenne.org/`

numlib::moebius

Möbius function

Syntax

```
numlib::moebius(n)
```

Description

`numlib::moebius(n)` returns the value of the Möbius function at `n`.

`numlib::moebius` returns an error if the argument evaluates to a number of wrong type.

If `n` is a natural number `numlib::moebius(n)` returns the value of the Möbius function in `n`.

If `n` is not a number, `numlib::moebius(n)` returns the function call with its argument evaluated.

Examples

Example 1

Computing the value of the Möbius function μ at 99937:

```
numlib::moebius(99937)
```

```
    0
```

Example 2

`numlib::moebius` works for arbitrarily large integers:

```
numlib::moebius(45397369416530795319729696969741061923382 6)
```

-1

Parameters

n

A natural number

Return Values

`numlib::moebius(n)` returns a nonnegative integer.

Algorithms

Internally, `ifactor` is used for factoring `n`.

See Also

MuPAD Functions
`numlib::lambda` | `numlib::phi`

numlib::mroots

Modular roots of polynomials

Syntax

```
numlib::mroots(P, m)
```

Description

`numlib::mroots(P,m)` returns an error if `P` is not a polynomial over the integers or `m` is not a natural number.

For a univariate polynomial `P` over the integers and for a natural number `m` the function call `numlib::mroots(P,m)` returns the sorted list of all integers $x \in \{0, 1, \ldots, m - 1\}$ such that $P(x) \equiv 0 \pmod{m}$.

For a multivariate polynomial `P`, `numlib::mroots(P, m)` returns a lexicographically sorted list of all lists $[x_1, \ldots, x_n]$ of integers between 0 and m - 1 such that $P(x_1, \ldots, x_n) \equiv 0 \pmod{m}$.

For a univariate polynomial `P` over the integers and for a natural number `m` the function call `numlib::mroots(P,m)` returns the sorted list of all integers $x \in \{0, 1, \ldots, m - 1\}$ such that $P(x) \equiv 0 \pmod{m}$.

For a multivariate polynomial `P`, `numlib::mroots(P, m)` returns a lexicographically sorted list of all lists $[x_1, \ldots, x_n]$ of integers between 0 and m - 1 such that $P(x_1, \ldots, x_n) \equiv 0 \pmod{m}$.

Examples

Example 1

Defining a polynomial

```
P := poly(3*T^7 + 2*T^2 + T - 17, [T])
```

$$\text{poly}(3\,T^7 + 2\,T^2 + T - 17, [T])$$

and computing its roots modulo 1751:

```
numlib::mroots(P, 1751)
```

$$[221, 260, 612, 736, 1127, 1496]$$

The polynomial `P` doesn't have roots modulo 1994:

```
numlib::mroots(P, 1994)
```

$$[]$$

Example 2

We use `numlib::mroots` to find all points on a particular elliptic curve modulo 13:

```
numlib::mroots(poly(y^2 - x^3 - x - 2, [x, y]), 13)
```

$$[[1, 2], [1, 11], [2, 5], [2, 8], [6, 4], [6, 9], [7, 1], [7, 12], [9, 5], [9, 8], [12, 0]]$$

Parameters

`P`

A polynomial over the integers

`m`

A natural number

Return Values

If `P` is univariate, `numlib::mroots` returns a list of nonnegative integers. If `P` has more than one variable, `numlib::mroots` returns a list of lists of nonnegative integers.

Algorithms

`numlib::mroots` uses `factor`.

See Also

MuPAD Functions
`numlib::lincongruence` | `numlib::msqrts`

numlib::msqrts

Modular square roots

Syntax

```
numlib::msqrts(a, m)
```

Description

`numlib::msqrts(a,m)` returns the list of all integers $x \in \{0, 1, \ldots, m - 1\}$ such that $x^2 \equiv a \pmod{m}$.

Examples

Example 1

Computing the square roots of 132132 modulo 3231227:

```
numlib::msqrts(132132,3231227)
```

```
[219207, 3012020]
```

Example 2

There are no square roots of 222222 modulo 324899:

```
numlib::msqrts(222222,324899)
```

```
[]
```

Example 3

48884 is a composite number, so a number can have more than two square roots modulo 48884:

```
numlib::msqrts(37,48884)
```

```
[383, 585, 23857, 24059, 24825, 25027, 48299, 48501]
```

Parameters

a

An integer

m

A natural number relatively prime to `a`

Return Values

`numlib::msqrts(a,m)` returns a list of nonnegative integers

Algorithms

`numlib::msqrts` uses D. Shanks' algorithm RESSOL.

See Also

MuPAD Functions

`numlib::lincongruence` | `numlib::mroots`

numlib::numdivisors

Number of divisors of an integer

Syntax

```
numlib::numdivisors(n)
```

Description

`numlib::numdivisors(n)` returns the number of positive divisors of `n`.

`numlib::numdivisors(0)` returns `0`.

`numlib::numdivisors` returns the function call with evaluated argument if the argument is not a number.

`numlib::numdivisors` returns an error if the argument evaluates to a number of wrong type.

`numlib::numdivisors` is the same function as `numlib::tau`.

Examples

Example 1

We compute the number of positive divisors of the number 6746328388800 (one of the highly composite numbers studied by S. Ramanujan in 1915):

```
numlib::numdivisors(6746328388800)
```

```
10080
```

Parameters

n

An integer

Return Values

`numlib::numdivisors(n)` returns a nonnegative integer.

Algorithms

Internally, `ifactor` is used for factoring `n`.

See Also

MuPAD Functions

`numlib::divisors` | `numlib::numprimedivisors` | `numlib::primedivisors`

numlib::numprimedivisors

Number of prime factors of an integer

Syntax

```
numlib::numprimedivisors(n)
```

Description

`numlib::numprimedivisors(n)` returns the number of prime factors of the integer `n`, counted without multiplicity.

`numlib::numprimedivisors` and `numlib::omega` are synonyms.

`numlib::numprimedivisors(0)` returns `0`.

`numlib::numprimedivisors` returns the function call with evaluated argument if the argument is not a number.

`numlib::numprimedivisors` returns an error if the argument evaluates to a number of wrong type.

Examples

Example 1

We compute the number of primes dividing 6746328388800:

```
numlib::numprimedivisors(6746328388800)
```

```
9
```

Parameters

n

An integer

Return Values

`numlib::numprimedivisors(n)` returns a nonnegative integer.

Algorithms

Internally, `ifactor` is used for factoring `n`.

See Also

MuPAD Functions

`numlib::numdivisors` | `numlib::omega` | `numlib::primedivisors`

numlib::omega

Number of prime factors of an integer

Syntax

```
numlib::omega(n)
```

Description

`numlib::omega(n)` returns the number of prime factors of the integer `n`, counted without multiplicity.

`numlib::numprimedivisors` and `numlib::omega` are synonyms.

`numlib::omega(0)` returns `0`.

`numlib::omega` returns the function call with evaluated argument if the argument is not a number.

`numlib::omega` returns an error if the argument evaluates to a number of wrong type.

Examples

Example 1

We compute the number of primes dividing 6746328388800:

```
numlib::numprimedivisors(6746328388800)
```

```
    9
```

Parameters

`n`

An integer

Return Values

`numlib::numprimedivisors(n)` returns a nonnegative integer.

Algorithms

Internally, `ifactor` is used for factoring `n`.

See Also

MuPAD Functions

`numlib::numdivisors` | `numlib::numprimedivisors` | `numlib::primedivisors`

numlib::Omega

Number of prime divisors (with multiplicity)

Syntax

```
numlib::Omega(a)
```

Description

`numlib::Omega(a)` returns, for a given positive integer `a`, the finite sum $\sum_p \alpha(p, a)$,

where p runs through all primes, and $\alpha(p, a)$ denotes the highest exponent for which p^α divides `a`.

Examples

Example 1

In contrast to `numlib::numprimedivisors`, the prime factor 2 of 120 is counted thrice:

```
numlib::Omega(120)
```

```
5
```

The same happens here:

```
numlib::Omega(8)
```

```
3
```

Parameters

a

Positive integer

Return Values

`numlib::Omega` returns a positive integer.

See Also

MuPAD Functions
`numlib::numprimedivisors`

numlib::order

Order of a residue class

Syntax

```
numlib::order(a, m)
```

Description

`numlib::order(a,m)` returns the order of the residue class modulo `m` of `a` in the group of units modulo `m` if `a` and `m` are coprime.

`numlib::order(a,m)` returns the function call with its arguments evaluated if `a` or `m` is not a number.

`numlib::order` returns an error if one of the arguments evaluates to a number of wrong type.

Examples

Example 1

We compute the order of the residue class of 23 in the unit group modulo 2161:

```
numlib::order(23, 2161)
```

```
2160
```

Example 2

We compute the order of all elements in the unit group modulo 13:

```
map([$ 1..12],numlib::order,13)
```

```
[1, 12, 3, 6, 4, 12, 12, 4, 3, 6, 12, 2]
```

Example 3

The residue class of 7 is not a unit in the ring $\mathbb{Z}$ modulo 21:

```
numlib::order(7,21)
```

```
FAIL
```

Parameters

a

An integer

m

A natural number

Return Values

`numlib::order(a,m)` returns a natural number if `a` is coprime to `m`, and `FAIL` if `a` is not coprime to `m`.

Algorithms

`numlib::order` uses `ifactor` and `numlib::phi`.

See Also

MuPAD Functions

`numlib::lambda` | `numlib::phi`

numlib::phi

Euler phi function

Syntax

```
numlib::phi(n)
```

Description

`numlib::phi(n)` calculates the Euler φ function of the argument `n`, i.e. the number of numbers smaller than $|n|$ which are relatively prime to `n`. Cf. “Example 1” on page 20-70.

`numlib::phi` returns an error if the argument is a number but not an integer unequal to zero.

`numlib::phi` returns the function call with evaluated arguments if the argument is not a number. Cf. “Example 2” on page 20-70.

Examples

Example 1

`numlib::phi` works on integers unequal zero:

```
numlib::phi(-7), numlib::phi(10)
```

```
6, 4
```

Example 2

`numlib::phi` is returned as a function call with evaluated argument:

```
x := a: numlib::phi(x)
```

numlib::phi(*a*)

Parameters

n

Integer not equal to zero

Return Values

`numlib::phi` returns a positive integer, if the argument evaluates to an integer unequal zero. If the argument cannot be evaluate to a number, the function call with evaluated arguments is returned .

Overloaded By

n

See Also

MuPAD Functions
`numlib::invphi`

numlib::pi

Number of primes up to a given bound

Syntax

```
numlib::pi(x)
```

Description

`numlib::pi(x)` returns the number of primes not exceeding *x*.

If the argument `x` is a real number (an integer, rational, or floating-point number), then the number of primes below `x` is returned. If `x` is a complex number, `numlib::pi` stops with an error. For every other kind of arithmetical expression `x`, an unevaluated call is returned.

`numlib::pi` becomes slightly faster if the internal prime number table is large. `ifactor(PrimeLimit)` displays the limit of the internal prime number table; it can be set by the user via the command line flag `-L`.

Internally, a fast kernel function with constant memory consumption is used for the computation.

Examples

Example 1

There are two primes less or equal 3:

```
numlib::pi(3)
```

```
2
```

Example 2

Also larger inputs can be handled fast:

```
numlib::pi(150000000)
```

```
8444396
```

Example 3

Floating point arguments are allowed, too.

```
numlib::pi(28.72)
```

```
9
```

Parameters

x

An arithmetical expression

Return Values

Non-negative integer or an unevaluated call to `numlib::pi`

Algorithms

A Lehmer-type algorithm is used, with no precomputed sieve array and no remember tables. In contrast to the algorithm in “Computing π: The Meissel-Lehmer method”, this means constant memory consumption, at the price of slowness.

References

[1] Lagarias, J.C., V.S. Miller, and A.M. Odlyzko. “Computing π: The Meissel-Lehmer method”, Math. Comp., Vol. 44, No. 170 (1985), pp. 537-560

See Also

MuPAD Functions
`isprime` | `ithprime` | `nextprime` | `prevprime`

numlib::proveprime

Primality proving using elliptic curves

Syntax

```
numlib::proveprime(n)
```

Description

`numlib::proveprime(n)` tests whether `n` is a prime. Unlike `isprime`, `numlib::proveprime` always returns a correct answer.

`numlib::proveprime` returns the following values:

- `TRUE` when it can prove that the number is a prime.
- `FALSE` when it can prove that the number is not a prime.
- `FAIL` when it cannot prove that the number is a prime and cannot prove otherwise. In such cases, the input most likely is a prime.
- A primality certificate, which is a list or a sequence of lists containing proof for the primality of the number. Typically, `numlib::proveprime` returns primality certificates for very large numbers.

A primality certificate is a sequence of lists of the form [N, D, l_m, a, b, x, y, l_s], where

- N is a pseudoprime
- D is an integer (fundamental discriminant)
- l_m is a list of prime factors
- a, b, x, y are integers modulo N
- l_s is another list of prime factors (subset of the factors in l_m)

`numlib::checkPrimalityCertificate` checks primality certificates produced by `numlib::proveprime`.

Examples

Example 1

Proving that 1159523 is prime can be reduced to proving that 10343 is prime:

```
certificate := numlib::proveprime(1159523)
```

```
[1159523, 7, [2, 2, 2, 2, 7, 10343], 1012280, 1061361, 0, 726669, [10343]]
```

Example 2

Typically, the primality of the input is reduced to the primality of a smaller integer, the primality of that integer is reduced to the primality of an even smaller integer, and so on.

```
numlib::proveprime(179424673)
```

```
[179424673, 7, [2, 2, 2, 2, 11213771], 116768436, 77845624, 1, 110562161, [11213771]],
  [11213771, 7, [2, 2, 2, 2, 487, 1439], 9255809, 9908463, 2, 1041453, [487, 1439]]
```

Use `numlib::checkPrimalityCertificate` to check the result:

```
numlib::checkPrimalityCertificate(%)
```

```
TRUE
```

Parameters

n

Positive integer.

Return Values

`TRUE`, `FALSE`, `FAIL`, or a list or sequence of lists.

References

`numlib::proveprime` implements the Atkin-Goldwasser-Kilian-Morain algorithm for proving primality. For information about primality proving and this particular algorithm, see:

- Atkin, A. O., and F. Morain. “Elliptic curves and primality proving.” *Mathematics of Computation*. Vol. 61, Number 203, 1993.
- Goldwasser, S., and J. Kilian. “Almost all primes can be quickly certified”. *Proceedings of the 18th annual ACM symposium on theory of computing*. Berkeley, CA, US, 1986, pp. 316–329.

See Also

MuPAD Functions

`ifactor` | `isprime` | `ithprime` | `nextprime` | `prevprime`

numlib::primedivisors

Prime factors of an integer

Syntax

```
numlib::primedivisors(n)
```

Description

`numlib::primedivisors(n)` returns a list containing the different prime divisors of the integer `n`.

If `a` is a non-zero integer then, `numlib::primedivisors(a)` returns the sorted list of the different prime divisors of `a`.

`numlib::primedivisors(0)` returns `[0]`.

`numlib::primedivisors` returns the function call with evaluated argument if the argument is not a number.

`numlib::primedivisors` returns an error if the argument evaluates to a number of wrong type.

Examples

Example 1

We compute the list of prime divisors of the number 6746328388800 (one of the highly composite numbers studied by S. Ramanujan in 1915):

```
numlib::primedivisors(6746328388800)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23]
```

Parameters

n

An integer

Return Values

`numlib::primedivisors(n)` returns a list of nonnegative integers.

Algorithms

Internally, `ifactor` is used for factoring `n`.

See Also

MuPAD Functions

`ifactor` | `isprime` | `numlib::divisors` | `numlib::numdivisors` | `numlib::numprimedivisors` | `numlib::proveprime`

numlib::primroot

Primitive roots

Syntax

```
numlib::primroot(m)
```

```
numlib::primroot(a, m)
```

Description

`numlib::primroot(m)` returns the least positive primitive root modulo `m` if there exist primitive roots modulo `m`.

`numlib::primroot(a, m)` returns the least primitive root modulo `m` not smaller than `a` if there exist primitive roots modulo `m`.

Examples

Example 1

We compute the least positive primitive root modulo the prime number 40487:

```
numlib::primroot(40487)
```

```
     5
```

Example 2

We compute the least primitive root modulo $40487^2 = 1639197169$:

```
numlib::primroot(1639197169)
```

```
     10
```

Example 3

Now we compute least primitive root modulo 40487 which is >=111111111:

```
numlib::primroot(111111111,40487)
```

```
111111116
```

Example 4

There are no primitive roots modulo 324013370:

```
numlib::primroot(324013370)
```

```
FAIL
```

Parameters

a

An integer

m

A natural number

Return Values

`numlib::primroot` returns an integer or `FAIL`.

Algorithms

`numlib::primroot` uses `ifactor`.

See Also

MuPAD Functions
`numlib::order`

numlib::reconstructRational

Rational number reconstruction

Syntax

```
numlib::reconstructRational(a,n)
```

Description

`numlib::reconstructRational(a,n)` returns two integers `p,q` of absolute value smaller than `sqrt(n/2)` with `p` congruent to `a*q` modulo `n`. It returns `FAIL` if such `p,q` do not exist.

`numlib::reconstructRational(a,n)` returns `p,q` by solving $p \equiv a\,q \pmod{n}$.

The solution `p,q` satisfies the following conditions: `p` is strictly between `-sqrt(n/2)` and `sqrt(n/2)`, `q` is strictly between `0` and `sqrt(n/2)`.

If several pairs `p,q` satisfy these conditions, then their ratios `p/q` are the same. In such case, `numlib::reconstructRational` returns the smallest of these pairs.

Examples

Example 1

Solve this linear congruence: $p \equiv 7\,q \pmod{12}$.

```
numlib::reconstructRational(7, 12)
```

$$2, 2$$

Modulo 98, the same congruence has no small solution. The solution `p=7, q=1` is not small enough as `7` is not smaller than `sqrt(98/2)` but just equal.

```
numlib::reconstructRational(7, 98)
```

```
FAIL
```

Example 2

Rational number reconstruction is mostly used as the last step of a modular algorithm. For example, find the greatest common divisors of the following polynomials.

```
f:=  poly(x^5 + 22/35*x^3 + 3/8*x^2 + 3/35*x + 9/56, [x]):
g:= poly(x^5 + 2/5*x^4 + 22/35*x^3 + 153/280*x^2 + 3/35*x + 9/56,
    [x]):
```

Typically, you can use `gcd` for this task. However, suppose you know that the greatest common divisor has small coefficients with numerator and denominator both smaller than 10. Then you can use a modular algorithm with a smaller modulus than `gcd` would do: to be able to reconstruct these from their residue class modulo `n`, it is sufficient that $\sqrt{\frac{n}{2}} > 10$, e.g., `n=211`.

```
gcd(poly(f, IntMod(211)), poly(g, IntMod(211)))
```

$$\mathrm{poly}(x^2 - 90, [x], \mathrm{IntMod}(211))$$

Rational number reconstruction shows that the constant coefficient must be `3/7`:

```
numlib::reconstructRational(-90, 211)
```

$$3, 7$$

```
gcd(f,g)
```

$$\mathrm{poly}\left(x^2 + \frac{3}{7}, [x]\right)$$

Parameters

a

An integer

n

A positive integer

Return Values

Sequence consisting of an integer and a positive integer, or `FAIL`

References

[1] Davenport, J. H. , Y.Siret, and E.Tournier “Computer Algebra: Systems and Algorithms for Algebraic Computation”. Academic Press Inc, 1988, p.142

See Also

MuPAD Functions

`numlib::lincongruence`

numlib::sigma

Sum of divisors of an integer

Syntax

```
numlib::sigma(n)
```

```
numlib::sigma(n, k)
```

Description

`numlib::sigma(n)` returns the sum of the positive divisors of `n`.

`numlib::sigma(n, k)` returns the sum of the `k`-th powers of the positive divisors of `n`.

`numlib::sigma(0)` returns `0`.

`numlib::sigma` returns the function call with evaluated argument if at least one argument is not a number.

`numlib::sigma` returns an error if one of its arguments evaluates to a number of wrong type.

`numlib::sigma(n,0)` is the same as `numlib::numdivisors(n)` and `numlib::tau(n)`.

`numlib::sigma(n,1)` is the same function as `numlib::sumdivisors(n)` and `numlib::sigma(n)`.

Examples

Example 1

The sum of the positive divisors of 120 is 360:

```
numlib::sigma(120)
```

```
360
```

Example 2

The sum of the fifth powers of the positive divisors of 120 is 25799815800:

```
numlib::sigma(120,5)
```

```
25799815800
```

Parameters

n

An integer

k

A nonnegative integer

Return Values

`numlib::sigma` returns an integer.

Algorithms

Internally, `ifactor` is used for factoring `n`.

See Also

MuPAD Functions

`numlib::divisors` | `numlib::numdivisors`

numlib::sqrt2cfrac

Continued fraction expansion of square roots

Syntax

```
numlib::sqrt2cfrac(a)
```

Description

`numlib::sqrt2cfrac(a)` returns the continued fraction expansion of the square root of `a` as a sequence of two lists: the first one contains the non-periodic (integer) part, the second one contains the periodic part of the expansion.

Examples

Example 1

The square root of 87 can be written as $9 + q$, where q is a number satisfying $q = \frac{1}{3+\frac{1}{18+q}}$:

```
numlib::sqrt2cfrac(87)
```

```
[9], [3, 18]
```

Example 2

Since 81 is a perfect square, there is no periodic part in the continued fraction expansion of its square root:

```
numlib::sqrt2cfrac(81)
```

```
[9]
```

Parameters

a

A positive integer

Return Values

If `a` is a perfect square, `numlib::sqrt2cfrac` returns a list with one entry; otherwise `numlib::sqrt2cfrac` returns a sequence of two lists, the first consisting of one integer, the second consisting of one or more integers.

See Also

MuPAD Functions
`numlib::contfrac`

numlib::sqrtmodp

Square root of a quadratic residue modulo a prime

Syntax

```
numlib::sqrtmodp(a, p)
```

Description

`numlib::sqrtmodp(a, p)` computes a solution x to the congruence $x^2 \equiv a \pmod{p}$.

`numlib::sqrtmodp(a, p)` computes an integer x that satisfies $x^2 \equiv a \pmod{p}$.

`a` must be a quadratic residue modulo `p`, and `p` must be a prime. This is not checked! Unless this is known to be the case, `numlib::msqrts` must be used. On the other hand, `numlib::sqrtmodp` is faster than `numlib::msqrts`.

Examples

Example 1

One square root of 132132 modulo 3231227 is 3012020:

```
numlib::sqrtmodp(132132,3231227)
```

```
3012020
```

Parameters

a

An integer

p

A prime unequal to 2

Return Values

`numlib::sqrtmodp` returns an integer.

Algorithms

`numlib::sqrtmodp` uses D. Shanks' algorithm RESSOL.

See Also

MuPAD Functions
`numlib::msqrts`

numlib::sumdivisors

Sum of divisors of an integer

Syntax

```
numlib::sumdivisors(n)
```

Description

`numlib::sumdivisors(n)` returns the sum of the positive divisors of the integer `n`.

`numlib::sumdivisors(0)` returns `0`.

`numlib::sumdivisors` returns the function call with evaluated argument if the argument is not a number.

`numlib::sumdivisors` returns an error if the argument evaluates to a number of wrong type.

`numlib::sumdivisors(n)` is the same as `numlib::sigma(n, 1)`.

Examples

Example 1

The sum of the positive divisors of 120 is 360:

```
numlib::sumdivisors(120)
```

```
360
```

Example 2

The sum of the positive divisors of - 63 is 104:

```
numlib::sumdivisors(-63)
```

```
     104
```

Parameters

n

An integer

Return Values

`numlib::sumdivisors(n)` returns a nonnegative integer.

Algorithms

Internally, `ifactor` is used for factoring `n`.

See Also

MuPAD Functions

`numlib::divisors` | `numlib::numdivisors` | `numlib::sigma`

numlib::sumOfDigits

Sum of digits of an integer

Syntax

```
numlib::sumOfDigits(n, <base>)
```

Description

`numlib::sumOfDigits(n, base)` computes the sum of digits of `n` in the given base `base`); if the base is not given, it defaults to 10.

The sum of digits may be larger than the base. For certain purposes (testing divisibility by b - 1, where b is the base), it may be useful to apply `numlib::sumOfDigits` over and over to the result. This is not done automatically. See “Example 2” on page 20-94.

Examples

Example 1

We compute the decimal and the binary sum of digits of 11:

```
numlib::sumOfDigits(11), numlib::sumOfDigits(11, 2)
```

```
2, 3
```

Example 2

We want to test whether 9 divides a given number, using the school method:

```
n:= 24373463462374324:
repeat n:= numlib::sumOfDigits(n); print(n) until n < 10 end:
delete n:
```

```
67
```

```
13
```

```
4
```

This only makes sense for demonstration purposes, as the following command achieves the same but much faster:

```
24373463462374324 mod 9
```

```
4
```

Parameters

n

Non-negative integer

base

Integer greater than one

Return Values

Non-negative integer

See Also

MuPAD Functions
`numlib::g_adic`

numlib::tau

Number of divisors of an integer

Syntax

```
numlib::tau(n)
```

Description

`numlib::tau(n)` returns the number of positive divisors of `n`.

`numlib::tau(0)` returns `0`.

`numlib::tau` returns the function call with evaluated argument if the argument is not a number.

`numlib::tau` returns an error if the argument evaluates to a number of wrong type.

`numlib::tau` is the same function as `numlib::numdivisors`.

Examples

Example 1

We compute the number of positive divisors of the number 6746328388800 (one of the highly composite numbers studied by S. Ramanujan in 1915):

```
numlib::tau(6746328388800)
```

```
10080
```

Parameters

n

An integer

Return Values

`numlib::tau` returns a nonnegative integer.

Algorithms

Internally, `ifactor` is used for factoring `n`.

See Also

MuPAD Functions

`numlib::divisors` | `numlib::numprimedivisors` | `numlib::primedivisors`

numlib::toAscii

ASCII encoding of a string

Syntax

```
numlib::toAscii(s)
```

Description

`numlib::toAscii(s)` returns the list of ASCII codes of the characters in the string `s`.

`numlib::toAscii` returns an error if its argument is not a string.

Examples

Example 1

The ASCII coding of a well-known name:

```
numlib::toAscii("MuPAD - Multi Processing Algebra Data Tool")
```

```
[77, 117, 80, 65, 68, 32, 45, 32, 77, 117, 108, 116, 105, 32, 80, 114, 111, 99, 101, 115, 115, 105,
  110, 103, 32, 65, 108, 103, 101, 98, 114, 97, 32, 68, 97, 116, 97, 32, 84, 111, 111, 108]
```

and the ASCII coding of an empty string:

```
numlib::toAscii("")
```

```
[]
```

Parameters

s

A string

Return Values

`numlib::toAscii(s)` returns a list of nonnegative integers.

See Also

MuPAD Functions
`numlib::fromAscii`

ode – Ordinary Differential Equations

ode::companionSystem
ode::cyclicVector
ode::dAlembert
ode::evalOde
ode::exponentialSolutions
ode::exponents
ode::getOrder
ode::indicialEquation
ode::isFuchsian
ode::isLODE
ode::mkODE
ode::normalize
ode::polynomialSolutions
ode::rationalSolutions
ode::ratSys
ode::scalarEquation
ode::series
ode::solve
ode::symmetricPower
ode::unimodular
ode::vectorize
ode::wronskian

ode::companionSystem

Companion matrix of a linear homogeneous ordinary differential equation

Syntax

```
ode::companionSystem(Ly, y(x), <R>)
```

Description

`ode::companionSystem(Ly, y(x))` returns the companion matrix associated to `Ly`. If the optional argument `R` is given, the elements of the matrix are in `R`.

Examples

Example 1

We compute the companion matrix of the following differential equation:

```
Ly := 4*x^2*diff(y(x),x$3)+diff(y(x),x$2)+4*x*diff(y(x),x)-y(x)
```

$$4\,x\,\frac{\partial}{\partial x}\,y(x)-y(x)+4\,x^2\,\frac{\partial^3}{\partial x^3}\,y(x)+\frac{\partial^2}{\partial x^2}\,y(x)$$

```
ode::companionSystem(Ly, y(x))
```

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ \frac{1}{4\,x^2} & -\frac{1}{x} & -\frac{1}{4\,x^2} \end{pmatrix}$$

Parameters

`Ly`

A linear homogeneous ordinary differential equation.

`y(x)`

The dependent function of `Ly`.

`R`

A field of functions or numbers of characteristic zero, default is `Dom::ExpressionField(normal)`.

Return Values

Object of type `Dom::Matrix`.

ode::cyclicVector

Transforms a linear differential system to an equivalent linear differential system with a companion matrix.

Syntax

```
ode::cyclicVector(A, x, <v>)
```

Description

`ode::cyclicVector(A, x, v)` converts a first order homogeneous differential system $Y' = AY$ into a corresponding first order homogeneous differential system $Z' = BZ$, where `B` is a companion matrix, by substituting $Z = PY$ using the potential cyclic vector `v`. If `v` is not cyclic then an empty list is returned otherwise a list is returned whose first element is a list corresponding to the last row of `B` and second element is the invertible matrix `P`.

When the optional argument `v` is not given then the vector `[1,0,...,0]` is tested. If it is not cyclic then a suitable one is determined randomly by the procedure.

Examples

Example 1

We compute a differential system equivalent to the following differential system:

```
A := matrix( [ [x^2-1,1,0], [0,x^2+5*x+1/3,1], [0,0,2]])
```

$$\begin{pmatrix} x^2-1 & 1 & 0 \\ 0 & x^2+5x+\frac{1}{3} & 1 \\ 0 & 0 & 2 \end{pmatrix}$$

```
ode::cyclicVector(A, x)
```

$$\left[\left[2x^4+6x^3-\frac{49x^2}{3}-\frac{38x}{3}+\frac{19}{3},\ -x^4-5x^3-\frac{10x^2}{3}+x+\frac{20}{3},\ 2x^2+5x+\frac{4}{3}\right],\right.$$

$$\left.\begin{pmatrix} 1 & 0 & 0 \\ x^2-1 & 1 & 0 \\ 2x-2x^2+x^4+1 & 5x+2x^2-\frac{2}{3} & 1 \end{pmatrix}\right]$$

So `[1,0,0]` is a cyclic vector; `[x,0,0]` is also a cyclic vector:

```
l := ode::cyclicVector(A, x, [x,0,0])
```

$$\left[\left[\frac{6x^7+21x^6-34x^5-28x^4+28x^3+10x^2+8x+18}{3x^3},\right.\right.$$

$$\left.\left.-\frac{3x^6+15x^5+10x^4+9x^3+10x^2+8x+18}{3x^2},\ \frac{6x^3+15x^2+4x+9}{3x}\right],\right.$$

$$\left.\begin{pmatrix} x & 0 & 0 \\ -x+x^3+1 & x & 0 \\ x+4x^2-2x^3+x^5-2 & -\frac{2x}{3}+5x^2+2x^3+2 & x \end{pmatrix}\right]$$

And we can build easily a linear homogeneous differential equation associated to it (c.f. `ode::mkODE`):

```
-ode::mkODE(l[1].[-1], y, x)
```

$$\frac{(3x^6+15x^5+10x^4+9x^3+10x^2+8x+18)\,\frac{\partial}{\partial x}y(x)}{3x^2}-\frac{(6x^3+15x^2+4x+9)\,\frac{\partial^2}{\partial x^2}y(x)}{3x}$$
$$-\frac{y(x)\,(6x^7+21x^6-34x^5-28x^4+28x^3+10x^2+8x+18)}{3x^3}+\frac{\partial^3}{\partial x^3}y(x)$$

Parameters

A

A square matrix of type `Dom::Matrix`.

x

The independent variable.

v

A list of size the dimension of `A`, default is `[1,0,...,0]`.

Return Values

List, possibly empty, of two lists.

See Also

MuPAD Functions
`ode::scalarEquation`

ode::dAlembert

D'Alembert reduction of a linear homogeneous ordinary differential equation

Syntax

```
ode::dAlembert(Ly, y(x), v)
```

Description

`ode::dAlembert(Ly, y(x), v)` returns the reduced differential equation of `Ly` using the method of reduction of d'Alembert and the function `v`. If `v` is a solution of `Ly` and `u` is a solution of the reduced differential equation then $v \in t\,u$ is another solution of `Ly`.

Examples

Example 1

Consider the following differential equation:

```
Ly := 2/x^3*y(x)-2/x^2*diff(y(x),x)+1/x*diff(y(x),x$2)+
      diff(y(x),x$3)
```

$$\frac{2\,y(x)}{x^3} - \frac{2\,\frac{\partial}{\partial x}y(x)}{x^2} + \frac{\frac{\partial^2}{\partial x^2}y(x)}{x} + \frac{\partial^3}{\partial x^3}y(x)$$

We easily check that `x` is a particular solution of `Ly`:

```
ode::evalOde(Ly, y(x)=x)
```

$$0$$

Then we reduce the equation `Ly` using this special solution:

```
R := ode::dAlembert(Ly, y(x), x)
```

$$\frac{4\,\frac{\partial}{\partial x}\,y(x)}{x}+\frac{\partial^2}{\partial x^2}\,y(x)$$

The solutions of the equation R are not too hard to find:

```
ode::evalOde(R, y(x)=1), ode::evalOde(R, y(x)=1/x^3)
```

$$0,\ 0$$

So a basis of solutions of Ly is therefore $\left\{x,\ x\int 1\,\mathrm{d}x=x^2,\ x\int\frac{1}{x^3}\,\mathrm{d}x=-\frac{1}{2x}\right\}$ which can be checked directly:

```
ode::solve(Ly, y(x))
```

$$\left\{\frac{C4}{x}-\frac{x\,(C3+2\,C2\,x)}{6}\right\}$$

Parameters

Ly

A homogeneous linear differential equation.

y(x)

The dependent function of Ly.

v

An expression.

Return Values

Expression.

ode::evalOde

Applies an expression at a linear ordinary differential equation

Syntax

```
ode::evalOde(Ly, y(x))
```

Description

`ode::evalOde(Ly, y(x) = v)` evaluates `Ly` replacing `y(x)` by `v` and simplifying the result. This can be useful to check a solution candidate, for example.

Examples

Example 1

We evaluate the following differential equation for various expressions:

```
Ly := (x^2+1)*diff(y(x),x$2)+x*diff(y(x),x)-4*y(x)
```

$$(x^2+1)\,\frac{\partial^2}{\partial x^2}\,y(x)+x\,\frac{\partial}{\partial x}\,y(x)-4\,y(x)$$

```
ode::evalOde(Ly, y(x) = 2*x^2+1),
ode::evalOde(Ly, y(x) = exp(x)),
ode::evalOde(Ly, y(x) = RootOf(Z^3+x*Z+1, Z))
```

$$0,\ x^2\,e^x-3\,e^x+x\,e^x,\ -18\,Z^2\,x^4-54\,Z^2\,x^2+243\,Z^2\,x-60\,Z\,x^6-4\,Z\,x^4-756\,Z\,x^3+54\,Z\,x$$

$$-2916\,Z-12\,x^5-36\,x^3+162\,x^2$$

Parameters

`Ly`

A linear ordinary differential equation.

`y(x)`

The dependent function of `Ly`.

`v`

An expression.

Return Values

Expression.

ode::exponentialSolutions

Exponential solutions of a homogeneous linear ordinary differential equation

Syntax

```
ode::exponentialSolutions(Ly, y(x), <Generic>)
```

Description

`ode::exponentialSolutions(Ly, y(x))` returns a fundamental set of the exponential solutions of `Ly`, i.e. solutions `z` such that $\frac{z'}{z}$ is a rational function of `x`. When the option `Generic` is given, a generic form of them is returned.

Note: `ode::exponentialSolutions` does not return any eventual solution that is exponential over the algebraic closure of $\mathbb{Q}(x)$ but not over $\mathbb{Q}(x)$.

Examples

Example 1

We compute the exponential solutions of the following differential equation :

```
Ly:=diff(y(x),x$4)-2*x*diff(y(x),x$3)+(-x+x^2-5)*diff(y(x),x$2)+
    (4*x+2*x^2)*diff(y(x),x)+(2+x-x^3)*y(x)
```

$$\frac{\partial^4}{\partial x^4} y(x) - 2\,x\,\frac{\partial^3}{\partial x^3} y(x) - (-x^2+x+5)\,\frac{\partial^2}{\partial x^2} y(x) + (2\,x^2+4\,x)\,\frac{\partial}{\partial x} y(x) + y(x)\,(-x^3+x+2)$$

```
ode::exponentialSolutions(Ly, y(x)),
ode::exponentialSolutions(Ly, y(x), Generic)
```

$$\left\{ e^{\frac{x^2}{2}}, x\,e^{\frac{x^2}{2}} \right\}, C2\,e^{\frac{x^2}{2}} + C1\,x\,e^{\frac{x^2}{2}}$$

Example 2

No exponential solution over the algebraic closure of $\mathbb{Q}(x)$ is returned:

```
ode::exponentialSolutions(diff(y(x),x$2)+y(x), y(x))
```

$$\varnothing$$

whereas $\left\{e^{ix},\ e^{-ix}\right\}$ is a basis of solutions of the above differential equation.

Parameters

`Ly`

A homogeneous linear ordinary differential equation with coefficients in the field $\mathbb{Q}(x)$ of rational functions over the rationals.

`y(x)`

The dependent function of `Ly`.

Return Values

`set`, possibly empty, of functions or an expression.

ode::exponents

Exponents of a linear ordinary differential equation

Syntax

```
ode::exponents(Ly, y(x), p)
```

Description

`ode::exponents` returns the set of exponents of a homogeneous linear differential equation at a given point.

`ode::exponents(Ly, y(x), p)` returns the set of (local) exponents of `Ly` at the place `p`. If the place is infinity then one uses $\frac{1}{x}$ instead. They are defined as roots (in an algebraic closure of $\mathbb{Q}(x)$) of the indicial equation (c.f. `ode::indicialEquation`) of `Ly` so the set of exponents may be empty, see "Example 2" on page 21-14.

Examples

Example 1

We compute the exponents of the following differential equation at the regular point `0` and at the singular points `-1` and *infinity*:

```
Ly := diff(y(x),x$2)+4/(x+1)*diff(y(x),x)+2/(x+1)^2*y(x)
```

$$\frac{4\,\frac{\partial}{\partial x}y(x)}{x+1}+\frac{2\,y(x)}{(x+1)^2}+\frac{\partial^2}{\partial x^2}y(x)$$

```
ode::exponents(Ly, y(x), x)
```

$$\{0,\ 1\}$$

```
ode::exponents(Ly, y(x), x+1)
```

$$\{-2, -1\}$$

```
ode::exponents(Ly, y(x), 1/x)
```

$$\{1, 2\}$$

Example 2

It may happen that at a place the set of exponents is empty; this corresponds to an irregular singular point:

```
Ly := (2*x+4)*diff(y(x),x)/(2*x+x^2-2)-2*y(x)/(2*x+x^2-2)-
      (4*x+x^2)/(2*x+x^2-2)*diff(y(x),x$2)+diff(y(x),x$3)
```

$$\frac{(2x+4)\,\frac{\partial}{\partial x}y(x)}{x^2+2x-2} - \frac{(x^2+4x)\,\frac{\partial^2}{\partial x^2}y(x)}{x^2+2x-2} - \frac{2\,y(x)}{x^2+2x-2} + \frac{\partial^3}{\partial x^3}y(x)$$

```
ode::exponents(Ly, y(x), 1/x)
```

$$\varnothing$$

```
ode::exponents(Ly, y(x), x^2+2*x-2)
```

$$\{0, 1, 3\}$$

Parameters

Ly

A homogeneous linear differential equation over $\mathbb{Q}(x)$.

y(x)

The dependent function of `Ly`.

p

An irreducible polynomial in `x` or `1/x`.

Return Values

`set`, possibly empty.

See Also

MuPAD Functions

`ode::indicialEquation` | `ode::isFuchsian`

ode::getOrder

Order of an ordinary differential equation

Syntax

```
ode::getOrder(Ly, y(x))
```

Description

`ode::getOrder(Ly, y(x))` returns the order of `Ly` for the dependent function `y(x)`, i.e. the highest degree of derivative of `Ly`.

Examples

Example 1

We can compute orders for linear and nonlinear ordinary differential equations:

```
ode::getOrder(diff(y(x),x$2)-x*y(x)-airyAi(x), y(x))
```

$$2$$

```
ode::getOrder(y(x)*diff(y(x),x$3)^2-exp(y(x)), y(x))
```

$$3$$

```
ode::getOrder(y(x)*diff(y(x),x$3)^2-exp(y(x)), z(x))
```

$$-\infty$$

Parameters

`Ly`

An ordinary differential equation.

`y(x)`

The dependent function of `Ly`.

Return Values

Either `-infinity` or a positive integer.

ode::indicialEquation

Indicial equation of a linear ordinary differential equation

Syntax

```
ode::indicialEquation(Ly, y(x), p, u)
```

Description

`ode::indicialEquation(Ly, y(x), p, u)` returns the indicial equation in the variable `u` of `Ly` at the place `p`. If the place is infinity then one uses $\frac{1}{x}$ instead. The result if `FAIL` if the place corresponds to an irregular singular point of `Ly`.

Examples

Example 1

We compute the indicial equations of the following differential equation at the regular point `1` and at the singular points `0` and *infinity*:

```
Ly := 1/x^3*y(x)*(4*x-10)-1/x^2*(4*x^3-10)*diff(y(x),x)-
      9/2/x*diff(y(x),x$2)+diff(y(x),x$3)
```

$$\frac{y(x)\,(4x-10)}{x^3}-\frac{9\,\frac{\partial^2}{\partial x^2}y(x)}{2x}-\frac{(4x^3-10)\,\frac{\partial}{\partial x}y(x)}{x^2}+\frac{\partial^3}{\partial x^3}y(x)$$

```
ode::indicialEquation(Ly, y(x), x-1, U)
```

$$U^3-3\,U^2+2\,U$$

```
ode::indicialEquation(Ly, y(x), 1/x, U)
```

FAIL

```
ode::indicialEquation(Ly, y(x), x, U)
```

$$2\,U^3 - 15\,U^2 + 33\,U - 20$$

The roots of the indicial equation correspond to the (local) exponents:

```
solve(%, U), ode::exponents(Ly, y(x), x)
```

$$\left\{1, \frac{5}{2}, 4\right\}, \left\{1, \frac{5}{2}, 4\right\}$$

Parameters

`Ly`

A homogeneous linear differential equation over $\mathbb{Q}(x)$.

`y(x)`

The dependent function of `Ly`.

`p`

An irreducible polynomial in `x` or `1/x`.

`u`

An element of type `DOM_IDENT`.

Return Values

`FAIL` or a polynomial expression in `u`.

See Also

MuPAD Functions

`ode::indicialEquation` | `ode::isFuchsian`

ode::isFuchsian

Tests if a homogeneous linear ordinary differential equation is of Fuchsian type

Syntax

```
ode::isFuchsian(Ly, y(x), <AllExponents>)
```

Description

`ode::isFuchsian` returns `TRUE` if `Ly` is of Fuchsian type, i.e., all the singular points (including the point at infinity) of `Ly` are regular. It returns `FALSE` if at least one singular point is irregular. When the option `AllExponents` is given, either `FALSE` is returned or a list where each element is a table containing, at each regular singular point of `Ly` the place, the indicial equation and the exponents.

Examples

Example 1

We test if the following differential equation is Fuchsian:

```
Ly:=x*(1-x)*diff(y(x),x$2)+(1-x)*diff(y(x),x)+10*y(x)
```

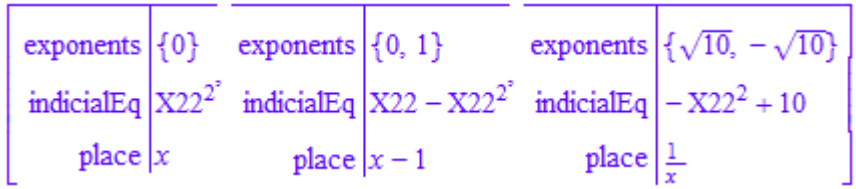
$$-x\,(x-1)\,\frac{\partial^2}{\partial x^2}\,y(x)-(x-1)\,\frac{\partial}{\partial x}\,y(x)+10\,y(x)$$

```
ode::isFuchsian(Ly, y(x))
```

TRUE

We can have a look of the indicial equations, exponents at each regular singular point of `Ly`:

```
ode::isFuchsian(Ly, y(x), AllExponents)
```



Example 2

In this example, the Airy equation, the only singular point is at infinity and is irregular:

```
ode::isFuchsian(diff(y(x),x$2)-x*y(x), y(x))
```

```
FALSE
```

Parameters

`Ly`

A homogeneous linear ordinary differential equation with coefficients in the field $\mathbb{Q}(x)$ of rational functions over the rationals.

`y(x)`

The dependent function of `Ly`.

Options

`AllExponents`

Return a list of tables of indical equations and exponents for regular singular points.

Return Values

`TRUE`, `FALSE` or a list.

ode::isLODE

Test for a linear ordinary differential equation

Syntax

```
ode::isLODE(Ly, y(x), <Homogeneous | HlodeOverRF | Hlode | LodeOverRF | Lode>)
```

Description

`ode::isLODE(Ly, y(x))` returns `TRUE` if `Ly` is a linear ordinary differential equation in `y(x)`, `FALSE` otherwise. If an optional argument is given then the result is discussed as follows:

- `Homogeneous`: returns `TRUE` if `Ly` is homogeneous, `FALSE` otherwise.
- `HlodeOverRF`: returns the sequence `Ly, y, x, n`, where `n` is the order of `Ly`, if `Ly` is homogeneous with rational functions coefficients, `FALSE` otherwise.
- `Hlode`: returns the sequence `Ly, y, x, n`, where `n` is the order of `Ly`, if `Ly` is homogeneous, `FALSE` otherwise.
- `LodeOverRF`: returns the sequence `Ly, y, x, n`, where `n` is the order of `Ly`, if `Ly` has rational functions coefficients, `FALSE` otherwise.
- `Lode`: returns the sequence `Ly, y, x, n`, where `n` is the order of `Ly`, if `Ly` is a linear ordinary differential equation, `FALSE` otherwise.

Examples

Example 1

We test the following differential equations:

```
ode::isLODE(y(x)^2+x^2*diff(y(x),x)+x, y(x))
```

```
FALSE
```

```
ode::isLODE(y(x)+x^2*diff(y(x),x)+x, y(x))
```

```
     TRUE
```

```
ode::isLODE(y(x)+x^2*diff(y(x),x)+x, y(x), Hlode)
```

```
     FALSE
```

```
ode::isLODE(
     y(x)+x^2*diff(y(x),x)+x*diff(y(x),x$2), y(x), HlodeOverRF)
```

$$y(x)+x^2\,\frac{\partial}{\partial x}\,y(x)+x\,\frac{\partial^2}{\partial x^2}\,y(x),\ y,\ x,\ 2$$

```
ode::isLODE(
     x+x^2*diff(y(x),x)+exp(x)*diff(y(x),x$2), y(x), LodeOverRF)
```

```
     FALSE
```

Parameters

`Ly`

An expression.

`y(x)`

The dependent function of `Ly`.

Return Values

Either `TRUE`, `FALSE` or a sequence of type `_exprseq`.

ode::mkODE

Builds a linear homogeneous ordinary differential equation from a list of coefficient functions

Syntax

```
ode::mkODE(l, y, x)
```

Description

`ode::mkODE(l, y, x)` returns a linear homogeneous ordinary differential equation `Ly` in `y(x)` where the coefficients are the entries of the list l. The last element of the list `l` corresponds to the leading coefficients of `Ly`.

Examples

Example 1

We generate the linear ODE for `y(x)` with the coefficients `-1`, `4*x` and `4*x^2` of `y(x)`, `y'(x)` and `y''(x)`, respectively:

```
ode::mkODE([-1, 4*x, 4*x^2], y, x)
```

$$4x^2 \frac{\partial^2}{\partial x^2} y(x) - y(x) + 4x \frac{\partial}{\partial x} y(x)$$

Parameters

`l`

A `list` of coefficient functions.

y

The dependent variable of the resulting differential equation.

x

The independent variable of the resulting differential equation.

Return Values

Expression.

ode::normalize

Normalized form of a linear ordinary differential equation

Syntax

```
ode::normalize(Ly, y, x, n)
```

Description

`ode::normalize(Ly, y, x, n)` computes the normalized form of the `n`-th order linear ordinary differential equation `Ly`, i.e. whose leading coefficient (the coefficient of the highest derivative of `y(x)` in `Ly`) is 1.

Examples

Example 1

We normalize the following differential equation:

```
Ly:=-diff(y(x),x,x)/x+y(x)/4/x^3-diff(y(x),x)/x^2
```

$$\frac{y(x)}{4\,x^3} - \frac{\frac{\partial}{\partial x}\,y(x)}{x^2} - \frac{\frac{\partial^2}{\partial x^2}\,y(x)}{x}$$

```
ode::normalize(Ly, y, x, 2)
```

$$\frac{\frac{\partial}{\partial x}\,y(x)}{x} - \frac{y(x)}{4\,x^2} + \frac{\partial^2}{\partial x^2}\,y(x)$$

Parameters

Ly

A homogeneous linear ordinary differential equation.

y

The dependent variable of `Ly`.

x

The independent variable of `Ly`.

n

The order of `Ly`.

Return Values

Expression representing a linear differential equation.

ode::polynomialSolutions

Polynomial solutions of a homogeneous linear ordinary differential equation

Syntax

```
ode::polynomialSolutions(Ly, y(x), <Generic>)
```

Description

`ode::polynomialSolutions` computes a fundamental set of polynomial solutions of a homogeneous linear ordinary differential equation.

`ode::polynomialSolutions` returns a fundamental set of the polynomial solutions of `Ly`, i.e., solutions in the ring $\mathbb{Q}[x]$. When the option `Generic` is given, a generic form of them is returned.

Examples

Example 1

We compute the polynomial solutions of the following differential equation:

```
Ly:=3*x*diff(y(x),x,x)-x*diff(y(x),x)+9*y(x)
```

$$3\,x\,\frac{\partial^2}{\partial x^2}\,y(x) - x\,\frac{\partial}{\partial x}\,y(x) + 9\,y(x)$$

```
ode::polynomialSolutions(Ly, y(x))
```

$$\{x^9 - 216\,x^8 + 18144\,x^7 - 762048\,x^6 + 17146080\,x^5 - 205752960\,x^4 + 1234517760\,x^3 - 3174474240\,x^2 + 2380855680\,x\}$$

```
ode::polynomialSolutions(Ly, y(x), Generic)
```

$$C1\,(x^9 - 216\,x^8 + 18144\,x^7 - 762048\,x^6 + 17146080\,x^5 - 205752960\,x^4 + 1234517760\,x^3 - 3174474240\,x^2 + 2380855680\,x)$$

Parameters

`Ly`

A homogeneous linear ordinary differential equation with coefficients in the field $\mathbb{Q}(x)$ of rational functions over the rationals.

`y(x)`

The dependent function of `Ly`.

Return Values

`set`, possibly empty, of functions or an expression

See Also

MuPAD Functions

`ode::rationalSolutions`

ode::rationalSolutions

Rational solutions of a homogeneous linear ordinary differential equation

Syntax

```
ode::rationalSolutions(Ly, y(x), <Generic>)
```

Description

`ode::rationalSolutions` returns a fundamental set of the rational solutions of `Ly`, i.e., solutions in the field $\mathbb{Q}(x)$. When the option `Generic` is given, a generic form of them is returned.

Examples

Example 1

We compute the rational solutions of the following differential equation:

```
Ly:=(4*x^5+8*x^3+4*x)*diff(y(x),x,x)+
    (36*x^4+32*x^2-4)*diff(y(x),x)+48*x^3*y(x)
```

$$48\,x^3\,y(x)+(4\,x^5+8\,x^3+4\,x)\,\frac{\partial^2}{\partial x^2}\,y(x)+(36\,x^4+32\,x^2-4)\,\frac{\partial}{\partial x}\,y(x)$$

```
ode::rationalSolutions(Ly, y(x))
```

$$\left\{\frac{1}{x^6+3\,x^4+3\,x^2+1},\ \frac{x^4+2\,x^2}{x^6+3\,x^4+3\,x^2+1}\right\}$$

```
ode::rationalSolutions(Ly, y(x), Generic)
```

$$\frac{C1}{x^6+3x^4+3x^2+1}+\frac{C2\,(x^4+2x^2)}{x^6+3x^4+3x^2+1}$$

Parameters

`Ly`

A homogeneous linear ordinary differential equation with coefficients in the field $\mathbb{Q}(x)$ of rational functions over the rationals.

`y(x)`

The dependent function of `Ly`.

Return Values

`set`, possibly empty, of functions or an expression

See Also

MuPAD Functions

`ode::polynomialSolutions`

ode::ratSys

Rational solutions of a first order homogeneous linear differential system

Syntax

```
ode::ratSys(M, x)
```

Description

`ode::ratSys(M, x)` computes a fundamental set of rational solutions of the first order homogeneous linear differential system `Y'=MY`. This method uses a cyclic vector and therefore is not optimal.

Examples

Example 1

We compute the rational solutions of the following differential system:

```
A := matrix([ [2*(x+x^2-9)/x/(x-2),2*(x^2-6)/x/(x-2)],
     [-3*(2*x+x^2-12)/x/(x-2),-(2*x+3*x^2-24)/x/(x-2)] ])
```

$$\begin{pmatrix} \frac{2x^2+2x-18}{x(x-2)} & \frac{2x^2-12}{x(x-2)} \\ -\frac{3x^2+6x-36}{x(x-2)} & -\frac{3x^2+2x-24}{x(x-2)} \end{pmatrix}$$

```
v := ode::ratSys(A, x)
```

$$\left\{ \begin{pmatrix} \frac{x+2}{x^3} \\ -\frac{x+4}{x^3} \end{pmatrix} \right\}$$

And we can check the result:

```
diff(v[1], x) = normal(A*v[1])
```

$$\begin{pmatrix} -\frac{2\,(x+3)}{x^4} \\ \frac{2\,(x+6)}{x^4} \end{pmatrix} = \begin{pmatrix} -\frac{2\,(x+3)}{x^4} \\ \frac{2\,(x+6)}{x^4} \end{pmatrix}$$

Parameters

M

A square matrix of type `Dom::Matrix` with coefficients in the field $\mathbb{Q}(x)$ of rational functions over the rationals.

x

The independent function.

Return Values

`set`, possibly empty, of objects of type `Dom::Matrix`.

See Also

MuPAD Functions

`ode::cyclicVector` | `ode::rationalSolutions` | `ode::scalarEquation`

ode::scalarEquation

Transforms a linear differential system to an equivalent scalar linear differential equation

Syntax

```
ode::scalarEquation(A, x, y, <Transform>)
```

Description

`ode::scalarEquation` converts a first order homogeneous linear differential system to an equivalent homogeneous scalar linear differential equation using the method of cyclic vector.

`ode::scalarEquation(A, x, y)` returns a scalar homogeneous linear differential equation in `y(x)` equivalent to the first order homogeneous differential system $Y' = \mathrm{AY}$ using the method of the cyclic vector. If the option `Transform` is given then a list is returned whose first element is the corresponding differential equation `Ly` and second element is an invertible matrix `P` such that $C = P' P^{-1} + P A P^{-1}$ is the companion matrix associated to `Ly`; hence if `Z` is a solution of the differential system $Y' = \mathrm{AY}$ then PZ is a solution of the system $Y' = \mathrm{CY}$.

Examples

Example 1

We compute a linear differential equation equivalent to the following differential system:

```
A := matrix( [ [x^2-1,1,0], [0,x^2+5*x+1/3,1], [0,0,2]])
```

$$\begin{pmatrix} x^2-1 & 1 & 0 \\ 0 & x^2+5x+\frac{1}{3} & 1 \\ 0 & 0 & 2 \end{pmatrix}$$

```
l := ode::scalarEquation(A, x, y, Transform)
```

$$\left[\frac{\partial^3}{\partial x^3} y(x) - \left(2x^2 + 5x + \frac{4}{3}\right) \frac{\partial^2}{\partial x^2} y(x) + \left(x^4 + 5x^3 + \frac{10x^2}{3} - x - \frac{20}{3}\right) \frac{\partial}{\partial x} y(x)\right.$$

$$\left. - y(x) \left(2x^4 + 6x^3 - \frac{49x^2}{3} - \frac{38x}{3} + \frac{19}{3}\right), \begin{pmatrix} 1 & 0 & 0 \\ x^2 - 1 & 1 & 0 \\ 2x - 2x^2 + x^4 + 1 & 5x + 2x^2 - \frac{2}{3} & 1 \end{pmatrix}\right]$$

And we can check that, for `P=l[2]`, $P' P^{-1} + P A P^{-1}$ is the companion matrix associated to `l[1]`:

```
P := l[2]:
bool( diff(P,x)*P^(-1)+P*A*P^(-1) =
      ode::companionSystem(l[1], y(x)) )
```

TRUE

Parameters

A

A square matrix of type `Dom::Matrix`.

x

The independent variable of the resulting scalar differential equation.

y

The dependent variable of the resulting scalar differential equation.

Return Values

Expression or a list.

See Also

MuPAD Functions
`ode::cyclicVector`

ode::series

Series solutions of an ordinary differential equation

Syntax

```
ode::series(Ly, y(x), x | x = x0, <order>)
```

```
ode::series({Ly, <inits>}, y(x), x | x = x0, <order>)
```

Description

`ode::series(Ly, y(x), x = x0)` computes the first terms of the series expansions of the solutions of `Ly` with respect to the variable `x` around the point `x0`.

`ode::series` tries to compute either the Taylor series, the Laurent series or the Puiseux series of the solutions of the differential equation `Ly` around the point `x=x0`.

Suppose that `Ly` is a nonlinear differential equation. If `x0` is an ordinary point of `Ly` then a Taylor series is computed otherwise an expression of type `"series"` is returned. If initial conditions are given at the point `x0` then the answer is expressed in terms of the function `y(x)` and its derivatives evaluated at the point `x0`. See “Example 1” on page 21-37.

Suppose that `Ly` is a linear differential equation. If `x0` is an ordinary point of `Ly` then a Taylor series is computed, if `Ly` is furthermore homogeneous and `x0` is a regular point then a Puiseux series is computed (containing possible logarithmic terms), otherwise an expression of type `"series"` is returned. If initial conditions are given at the point `x0` then the answer is either expressed in terms of the function `y(x)` and its derivatives evaluated at the point `x0` or it may be expressed in terms of arbitrary constants.

Examples

Example 1

Consider the following nonlinear differential equation:

```
Ly := x^2*diff(y(x),x)+y(x)-x
```

$$y(x) - x + x^2 \frac{\partial}{\partial x} y(x)$$

We compute the series solutions at the point 0 which is a singular point:

```
ode::series(Ly, y(x), x=0)
```

$$\text{series}\left(y(x) - x + x^2 \frac{\partial}{\partial x} y(x),\ y(x),\ x = 0\right)$$

Then we compute the series solutions at the regular point 1:

```
ode::series(Ly, y(x), x=1)
```

$$\left\{y(1) - (x-1)\,(y(1)-1) + \left(\frac{3\,y(1)}{2} - 1\right)(x-1)^2 - \left(\frac{13\,y(1)}{6} - \frac{4}{3}\right)(x-1)^3 + \left(\frac{73\,y(1)}{24} - \frac{11}{6}\right)(x-1)^4 - \left(\frac{167\,y(1)}{40} - \frac{5}{2}\right)(x-1)^5 + O((x-1)^6)\right\}$$

And we can also put some initial conditions at the point 1:

```
ode::series({y(1)=1, Ly}, y(x), x=1)
```

$$\left\{1 + \frac{(x-1)^2}{2} - \frac{5\,(x-1)^3}{6} + \frac{29\,(x-1)^4}{24} - \frac{67\,(x-1)^5}{40} + O((x-1)^6)\right\}$$

Example 2

Consider the following linear differential equation:

```
Ly := (2*x+x^3)*diff(y(x),x$2)-diff(y(x),x)-6*x*y(x)
```

$$(x^3 + 2x)\frac{\partial^2}{\partial x^2} y(x) - \frac{\partial}{\partial x} y(x) - 6x\,y(x)$$

We compute the series solutions at the regular point 1:

```
ode::series(Ly, y(x), x=1)
```

$$\left\{y(1)+y'(1)\,(x-1)+(x-1)^2\left(\frac{y'(1)}{6}+y(1)\right)-\left(\frac{y(1)}{9}-\frac{7\,y'(1)}{27}\right)(x-1)^3 + \left(\frac{y(1)}{12}-\frac{y'(1)}{36}\right)(x-1)^4+\left(\frac{y(1)}{90}-\frac{2\,y'(1)}{135}\right)(x-1)^5+O((x-1)^6)\right\}$$

The series solutions at the regular singular point 0:

```
ode::series(Ly, y(x), x=0)
```

$$\left\{1+3\,x^2+\frac{3\,x^4}{5}+O(x^6),\ x^{3/2}+\frac{3\,x^{7/2}}{8}-\frac{3\,x^{11/2}}{128}+O\left(x^{15/2}\right)\right\}$$

An also the series solutions at the regular singular point *infinity*:

```
ode::series(Ly, y(x), x=infinity)
```

$$\left\{\frac{1}{x^2}-\frac{1}{x^4}+\frac{11}{9\,x^6}+O\left(\frac{1}{x^8}\right),\ 2880\,x^3+4320\,x-\frac{1080}{x}+O\left(\frac{1}{x^3}\right)\right\}$$

Example 3

Consider the following linear differential equation:

```
Ly := x^2*diff(y(x),x$2)-x*diff(y(x),x)+(1-x)*y(x)
```

$$x^2\,\frac{\partial^2}{\partial x^2}\,y(x)-y(x)\,(x-1)-x\,\frac{\partial}{\partial x}\,y(x)$$

We compute the series solutions at the regular singular point 0:

```
ode::series(Ly, y(x), x)
```

$$\left\{x\ln(x)+x^2(\ln(x)-2)+x^3\left(\frac{\ln(x)}{4}-\frac{3}{4}\right)+x^4\left(\frac{\ln(x)}{36}-\frac{11}{108}\right)+x^5\left(\frac{\ln(x)}{576}-\frac{25}{3456}\right)\right.$$
$$\left.+x^6\left(\frac{\ln(x)}{14400}-\frac{137}{432000}\right)+O(x^7),\ x+x^2+\frac{x^3}{4}+\frac{x^4}{36}+\frac{x^5}{576}+\frac{x^6}{14400}+O(x^7)\right\}$$

And at the same point we look for solutions satisfying the initial condition $y(0) = 1$ and $y(0) = 0$:

```
ode::series({y(0)=1, Ly}, y(x), x)
```

$$\varnothing$$

```
ode::series({y(0)=0, Ly}, y(x), x)
```

$$\left\{C4\,x\ln(x)+C4\,x^2(\ln(x)-2)+C4\,x^3\left(\frac{\ln(x)}{4}-\frac{3}{4}\right)+C4\,x^4\left(\frac{\ln(x)}{36}-\frac{11}{108}\right)\right.$$
$$+C4\,x^5\left(\frac{\ln(x)}{576}-\frac{25}{3456}\right)+C4\,x^6\left(\frac{\ln(x)}{14400}-\frac{137}{432000}\right)+O(x^7),$$
$$\left.C3\,x+C3\,x^2+\frac{C3\,x^3}{4}+\frac{C3\,x^4}{36}+\frac{C3\,x^5}{576}+\frac{C3\,x^6}{14400}+O(x^7)\right\}$$

Parameters

Ly

An ordinary differential equation.

y(x)

The dependent function of `Ly`.

x

The independent variable of `Ly`.

`x0`

The expansion point: an arithmetical expression. If not specified, the default expansion point 0 is used .

`inits`

The initial or boundary conditions: a sequence of equations.

`order`

The number of terms to be computed: a nonnegative integer. The default order is given by the environment variable `ORDER` (default value 6).

Return Values

Either a `list`, maybe empty, of objects of type `Series::Puiseux` or an expression of type `"series"`.

ode::solve

Solving ordinary differential equations

Syntax

```
ode::solve(o, options)
```

```
solve(o, options)
```

Description

`ode::solve` computes solutions for ordinary differential equations.

`ode::solve(o)` returns the set of solutions of the ordinary differential equation `o`. You can also call the generic function `solve(o)`.

The solver detects the type of the differential equation and chooses an algorithm according to the detected equation type. If you know the type of the equation, you can use the option `Type` = OdeType to pass the equation type to the solver. Passing the equation type to the solver increases performance.

The solver recognizes the following values of `OdeType`:

- `Abel` - Abel differential equation
- `Bernoulli` - Bernoulli differential equation
- `Chini` - Chini differential equation
- `Clairaut` - Clairaut differential equation
- `ExactFirstOrder` - exact first order ordinary differential equation
- `ExactSecondOrder` - exact second order ordinary differential equation
- `Homogeneous` - homogeneous first order ordinary differential equation
- `Lagrange` - Lagrange differential equation
- `Riccati` - Riccati differential equation

See the Background section for more details on the classes of ordinary differential equations.

If the solver cannot identify the equation with the type you indicated, it issues a warning and returns the special value `FAIL`.

To solve an ordinary differential equation disregarding possible conditions on the parameters of the equation, use `IgnoreSpecialCases` option. This option eliminates receiving a set of special cases as an answer.

To solve an ordinary differential equation in a simplified manner, use the `IgnoreAnalyticConstraints` option. This option can provide simple solutions for the equations for which the direct use of the solver gives complicated results. If you use the `IgnoreAnalyticConstraints` option, always check the answer. This option can lead to wrong or incomplete results. See "Example 3" on page 21-45.

The solutions of ordinary differential equations can contain arbitrary constants of integration. The solver generates the constants of integration using the format of an uppercase letter `C` followed by an automatically generated number, for example `C13`.

The solver does not always verify the uniqueness and completeness of the returned solution. For example:

- The solver does not validate the Lipschitz-conditions on the ordinary differential equation for the Picard-Lindelöf Theorem.
- For some complex nonlinear systems of differential equations the solver returns constant solutions and does not warn you that other solutions exist.

The solver might ignore assumptions that you set on symbolic parameters and variables or use them only partially. More precisely, `ode::solve` passes assumptions to the functions that it calls internally. While these functions can use the specified assumptions, `ode::solve` itself does not use them in most of its internal algorithms. The same happens if you define an ordinary differential equation using `ode` and solve it using `solve`.

Examples

Example 1

To define an ordinary differential equation, use the `ode` command:

```
o:= ode(y'(x) = y(x)^2, y(x))
```

$$\text{ode}(y'(x) - y(x)^2,\ y(x))$$

To solve the equation, enter:

```
ode::solve(o)
```

$$\left\{0,\ -\frac{1}{C3 + x}\right\}$$

or more efficiently:

```
solve(o)
```

$$\left\{0,\ -\frac{1}{C7 + x}\right\}$$

Internally, the function `ode::solve` calls the function `solve`.

Example 2

You can solve an ordinary differential equation with a symbolic parameter and an initial condition:

```
o:= ode({y'(x) = a*y(x)^2, y(a) = ln(a)}, y(x)):
solve(o)
```

$$\begin{cases} \{0\} & \text{if } a = 1 \\ \left\{\dfrac{1}{\frac{1}{\ln(a)} - a\,x + a^2}\right\} & \text{if } a \neq 1 \end{cases}$$

To reduce the number of returned solutions, use the option `IgnoreSpecialCases`. For example, you can drop the solution for the parameter $a = 1$:

```
solve(o, IgnoreSpecialCases)
```

$$\left\{\frac{1}{\frac{1}{\ln(a)} - a\,x + a^2}\right\}$$

With the `IgnoreSpecialCases` option, a returned set of solutions can be incomplete.

Example 3

The solver can return `piecewise` solutions:

```
o:= ode(y'(x) = a/y(x)^2 + b*y(x), y(x)):
solve(o)
```

$$\begin{cases} \{C18\} & \text{if } a = 0 \wedge b = 0 \\ \left\{ \frac{\sigma_1}{b^{1/3}}, \sigma_3, \sigma_4, \frac{\sigma_1 \left(-\frac{1}{2}+\sigma_5\right)}{b^{1/3}}, \sigma_2, -\frac{\sigma_1 \left(\frac{1}{2}+\sigma_5\right)}{b^{1/3}} \right\} & \text{if } a \neq 0 \wedge b \neq 0 \\ \varnothing & \text{if } (a = 0 \vee b = 0) \wedge (a = 0 \vee b \neq 0) \wedge (a \neq 0 \vee b = 0) \wedge (a \neq 0 \vee b \neq 0) \\ \{\sigma_3, \sigma_4, \sigma_2\} & \text{if } ((a = 0 \wedge b \neq 0) \vee (a \neq 0 \wedge b = 0)) \wedge (a = 0 \vee b = 0) \wedge (a \neq 0 \vee b \neq 0) \end{cases}$$

where

$$\sigma_1 = (-a)^{1/3}$$

$$\sigma_2 = -\left(\frac{1}{2}+\sigma_5\right)\sigma_4$$

$$\sigma_3 = \left(-\frac{1}{2}+\sigma_5\right)\sigma_4$$

$$\sigma_4 = \left(-\frac{a - e^{3\,b\,(C19+x)}}{b}\right)^{1/3}$$

$$\sigma_5 = \frac{\sqrt{3}\,i}{2}$$

This solution is complete and mathematically correct for all possible values of the parameter a and variable x. Also you can try the option `IgnoreAnalyticConstraints` to obtain a particular solution that is correct under a set of common assumptions:

```
solve(o, IgnoreAnalyticConstraints)
```

$$\left\{ \frac{\sigma_2}{b^{1/3}}, \left(-\frac{1}{2}+\frac{\sqrt{3}\,i}{2}\right)\sigma_1, \sigma_1, \frac{\sigma_2\left(-\frac{1}{2}+\frac{\sqrt{3}\,i}{2}\right)}{b^{1/3}}, -\left(\frac{1}{2}+\frac{\sqrt{3}\,i}{2}\right)\sigma_1, -\frac{\sigma_2\left(\frac{1}{2}+\frac{\sqrt{3}\,i}{2}\right)}{b^{1/3}} \right\}$$

where

$$\sigma_1 = \left(-\frac{a-e^{3\,b\,(C24+x)}}{b}\right)^{1/3}$$

$$\sigma_2 = (-a)^{1/3}$$

The solver accepts several options:

```
solve(o, Type = Bernoulli, IgnoreAnalyticConstraints)
```

$$\left\{ \left(-\frac{1}{2}+\frac{\sqrt{3}\,i}{2}\right)\sigma_1, \sigma_1, -\left(\frac{1}{2}+\frac{\sqrt{3}\,i}{2}\right)\sigma_1 \right\}$$

where

$$\sigma_1 = \left(-\frac{a-C26\,b\,e^{3\,b\,x}}{b}\right)^{1/3}$$

Example 4

Suppose, you want to solve an ordinary differential equation from the class of Bernoulli equations:

```
o:= ode(y'(x) = (- 1/x + 2*I)*y(x) + 1/x*y(x)^2, y(x)):
solve(o)
```

$$\left\{0,\ \frac{e^{-\ln(x)+2\,x\,i}}{C28+2\,Ei(1,\ -2\,x\,i)\,i+\frac{e^{2\,x\,i}}{x}}\right\}$$

The solver recognizes the type of the equation and uses the algorithm for solving Bernoulli equations. To improve performance, you can explicitly pass the type of the equation to the solver:

```
solve(o, Type = Bernoulli)
```

$$\left\{0,\ \frac{x\,e^{-\ln(x)+2\,x\,i}}{e^{2\,x\,i}+C29\,x+2\,x\,Ei(1,\ -2\,x\,i)\,i}\right\}$$

To solve the Clairaut equation with the initial conditions, enter:

```
o:= ode({(x*y'(x)-y(x))^2 - y'(x)^2 - 1 = 0, y(1) = 1}, y(x)):
solve(o, Type = Clairaut)
```

$$\{1\}$$

If the solver cannot identify the equation with the type you indicated, it issues a warning and returns the special value `FAIL`:

```
o:= ode({(x*y'(x)-y(x))^2 - y'(x)^2 - 1 = 0, y(1) = 1}, y(x)):
solve(o, Type = Lagrange)
```

```
Warning: Cannot detect the Lagrange ODE. [ode::lagrange]
```

$$\text{FAIL}$$

Example 5

Some ordinary differential equations belong to several classes. For example, some Chini equations are also homogeneous and some Lagrange equations are also Clairaut equations. If an equation belongs to several classes simultaneously, the solver can present its solution in different forms. The form of a solution depends on the class with

which an equation is identified. For example, suppose you want to solve the Chini differential equation. You can explicitly pass the type of the equation to the solver:

```
o:= ode(y'(x) = 1/x*y(x)^2 + 1/x*y(x) + x, y(x)):
L:= solve(o, Type = Chini)
```

$$\{x \tan(C31 + x),\ -x\,i,\ x\,i\}$$

You also can let the solver recognize the type of the equation:

```
solve(o)
```

$$\left\{-x\,i,\ -x\,i + \frac{2\,e^{2\,x\,i}\,e^{\ln(x)-2\,x\,i}}{2\,C33\,e^{2\,x\,i} - i}\right\}$$

The solver does not return the type with which an ordinary differential equation is internally identified. If you want to verify that both solution sets are equivalent, use the `rewrite` function with target `exp` on the first set of solutions:

```
rewrite(L, exp)
```

$$\left\{-x\,i,\ x\,i,\ -\frac{x\left(e^{2\,C31\,i+2\,x\,i}\,i - i\right)}{e^{2\,C31\,i+2\,x\,i} + 1}\right\}$$

Example 6

MuPAD solves some classes of Riccati ordinary differential equations that involve arbitrary functions. For example, the following equation contains the arbitrary function `f(x)`:

```
eq := diff(y(x), x) - f(x)*y(x)^2 + a^2*x^(2*n)*f(x) - a*n*x^(n - 1)
```

$$\frac{\partial}{\partial x}\,y(x) - f(x)\,y(x)^2 + a^2\,x^{2\,n}\,f(x) - a\,n\,x^{n-1}$$

For this equation the solver returns:

```
solve(ode(eq,y(x)))
```

$$\left\{ a\,x^n,\ a\,x^n + \frac{e^{\int 2\,a\,x^n\,f(x)\,dx}}{C35 - \int e^{\int 2\,a\,x^n\,f(x)\,dx}\,f(x)\,dx} \right\}$$

You also can solve an equation with more than one arbitrary function. For example, the following equations contain `f(x)` and `g(x)`:

```
eq:= diff(y(x), x) - g(x)*f(x)*y(x) - g(x) - diff(f(x), x)*y(x)^2
```

$$-y(x)^2\,\frac{\partial}{\partial x}f(x) + \frac{\partial}{\partial x}y(x) - f(x)\,g(x)\,y(x) - g(x)$$

The returned solution is:

```
solve(ode(eq,y(x)))
```

$$\left\{ \frac{e^{\int f(x)\,g(x)\,dx - 2\ln(f(x))}}{C37 - \int e^{\int f(x)\,g(x)\,dx - 2\ln(f(x))}\,\frac{\partial}{\partial x}f(x)\,dx} - \frac{1}{f(x)},\ -\frac{1}{f(x)} \right\}$$

Example 7

Suppose, you want to solve the following second-order ordinary differential equation:

```
eq := x^2*(x^2 + 1)*diff(y(x),x,x) +
      x*(2*x^2 + 1)*diff(y(x),x) -
     (nu*(nu + 1)*x^2 + n^2)*y(x)
```

$$x\,(2\,x^2+1)\,\frac{\partial}{\partial x}y(x) - y(x)\,(n^2 + \mathrm{nu}\,(\mathrm{nu}+1)\,x^2) + x^2\,(x^2+1)\,\frac{\partial^2}{\partial x^2}y(x)$$

The solver returns the result in terms of the hypergeometric function ${}_1F_2$ (see `hypergeom`):

```
solve(ode(eq,y(x)))
```

$$\left\{ \frac{C39\ {}_2F_1\left(\frac{nu}{2}-\frac{n}{2}+\frac{1}{2}, \frac{n}{2}+\frac{nu}{2}+\frac{1}{2}; nu+\frac{3}{2}; \sigma_1\right)}{x^{nu+1}} + C40\ x^{nu}\ {}_2F_1\left(-\frac{n}{2}-\frac{nu}{2}, \frac{n}{2}-\frac{nu}{2}; \frac{1}{2}-nu; \sigma_1\right) \right\}$$

where

$$\sigma_1 = -\frac{1}{x^2}$$

Example 8

The solver can handle some third- and higher-order ordinary differential equations. For example, solve the following third-order linear differential equations:

```
eq := ode(sin(x)*y'''(x) + cos(x)*y(x), y(x)):
solve(eq)
```

$$\{C42\ \sin(x)\}$$

```
eq := ode(6*y(x) + x^3*y'''(x), y(x)):
solve(eq)
```

$$\left\{ C47\ \sigma_1 - \frac{\sigma_1\left(\frac{C46}{22}+\frac{3\sqrt{2}\ C46\ i}{44}\right)}{x^{3+\sqrt{2}\ i}} + \frac{\sqrt{2}\ C48\ \sigma_2\ \sigma_1\ i}{4} + \frac{\sqrt{2}\ \sigma_2\ \sigma_1\left(\frac{3\ C46}{11}+\frac{\sqrt{2}\ C46\ i}{11}\right) i}{4\ x^{3-\sqrt{2}\ i}} \right\}$$

where

$$\sigma_1 = x^{2+\sqrt{2}\ i}$$

$$\sigma_2 = \frac{1}{x^{2\sqrt{2}\ i}}$$

Example 9

The solver also can handle some nonlinear first-order ordinary differential equations. For example, solve the following first-order linear differential equations:

```
eq := ode(y(x)*diff(y(x), x) - y(x) - x^3 - 4*x^4 - 4*x^7, y(x)):
solve(eq)
```

$$\left\{x^4 + x + \frac{1}{4}\right\}$$

```
eq := ode(exp(x/2)/4 - 2*exp(x) - y(x) + x*exp(x/2) + y(x)*y'(x), y(x)):
solve(eq)
```

$$\left\{x - 2\,e^{\frac{x}{2}} + \frac{1}{4}\right\}$$

Parameters

o

An ordinary differential equation, an object of the type `ode`.

Options

Type

Option, specified as `Type = OdeType`

Indicates the type of the ordinary differential equation and accepts the following arguments: `Abel`, `Bernoulli`, `Chini`, `Clairaut`, `ExactFirstOrder`, `ExactSecondOrder`, `Homogeneous`, `Lagrange`, `Riccati`.

MaxDegree

Option, specified as `MaxDegree = n`

Pass the option to the generic solver, which is called internally for all intermediate equations. See the list of options for the `solve` function for further information.

IgnoreSpecialCases

Pass the option to the generic solver, which is called internally for all intermediate equations, and to the integrator `int`, which is called for computing all intermediate integrals. See the list of options for the `solve` function for further information.

`IgnoreAnalyticConstraints`

Pass the option to the generic solver, which is called internally for all intermediate equations, and to the integrator `int`, which is called for computing all intermediate integrals. See the list of options for the `solve` function for further information.

Return Values

Set of solutions of the ordinary differential equation or the special value `FAIL`. For additional information on the return values, see the `solve` help page.

References

For more information on the particular classes of ordinary differential equations see:

- E. Kamke: Differentialgleichungen: Lösungmethoden und Lösungen. B.G. Teubner, Stuttgart, 1997
- G.M. Murphy: Ordinary differential equations and their solutions. Van Nostrand, Princeton, 1960
- Andrei D. Polyanin and Valentin F. Zaitsev: Handbook of exact solutions for ordinary differential equations, second ed., Chapman & Hall/CRC, Boca Raton, FL, 2003
- D. Zwillinger: Handbook of differential equations. San Diego: Academic Press, 1992

More About

- "Solve Ordinary Differential Equations and Systems"

ode::symmetricPower

Symmetric power of a homogeneous linear ordinary differential equation

Syntax

```
ode::symmetricPower(Ly, y(x), m)
```

Description

`ode::symmetricPower(Ly, y(x), m)` computes the m-th symmetric power of `Ly`. This is the lowest order linear ordinary differential equation whose solution space consists exactly of all possible m-th powerproducts of solutions of `Ly`.

Examples

Example 1

We compute the second symmetric power of the following differential equation:

```
Ly:=x^2*diff(y(x),x$2)-(36*x^6*exp(4*x^3)+9*x^6+2)*y(x)
```

$$x^2 \frac{\partial^2}{\partial x^2} y(x) - y(x) \left(36\, x^6\, e^{4\, x^3} + 9\, x^6 + 2\right)$$

```
ode::symmetricPower(Ly, y(x), 2)
```

$$\frac{\partial^3}{\partial x^3} y(x) - \left(144\, x^4\, e^{4\, x^3} + \frac{8}{x^2} + 36\, x^4\right) \frac{\partial}{\partial x} y(x)$$

$$- y(x) \left(288\, x^3\, e^{4\, x^3} + 864\, x^6\, e^{4\, x^3} - \frac{8}{x^3} + 72\, x^3\right)$$

Parameters

`Ly`

A homogeneous linear ordinary differential equation.

`y(x)`

The dependent function of `Ly`.

`m`

A positive integer.

Return Values

Linear differential equation

ode::unimodular

Unimodular transformation of a linear ordinary differential equation

Syntax

```
ode::unimodular(Ly, y(x), <Transform>)
```

Description

`ode::unimodular(Ly, y(x))` tests if the linear homogeneous differential equation `Ly` has a unimodular Galois group (i.e. the wronskian lies in the base field $\mathbb{Q}(x)$), if not transforms `Ly` into a unimodular one (by changing the second highest coefficient to zero) and returns a `table` with index `equation` and `factorOfTransformation` containing respectively the transformed differential equation and the factor of transformation `Wn` such that a solution of the transformed equation multiplied by `Wn` is a solution of `Ly`.

If the option `Transform` is given then `Ly` is transformed unconditionally even if `Ly` has yet a unimodular Galois group.

Examples

Example 1

We test if the following differential equation has a unimodular Galois group:

```
Ly := y(x)*6+x*diff(y(x),x)*(-2)+diff(y(x),x$2)*(-x^2+1)
```

$$-(x^2-1)\,\frac{\partial^2}{\partial x^2}\,y(x)-2\,x\,\frac{\partial}{\partial x}\,y(x)+6\,y(x)$$

```
ode::unimodular(Ly, y(x))
```

equation	$6\,y(x) - 2\,x\,\frac{\partial}{\partial x}\,y(x) - (x^2-1)\,\frac{\partial^2}{\partial x^2}\,y(x)$
factorOfTransformation	1

It is unimodular since the factor of transformation is 1. We can also check this by computing the wronskian of Ly which is a rational function:

```
ode::wronskian(Ly,y(x))
```

$$\frac{1}{x^2-1}$$

Now we transform Ly into a differential equation whose wronskian is 1:

```
ode::unimodular(Ly, y(x), Transform)
```

equation	$\frac{\partial^2}{\partial x^2}\,y(x) - \frac{y(x)\left(6\,x^2-7\right)}{-2\,x^2+x^4+1}$
factorOfTransformation	$\sqrt{\frac{1}{x^2-1}}$

```
ode::wronskian(%[equation], y(x))
```

$$1$$

Parameters

Ly

A homogeneous linear differential equation over $\mathbb{Q}(x)$.

y(x)

The dependent function of Ly.

Return Values

`table`.

See Also

MuPAD Functions
`ode::wronskian`

ode::vectorize

Coefficients of a homogeneous linear ODE

Syntax

```
ode::vectorize(Ly, y, x, n)
```

Description

`ode::vectorize(Ly, y, x, n)` returns the list of coefficients of the `n`-th order homogeneous linear ordinary differential equation `Ly`.

Examples

Example 1

We compute the list of coefficients of the following differential equation:

```
Ly := 4*x^2*diff(y(x), x$3)+exp(x^2)*diff(y(x),x$2)+
      4*x*diff(y(x),x)-y(x)
```

$$4\,x\,\frac{\partial}{\partial x}\,y(x)-y(x)+e^{x^2}\,\frac{\partial^2}{\partial x^2}\,y(x)+4\,x^2\,\frac{\partial^3}{\partial x^3}\,y(x)$$

```
ode::vectorize(Ly, y, x, 3)
```

$$\left[-1,\,4\,x,\,e^{x^2},\,4\,x^2\right]$$

Parameters

Ly

A homogeneous linear ordinary differential equation.

y

The dependent variable of `Ly`.

x

The independent variable of `Ly`.

n

The order of `Ly`, a positive integer.

Return Values

`list`

ode::wronskian

Wronskian of functions or of a linear homogeneous ordinary differential equation

Syntax

```
ode::wronskian(l, x, <R>)
```

```
ode::wronskian(Ly, y(x), <R>)
```

Description

`ode::wronskian` computes the wronskian (determinant) of functions or of a linear homogeneous ordinary differential equation.

`ode::wronskian(l, x)` returns the wronskian, i.e. the determinant of the wronskian matrix, of the elements of `l` with respect to `x`.

`ode::wronskian(Ly, y(x))` returns the wronskian of `Ly` defined as the element `w` such that $w' = -a_{n-1} w$, where a_{n-1} is the coefficient of `Ly` of degree n - 1 and n the order of `Ly`.

If the optional argument `R` is given, then the specified differential ring will be chosen for representing the entries of the wronskian matrix.

Examples

Example 1

We compute the wronskian of `[2*x^2+1, x*sqrt(1+x^2), y(x)]` which is a linear differential equation in `y(x)`:

```
Ly:=ode::wronskian([2*x^2+1, x*sqrt(1+x^2), y(x)], x)
```

$$4\,x\,\sigma_4\,\frac{\partial}{\partial x}\,y(x) - 4\,x^2\,\sigma_4\,\frac{\partial^2}{\partial x^2}\,y(x) - 4\,y(x)\,\sigma_3 - \sigma_1\,\sigma_2\,\frac{\partial}{\partial x}\,y(x) + 4\,x\,y(x)\,\sigma_2 + \sigma_1\,\sigma_3\,\frac{\partial^2}{\partial x^2}\,y(x)$$

where

$$\sigma_1 = 2\,x^2 + 1$$

$$\sigma_2 = \frac{3\,x}{\sigma_4} - \frac{x^3}{\left(x^2+1\right)^{3/2}}$$

$$\sigma_3 = \sigma_4 + \frac{x^2}{\sigma_4}$$

$$\sigma_4 = \sqrt{x^2+1}$$

```
Ly := numer( normal(Ly) )
```

$$x^2\,\frac{\partial^2}{\partial x^2}\,y(x) - 4\,y(x) + x\,\frac{\partial}{\partial x}\,y(x) + \frac{\partial^2}{\partial x^2}\,y(x)$$

And we can check that a basis of solutions of `Ly` is as expected:

```
ode::solve(Ly, y(x))
```

$$\left\{ C2\left(x+\sqrt{x^2+1}\right)^2 + \frac{C3}{\left(x+\sqrt{x^2+1}\right)^2} \right\}$$

We can also compute the wronskian of `Ly`, which is, up to a constant, the wronskian of `x^2+1` and `x*sqrt(x^2+1)`:

```
ode::wronskian(Ly, y(x)),
simplify(ode::wronskian([x^2+1/2,x*sqrt(1+x^2)], x))
```

$$\frac{1}{\sqrt{x^2+1}},\ \frac{1}{2\sqrt{x^2+1}}$$

Parameters

`l`

A list of functions of the variable `x`.

`Ly`

A homogeneous linear ordinary differential equation.

`y(x)`

The dependent function of `Ly`.

`R`

A differential ring, default is `Dom::ExpressionField(id, iszero@normal)`.

Return Values

Expression in `x`.

orthpoly – Orthogonal Polynomials

orthpoly::chebyshev1
orthpoly::chebyshev2
orthpoly::curtz
orthpoly::gegenbauer
orthpoly::hermite
orthpoly::jacobi
orthpoly::laguerre
orthpoly::legendre

orthpoly::chebyshev1

The Chebyshev polynomials of the first kind

Syntax

```
orthpoly::chebyshev1(n, x)
```

Description

`orthpoly::chebyshev1(n,x)` computes the value of the n-th degree Chebyshev polynomial of the first kind at the point x.

These polynomials have integer coefficients.

Evaluation is fast and numerically stable for real floating point values x from the interval [- 1.0, 1.0]. See “Example 2” on page 22-3.

`orthpoly::chebyshev2` implements the Chebyshev polynomials of the second kind.

Examples

Example 1

Polynomial expressions are returned if identifiers or indexed identifiers are specified:

```
orthpoly::chebyshev1(2, x)
```

$$2\,x^2 - 1$$

```
orthpoly::chebyshev1(3, x[1])
```

$$4\,x_1^{\,3} - 3\,x_1$$

Using arithmetical expressions as input, the “values” of these polynomials are returned:

```
orthpoly::chebyshev1(2, 3 + 2*I)
```

$$9 + 24\,i$$

```
orthpoly::chebyshev1(3, exp(x[1]+2))
```

$$4\,e^{6}\,e^{3\,x_1} - 3\,e^{x_1}\,e^{2}$$

"Arithmetical expressions" include numbers:

```
orthpoly::chebyshev1(2, sqrt(2)),
orthpoly::chebyshev1(3, 8 + I),
orthpoly::chebyshev1(1000, 0.3)
```

$$3,\ 1928 + 761\,i,\ -0.9991251116$$

If the degree of the polynomial is a variable or expression, then `orthpoly::chebyshev1` returns itself symbolically:

```
orthpoly::chebyshev1(n, x)
```

$$\text{orthpoly::chebyshev1}(n, x)$$

Example 2

If a floating-point value is desired, then a direct call such as

```
orthpoly::chebyshev1(200, 0.3)
```

$$-0.3169632681$$

is appropriate and yields a correct result. One should not evaluate the symbolic polynomial at a floating-point value, because this may be numerically unstable:

```
T200 := orthpoly::chebyshev1(200, x):

DIGITS := 10: evalp(T200, x = 0.3)
```

```
                                  - 3912167.235
```

This result is caused by numerical round-off. Also with increased `DIGITS` only a few leading digits are correct:

```
DIGITS := 20: evalp(T200, x = 0.3)
```

```
                                  - 0.31710101442512309682
```

```
delete DIGITS, T200:
```

Parameters

`n`

A nonnegative integer or an arithmetical expression representing a nonnegative integer: the degree of the polynomial.

`x`

An indeterminate or an arithmetical expression. An indeterminate is either an identifier (of domain type `DOM_IDENT`) or an indexed identifier (of type `"_index"`).

Return Values

The value of the Chebyshev polynomial at point `x` is returned as an arithmetical expression. If `n` is an arithmetical expression, then `orthpoly::chebyshev1` returns itself symbolically.

Algorithms

The Chebyshev polynomials are given by $T(n, x) = cos(n\ acos(x))$ for real $x \in [-1, 1]$. This representation is used by `orthpoly::chebyshev1` for floating-point values in this range.

These polynomials satisfy the recursion formula

$$T(n, x) = 2\,x\,T(n-1, x) - T(n-2, x)$$

with $T(0, x) = 1$ and $T(1, x) = x$.

They are orthogonal on the interval [- 1, 1] with respect to the weight function $w(x) = \frac{1}{\sqrt{(1-x^2)}}$.

$T(n, x)$ is a special Jacobi polynomial:

$$T(n, x) = \frac{2^{2n}\,n!^2}{(2\,n)!}\,P\left(n, -\frac{1}{2}, -\frac{1}{2}, x\right)$$

See Also

MuPAD Functions

`orthpoly::chebyshev2` | `orthpoly::jacobi`

orthpoly::chebyshev2

The Chebyshev polynomials of the second kind

Syntax

```
orthpoly::chebyshev2(n, x)
```

Description

`orthpoly::chebyshev2(n,x)` computes the value of the n-th degree Chebyshev polynomial of the second kind at the point x.

These polynomials have integer coefficients.

Evaluation is fast and numerically stable for real floating point values x from the interval [- 1.0, 1.0]. See “Example 2” on page 22-7.

`orthpoly::chebyshev1` implements the Chebyshev polynomials of the first kind.

Examples

Example 1

Polynomial expressions are returned if identifiers or indexed identifiers are specified:

```
orthpoly::chebyshev2(2, x)
```

$$4x^2 - 1$$

```
orthpoly::chebyshev2(3, x[1])
```

$$8x_1^3 - 4x_1$$

Using arithmetical expressions as input, the “values” of these polynomials are returned:

```
orthpoly::chebyshev2(2, 3 + 2*I)
```

$$19 + 48\,i$$

```
orthpoly::chebyshev2(3, exp(x[1] + 2))
```

$$8\,e^{6}\,e^{3\,x_1} - 4\,e^{x_1}\,e^{2}$$

"Arithmetical expressions" include numbers:

```
orthpoly::chebyshev2(2, sqrt(2)),
orthpoly::chebyshev2(3, 8 + I),
orthpoly::chebyshev2(1000, 0.3)
```

$$7,\ 3872 + 1524\,i,\ -1.012277265$$

If the degree of the polynomial is a variable or expression, then `orthpoly::chebyshev2` returns itself symbolically:

```
orthpoly::chebyshev2(n, x)
```

$$\text{orthpoly::chebyshev2}(n, x)$$

Example 2

If a floating-point value is desired, then a direct call such as

```
orthpoly::chebyshev2(200, 0.3)
```

$$-0.01869337443$$

is appropriate and yields a correct result. One should not evaluate the symbolic polynomial at a floating-point value, because this may be numerically unstable:

```
U200 := orthpoly::chebyshev2(200, x):

DIGITS := 10: evalp(U200, x = 0.3)
```

```
                    - 3872355.739
```

This result is caused by numerical round-off. Also with increased `DIGITS` only a few leading digits are correct:

```
DIGITS := 20: evalp(U200, x = 0.3)
```

```
                  - 0.018233184138451814741
```

```
delete DIGITS, U200:
```

Parameters

`n`

A nonnegative integer or an arithmetical expression representing a nonnegative integer: the degree of the polynomial.

`x`

An indeterminate or an arithmetical expression. An indeterminate is either an identifier (of domain type `DOM_IDENT`) or an indexed identifier (of type `"_index"`).

Return Values

The value of the Chebyshev polynomial at point `x` is returned as an arithmetical expression. If `n` is an arithmetical expression, then `orthpoly::chebyshev2` returns itself symbolically.

Algorithms

The Chebyshev polynomials of the second kind are given by

$$U(n, x) = \frac{\sin((n+1)\ \mathrm{acos}(x))}{\sin(\mathrm{acos}(x))}$$

for real $x \in [-1, 1]$. This representation is used by `orthpoly::chebyshev2` for floating-point values in this range.

These polynomials satisfy the recursion formula

$$U(n, x) = 2\,x\,U(n-1, x) - U(n-2, x)$$

with $U(0, x) = 1$ and $U(1, x) = 2\,x$.

They are orthogonal on the interval $[-1, 1]$ with respect to the weight function $w(x) = \sqrt{1-x^2}$.

$U(n, x)$ coincides with the Gegenbauer polynomial $G(n, 1, x)$.

$U(n, x)$ is a special Jacobi polynomial:

$$U(n, x) = \frac{2^{2n}\, n!\,(n+1)!}{(2n+1)!}\, P\left(n, \frac{1}{2}, \frac{1}{2}, x\right)$$

See Also

MuPAD Functions

`orthpoly::chebyshev1` | `orthpoly::gegenbauer` | `orthpoly::jacobi`

orthpoly::curtz

The Curtz polynomials

Syntax

```
orthpoly::curtz(n, x)
```

Description

`orthpoly::curtz(n,x)` computes the value of the n-th degree Curtz polynomial at the point x.

These polynomials have rational coefficients.

Evaluation for real floating-point values x is numerically stable. See "Example 2" on page 22-11.

Examples

Example 1

Polynomial expressions are returned if identifiers or indexed identifiers are specified:

```
orthpoly::curtz(2, x)
```

$$x^2 - x + \frac{1}{3}$$

```
orthpoly::curtz(3, x[1])
```

$$x_1^3 - \frac{3\,x_1^2}{2} + \frac{11\,x_1}{12} - \frac{1}{4}$$

Using arithmetical expressions as input, the "values" of these polynomials are returned:

```
orthpoly::curtz(2, 3+2*I)
```

$$\frac{7}{3} + 10\,i$$

```
orthpoly::curtz(3, exp(x[1] + 2))
```

$$\frac{11\,e^{x_1}\,e^{2}}{12} - \frac{3\,e^{4}\,e^{2\,x_1}}{2} + e^{6}\,e^{3\,x_1} - \frac{1}{4}$$

"Arithmetical expressions" include numbers:

```
orthpoly::curtz(2, sqrt(2)), orthpoly::curtz(3, 8 + I),
orthpoly::curtz(100, 0.3)
```

$$\frac{7}{3} - \sqrt{2},\ \frac{4807}{12} + \frac{2015\,i}{12},\ 0.001395122936$$

If the degree of the polynomial is a variable or expression, then `orthpoly::curtz` returns itself symbolically:

```
orthpoly::curtz(n, x)
```

$$\text{orthpoly::curtz}(n, x)$$

Example 2

If a floating-point value is desired, then a direct call such as

```
orthpoly::curtz(50, 1.2)
```

$$0.0003843630923$$

is appropriate and yields a correct result. One should not evaluate the symbolic polynomial at a floating-point value, because this may be numerically unstable:

```
orthpoly::curtz(50, x): evalp(%, x = 1.2)
```

$$0.0003849036173$$

Note that only 3 digits are correct due to numerical round-off.

Parameters

n

A nonnegative integer or an arithmetical expression representing a nonnegative integer: the degree of the polynomial.

x

An indeterminate or an arithmetical expression. An indeterminate is either an identifier (of domain type `DOM_IDENT`) or an indexed identifier (of type `"_index"`).

Return Values

The value of the Curtz polynomial at point `x` is returned as an arithmetical expression. If `n` is an arithmetical expression, then `orthpoly::curtz` returns itself symbolically.

Algorithms

The Curtz polynomials are given by the recursion formula

$$C(n, x) = x^n + x \left(\sum_{i=1}^{n-1} \frac{(-1)^i}{i+1} C(n-i-1, x) \right) + \frac{(-1)^n}{n+1}$$

with $C(0, x) = 1$.

orthpoly::gegenbauer

The Gegenbauer (ultraspherical) polynomials

Syntax

```
orthpoly::gegenbauer(n, a, x)
```

Description

`orthpoly::gegenbauer(n,a,x)` computes the value of the n-th degree Gegenbauer polynomial with parameter a at the point x.

Evaluation for real floating-point values x from the interval [- 1.0, 1.0] is numerically stable. See “Example 2” on page 22-14.

Examples

Example 1

Polynomial expressions are returned, if identifiers or indexed identifiers are specified:

```
orthpoly::gegenbauer(2, a, x)
```

$$x^2 (2 a^2 + 2 a) - a$$

```
orthpoly::gegenbauer(3, 2, x[1])
```

$$32 x_1^3 - 12 x_1$$

Using arithmetical expressions as input, the “values” of these polynomials are returned:

```
orthpoly::gegenbauer(2, 1, 3+2*I)
```

$$19 + 48\,i$$

```
orthpoly::gegenbauer(3, 2, exp(x[1] + 2))
```

$$32\,e^{6}\,e^{3\,x_1} - 12\,e^{x_1}\,e^{2}$$

"Arithmetical expressions" include numbers:

```
orthpoly::gegenbauer(2, a, sqrt(2)),
orthpoly::gegenbauer(3, 0.4, 8 + I),
orthpoly::gegenbauer(1000, -1/3, 0.3)
```

$$4\,a^2 + 3\,a,\ 865.536 + 341.152\,i,\ 0.00006046127974$$

If the degree of the polynomial is a variable or expression, then `orthpoly::gegenbauer` returns itself symbolically:

```
orthpoly::gegenbauer(n, a, x)
```

$$\text{orthpoly::gegenbauer}(n,\ a,\ x)$$

Example 2

If a floating-point value is desired, then a direct call such as

```
orthpoly::gegenbauer(200, 4, 0.3)
```

$$165549.7263$$

is appropriate and yields a correct result. One should not evaluate the symbolic polynomial at a floating-point value, because this may be numerically unstable:

```
G200 := orthpoly::gegenbauer(200, 4, x):
DIGITS := 10: evalp(G200, x = 0.3)
```

$$-6.270612376\ 10^{11}$$

This result is caused by numerical round-off. Also with increased `DIGITS` only a few leading digits are correct:

```
DIGITS := 20: evalp(G200, x = 0.3)
```

```
     165454.59819021060509
```

```
delete DIGITS, G200:
```

Parameters

n

A nonnegative integer or an arithmetical expression representing a nonnegative integer: the degree of the polynomial.

a

An arithmetical expression.

x

An indeterminate or an arithmetical expression. An indeterminate is either an identifier (of domain type `DOM_IDENT`) or an indexed identifier (of type `"_index"`).

Return Values

The value of the Gegenbauer polynomial at point `x` is returned as an arithmetical expression. If `n` is an arithmetical expression, then `orthpoly::gegenbauer` returns itself symbolically.

Algorithms

The Gegenbauer polynomials are given by the recursion formula

$$G(n, a, x) = \frac{2\,(n-1+a)}{n}\,x\,G(n-1, a, x) - \frac{n-2+2\,a}{n}\,G(n-2, a, x)$$

with $G(0, a, x) = 1$, $G(1, a, x) = 2\,a\,x$.

For fixed real $a > -\frac{1}{2}$ these polynomials are orthogonal on the interval [- 1, 1] with respect to the weight function $w(x) = (1 - x^2)^{a-\frac{1}{2}}$.

$G\left(n, \frac{1}{2}, x\right)$ coincides with the Legendre polynomial $P(n, x)$.

$G(n, 1, x)$ coincides with the Chebyshev polynomial $U(n, x)$ of the second kind.

The polynomials $G(n, 0, x)$ are trivial.

See Also

MuPAD Functions

`orthpoly::chebyshev2` | `orthpoly::legendre`

orthpoly::hermite

The Hermite polynomials

Syntax

```
orthpoly::hermite(n, x)
```

Description

`orthpoly::hermite(n,x)` computes the value of the n-th degree Hermite polynomial at the point x.

These polynomials have integer coefficients.

Examples

Example 1

Polynomial expressions are returned if identifiers or indexed identifiers are specified:

```
orthpoly::hermite(2, x)
```

$$4\,x^2 - 2$$

```
orthpoly::hermite(3, x[1])
```

$$8\,x_1^{\,3} - 12\,x_1$$

Using arithmetical expressions as input, the “values” of these polynomials are returned:

```
orthpoly::hermite(2, 3+2*I)
```

$$18 + 48\,\mathrm{i}$$

```
orthpoly::hermite(3, exp(x[1] + 2))
```

$$8\,e^6\,e^{3\,x_1} - 12\,e^{x_1}\,e^2$$

"Arithmetical expressions" include numbers:

```
orthpoly::hermite(2, sqrt(2)), orthpoly::hermite(3, 8 + I),
orthpoly::hermite(1000, 0.3);
```

$$6,\ 3808 + 1516\,i,\ 2.26821486\ 10^{1433}$$

If the degree of the polynomial is a variable or expression, then `orthpoly::hermite` returns itself symbolically:

```
orthpoly::hermite(n, x)
```

$$\text{orthpoly::hermite}(n, x)$$

Parameters

n

A nonnegative integer or an arithmetical expression representing a nonnegative integer: the degree of the polynomial.

x

An indeterminate or an arithmetical expression. An indeterminate is either an identifier (of domain type `DOM_IDENT`) or an indexed identifier (of type `"_index"`).

Return Values

The value of the Hermite polynomial at point `x` is returned as an arithmetical expression. If `n` is an arithmetical expression, then `orthpoly::hermite` returns itself symbolically.

Algorithms

The Hermite polynomials are given by the recursion formula

$$H(n, x) = 2\,x\,H(n-1, x) - 2\,(n-1)\,H(n-2, x)$$

with $H(0, x) = 1$ and $H(1, x) = 2\,x$.

These polynomials are orthogonal on the real line with respect to the weight function $w(x) = e^{-x^2}$.

orthpoly::jacobi

The Jacobi polynomials

Syntax

```
orthpoly::jacobi(n, a, b, x)
```

Description

`orthpoly::jacobi(n,a,b,x)` computes the value of the n-th degree Jacobi polynomial with parameters a and b at the point x.

Evaluation for real floating-point values x from the interval [- 1.0, 1.0] is numerically stable. See "Example 2" on page 22-21.

Examples

Example 1

Polynomial expressions are returned if identifiers or indexed identifiers are specified:

```
orthpoly::jacobi(2, a, b, x)
```

$$x^2\left(\frac{a^2}{8}+\frac{a\,b}{4}+\frac{7\,a}{8}+\frac{b^2}{8}+\frac{7\,b}{8}+\frac{3}{2}\right)-\frac{b}{8}-\frac{a\,b}{4}-\frac{a}{8}+\frac{a^2}{8}+\frac{b^2}{8}+x\left(\frac{a^2}{4}+\frac{3\,a}{4}-\frac{b^2}{4}-\frac{3\,b}{4}\right)$$

$$-\frac{1}{2}$$

```
orthpoly::jacobi(3, 4, 5, x[1])
```

$$\frac{455\,x_1^{\,3}}{8}-\frac{91\,x_1^{\,2}}{8}-\frac{91\,x_1}{8}+\frac{7}{8}$$

Using arithmetical expressions as input, the "values" of these polynomials are returned:

```
orthpoly::jacobi(2, 4, 1, 3+2*I)
```

$$63 + 120\,i$$

```
orthpoly::jacobi(2, 0, I, exp(x[1] + 2))
```

$$e^{x_1}\,e^{2}\left(\frac{1}{4}-\frac{3\,i}{4}\right)+e^{4}\,e^{2\,x_1}\left(\frac{11}{8}+\frac{7\,i}{8}\right)-\frac{5}{8}-\frac{i}{8}$$

"Arithmetical expressions" include numbers:

```
orthpoly::jacobi(2, 1/2, -1/2, sqrt(2)),
orthpoly::jacobi(3, 2, 5, 8 + I),
orthpoly::jacobi(1000, 1, 2, 0.3);
```

$$\frac{3\,\sqrt{2}}{4}+\frac{21}{8},\ \frac{31733}{2}+\frac{12859\,i}{2},\ -0.06546648097$$

If the degree of the polynomial is a variable or expression, then `orthpoly::jacobi` returns itself symbolically:

```
orthpoly::jacobi(n, a, b, x)
```

$$\text{orthpoly::jacobi}(n,\ a,\ b,\ x)$$

Example 2

If a floating-point value is desired, then a direct call such as

```
orthpoly::jacobi(100, 1/2, 3/2, 0.9)
```

$$0.2560339406$$

is appropriate and yields a correct result. One should not evaluate the symbolic polynomial at a floating-point value, because this may be numerically unstable:

```
P100 := orthpoly::jacobi(100, 1/2, 3/2, x):

evalp(P100, x = 0.9)
```

$$-8.6781052\ 10^{15}$$

This result is caused by numerical round-off. Also with increased `DIGITS` only a few leading digits are correct:

```
DIGITS := 30: evalp(P100, x = 0.9)
```

```
     0.256030968488207303016930946513
```

```
delete P100, DIGITS:
```

Parameters

n

A nonnegative integer or an arithmetical expression representing a nonnegative integer: the degree of the polynomial.

a, b

Arithmetical expressions.

x

An indeterminate or an arithmetical expression. An indeterminate is either an identifier (of domain type `DOM_IDENT`) or an indexed identifier (of type `"_index"`).

Return Values

The value of the Jacobi polynomial at point `x` is returned as an arithmetical expression. If `n` is an arithmetical expression, then `orthpoly::jacobi` returns itself symbolically.

Algorithms

The Jacobi polynomials are given by the recursion formula

$$2\,n\,c_n\,c_{2\,n-2}\,P(n,a,b,x) = c_{2\,n-1}\left(c_{2\,n-2}\,c_{2\,n}\,x + a^2 - b^2\right)P(n-1,a,b,x)$$

$$-2\,(n-1+a)\,(n-1+b)\,c_{2\,n}\,P(n-2,a,b,x)$$

with $c_i = i + a + b$ and

$$P(0, a, b, x) = 1,\ P(1, a, b, x) = \frac{a-b}{2} + \left(1 + \frac{a+b}{2}\right) x$$

.

For fixed real $a > -1$, $b > -1$ the Jacobi polynomials are orthogonal on the interval $[-1, 1]$ with respect to the weight function $w(x) = (1 - x)^a (1 + x)^b$.

For special values of the parameters a, b the Jacobi polynomials are related to the Legendre polynomials

$$P(n, x) = P(n, 0, 0, x)$$

,

to the Chebyshev polynomials of the first kind

$$T(n, x) = \frac{2^{2n} n!^2}{(2n)!} P\left(n, -\frac{1}{2}, -\frac{1}{2}, x\right)$$

,

to the Chebyshev polynomials of the second kind

$$U(n, x) = \frac{2^{2n} n!\,(n+1)!}{(2n+1)!} P\left(n, \frac{1}{2}, \frac{1}{2}, x\right)$$

,

and to the Gegenbauer polynomials, respectively:

$$G(n, a, x) = \frac{\Gamma\left(a + \frac{1}{2}\right) \Gamma(n + 2a)}{\Gamma(2a)\, \Gamma\left(n + a + \frac{1}{2}\right)} P\left(n, a - \frac{1}{2}, a - \frac{1}{2}, x\right)$$

.

See Also

MuPAD Functions

`orthpoly::chebyshev1` | `orthpoly::chebyshev2` | `orthpoly::gegenbauer` | `orthpoly::legendre`

orthpoly::laguerre

The (generalized) Laguerre polynomials

Syntax

```
orthpoly::laguerre(n, a, x)
```

Description

`orthpoly::laguerre(n,a,x)` computes the value of the generalized n-th degree Laguerre polynomial with parameter a at the point x.

The standard Laguerre polynomials correspond to $a = 0$. They have rational coefficients.

Examples

Example 1

Polynomial expressions are returned if identifiers or indexed identifiers are specified:

```
orthpoly::laguerre(2, a, x)
```

$$\frac{3\,a}{2} - x\,(a+2) + \frac{a^2}{2} + \frac{x^2}{2} + 1$$

```
orthpoly::laguerre(3, a, x[1])
```

$$\frac{11\,a}{6} - \frac{x_1^{\,3}}{6} - x_1\left(\frac{a^2}{2} + \frac{5\,a}{2} + 3\right) + \left(\frac{a}{2} + \frac{3}{2}\right)x_1^{\,2} + a^2 + \frac{a^3}{6} + 1$$

Using arithmetical expressions as input, the “values” of these polynomials are returned:

```
orthpoly::laguerre(2, 4, 3+2*I)
```

$$-\frac{1}{2}-6\,i$$

```
orthpoly::laguerre(2, 2/3*I, exp(x[1] + 2))
```

$$\frac{e^4\,e^{2\,x_1}}{2}+e^{x_1}\,e^2\left(-2-\frac{2\,i}{3}\right)+\frac{7}{9}+i$$

“Arithmetical expressions” include numbers:

```
orthpoly::laguerre(2, a, sqrt(2)),
orthpoly::laguerre(3, 0.4, 8 + I),
orthpoly::laguerre(1000, 3, 0.3);
```

$$\frac{3\,a}{2}-\sqrt{2}\,a-2\,\sqrt{2}+\frac{a^2}{2}+2,\ -4.969333333-8.713333333\,i,\ -15691.69498$$

If the degree of the polynomial is a variable or expression, then `orthpoly::laguerre` returns itself symbolically:

```
orthpoly::laguerre(n, a, x)
```

$$\text{orthpoly::laguerre}(n,\,a,\,x)$$

Parameters

n

A nonnegative integer or an arithmetical expression representing a nonnegative integer: the degree of the polynomial.

a

An arithmetical expression.

x

An indeterminate or an arithmetical expression. An indeterminate is either an identifier (of domain type `DOM_IDENT`) or an indexed identifier (of type `"_index"`).

Return Values

The value of the Laguerre polynomial at point `x` is returned as an arithmetical expression. If `n` is an arithmetical expression, then `orthpoly::laguerre` returns itself symbolically.

Algorithms

The Laguerre polynomials are given by the recursion formula

$$L(n, a, x) = \frac{2n + a - 1 - x}{n} L(n-1, a, x) - \frac{n + a - 1}{n} L(n-2, a, x)$$

with $L(0, a, x) = 1$ and $L(1, a, x) = 1 + a - x$.

For fixed real $a > -1$ these polynomials are orthogonal on the interval $[0, \infty)$ with respect to the weight function $w(x) = \frac{1}{(x^{ae})^x}$.

orthpoly::legendre

The Legendre polynomials

Syntax

```
orthpoly::legendre(n, x)
```

Description

`orthpoly::legendre(n,x)` computes the value of the n-th degree Legendre polynomial at the point x.

These polynomials have rational coefficients.

Evaluation for real floating-point values x from the interval [- 1.0, 1.0] is numerically stable. See “Example 2” on page 22-28.

Use `numeric::gldata` to compute the roots of the Legendre polynomials. Cf. “Example 3” on page 22-29.

Examples

Example 1

Polynomial expressions are returned if identifiers or indexed identifiers are specified:

```
orthpoly::legendre(2, x)
```

$$\frac{3\,x^2}{2} - \frac{1}{2}$$

```
orthpoly::legendre(3, x[1])
```

$$\frac{5\,x_1^{\,3}}{2} - \frac{3\,x_1}{2}$$

Using arithmetical expressions as input, the "values" of these polynomials are returned:

```
orthpoly::legendre(2, 3+2*I)
```

$$7 + 18\,i$$

```
orthpoly::legendre(3, exp(x[1] + 2))
```

$$\frac{5\,e^{6}\,e^{3\,x_1}}{2} - \frac{3\,e^{x_1}\,e^{2}}{2}$$

"Arithmetical expressions" include numbers:

```
orthpoly::legendre(2, sqrt(2)), orthpoly::legendre(3, 8 + I),
orthpoly::legendre(1000, 0.3)
```

$$\frac{5}{2},\ 1208 + 476\,i,\ -0.02566916751$$

If the degree of the polynomial is a variable or expression, then `orthpoly::legendre` returns itself symbolically:

```
orthpoly::legendre(n, x)
```

$$\text{orthpoly::legendre}(n, x)$$

Example 2

If a floating-point value is desired, then a direct call such as

```
orthpoly::legendre(100, 0.9)
```

$$0.1022658206$$

is appropriate and yields a correct result. One should not evaluate the symbolic polynomial at a floating-point value, because this may be numerically unstable:

```
P100 := orthpoly::legendre(100, x):

evalp(P100, x = 0.9)
```

$$8.284745953\ 10^{14}$$

This result is caused by numerical round-off. Also with increased `DIGITS` only a few leading digits are correct:

```
DIGITS := 30: evalp(P100, x = 0.9)
```

$$0.102276303910546875548266967112$$

```
delete P100, DIGITS:
```

Example 3

We recommend to use `numeric::gldata` for computing roots of the Legendre polynomial $P(n, x)$. This routine provides all roots of the function $Q(n, y) = P(n, 2\,y - 1)$:

```
QRoots := numeric::gldata(5, DIGITS)[2]
```

$$\left[0.04691007703,\ 0.2307653449,\ \frac{1}{2},\ 0.7692346551,\ 0.953089923\right]$$

These values are easily transformed to roots of $P(n, x)$:

```
PRoots := map(QRoots, y -> 2*y - 1)
```

$$[-0.9061798459,\ -0.5384693101,\ 0,\ 0.5384693101,\ 0.9061798459]$$

```
orthpoly::legendre(5, r) $ r in PRoots
```

$$-1.080385781\ 10^{-14},\ -1.387778781\ 10^{-18},\ 0,\ 1.387778781\ 10^{-18},\ 1.081218448\ 10^{-14}$$

```
delete QRoots, PRoots:
```

Parameters

n

A nonnegative integer or an arithmetical expression representing a nonnegative integer: the degree of the polynomial.

x

An indeterminate or an arithmetical expression. An indeterminate is either an identifier (of domain type `DOM_IDENT`) or an indexed identifier (of type `"_index"`).

Return Values

The value of the Legendre polynomial at point `x` is returned as an arithmetical expression. If `n` is an arithmetical expression, then `orthpoly::legendre` returns itself symbolically.

Algorithms

The Legendre polynomials are given by $P(n, x) = \frac{1}{2^n\, n!}\, \frac{d^n}{dx^n}(x^2 - 1)^n$.

They satisfy the recursion formula

$$P(n, x) = \frac{2\,n-1}{n}\, x\, P(n-1, x) - \frac{n-1}{n}\, P(n-2, x)$$

with $P(0, x) = 1$ and $P(1, x) = x$.

They are orthogonal on the interval [- 1, 1] with respect to the weight function $w(x) = 1$.

$P(n, x)$ coincides with the Gegenbauer polynomial $G\left(n, \frac{1}{2}, x\right)$.

$P(n, x)$ coincides with the Jacobi polynomial $P(n, 0, 0, x)$.

See Also

MuPAD Functions

`numeric::gldata` | `orthpoly::gegenbauer` | `orthpoly::jacobi`

output – Formatted Output

output::asciiAbbreviate
output::fence
output::mathText
output::ordinal
output::roman
output::subexpr
output::tableForm
output::tree

output::asciiAbbreviate

Generates a procedure for creating an ASCII formatted output

Syntax

```
output::asciiAbbreviate(<varname>)
```

Description

`output::asciiAbbreviate` generates a procedure for creating ASCII formatted outputs of expressions

By default, MuPAD generates ASCII abbreviations using the # symbol followed by numbers. Using the argument `varname` you can customize the names of abbreviated subexpressions. See “Example 2” on page 23-3.

If you want to use abbreviations in all ASCII PrettyPrint output expressions, pass the procedure generated by `output::asciiAbbreviate` to `Pref::output`. See “Example 3” on page 23-3.

Examples

Example 1

The generated procedure `ascii` produces ASCII code for MuPAD expressions:

```
ascii:=output::asciiAbbreviate():
y := solve(x^3 + x + 1 = 0, x, MaxDegree = 3):
ascii(y)
```

```
{                                 /   1       \
{                          sqrt(3) | ---- + #1 | I
{        1     1    #1            \ 3 #1      /
{ #1 - ----, ---- - -- - ----------------------,
{      3 #1  6 #1    2               2
```

```
                      /   1       \    }
                sqrt(3) | ---- + #1 | I }
    1    #1            \ 3 #1      /    }
  ---- - -- + ----------------------- }
  6 #1    2               2            }

where

        / sqrt(31) sqrt(108)   1 \1/3
   #1 = | ------------------ - - |
        \          108         2 /
```

Example 2

You can customize the names of abbreviated subexpressions:

```
ascii:=output::asciiAbbreviate(t):
y := solve(x^3 + x + 1 = 0, x, MaxDegree = 3):
ascii(y)
```

```
{                                /   1       \
{                          sqrt(3) | ---- + t1 | I
{        1     1    t1            \ 3 t1      /
{ t1 - ----, ---- - -- - ----------------------,
{      3 t1  6 t1    2               2

                      /   1       \    }
                sqrt(3) | ---- + t1 | I }
    1    t1            \ 3 t1      /    }
  ---- - -- + ----------------------- }
  6 t1    2               2            }

where

        / sqrt(31) sqrt(108)   1 \1/3
   t1 = | ------------------ - - |
        \          108         2 /
```

Example 3

The generated procedure can serve as an input for `Pref::output`:

```
Pref::output(output::asciiAbbreviate(u)):
y := solve(x^3 + x + 1 = 0, x, MaxDegree = 3)
```

```
{                                  /   1       \
{                          sqrt(3) | ---- + u1 | I
{        1     1    u1            \ 3 u1      /
{ u1 - ----, ---- - -- + ----------------------,
{      3 u1  6 u1    2               2

                       /   1       \    }
               sqrt(3) | ---- + u1 | I }
    1    u1            \ 3 u1      /    }
  ---- - -- - ----------------------- }
  6 u1    2               2            }

where

        / sqrt(31) sqrt(108)   1 \1/3
   u1 = | ------------------ - - |
        \          108         2 /
```

Parameters

`varname`

A base name for the abbreviation variables

Return Values

Procedure for creating an ASCII formatted output

See Also

MuPAD Functions

`output::subexpr` | `Pref::abbreviateOutput` | `Pref::output`

output::fence

Put delimiters around multiline strings

Syntax

```
output::fence(left, right, string, <width, <base>>)
```

Description

`output::fence(left, right, string)` encloses the multiline string in the delimiters indicated by `left` and `right`

You can use `output::fence` in combination with `strprint` to overload `print` (and thereby, standard screen output) for pretty-printing. Note that it does not affect typeset output.

Examples

Example 1

Set `TEXTWIDTH` to 75:

```
TEXTWIDTH := 75:
```

First, define a domain that uses `output::fence` for output:

```
domain Fence
  print := x -> output::fence("(", ")", extop(x, 1));
  new   := x -> new(dom, x);
end_domain:
```

`Fence` expects a string in its constructor. The output uses `output::fence` to put parentheses around the input:

```
print(Plain, Fence("abc"))
```

```
(abc)
```

Strings of height two are only partly placed inside parentheses, for consistency with the pretty-printer:

```
print(Plain, Fence("abc\ndef"), sin(x^2))
```

```
 abc        2
(def), sin(x )
```

Strings of height more than two are fully bracketed:

```
print(Plain, Fence("abc\ndef\nghi"))
```

```
/ abc \
| def |
\ ghi /
```

Example 2

Set `TEXTWIDTH` to 75:

```
TEXTWIDTH := 75:
```

The next step in using `output::fence` is to enclose expressions in parentheses. For this, the information from `strprint` is useful:

```
domain FenceExpr
  print := proc(x)
             local str, h1, w1, h, w, b;
           begin
             [str, h1, w1, h, w, b] := strprint(All, extop(x));
             output::fence("{", "]", str, w, b);
           end_proc;
  new   := x -> new(dom, x);
end_domain:
```

The sixth operand of the return value of `strprint(All, …)` must be given to `output::fence` to align baselines properly:

```
print(Plain, FenceExpr(x), FenceExpr(x^2), FenceExpr(x^2/2*y))
```

```
           {  2   --
       2   { x  y  |
{x], {x ], { ----  |
           {   2  --
```

`strprint` reacts to `TEXTWIDTH` and can return a string consisting or more than one logical line. In this case, fencing the returned string leads to strange results:

```
print(Plain, FenceExpr(_plus(x.i $ i = 0..30)))
```

```
{ x0 + x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10 + x11 + x12 + x13 \
   --
{                                                                        \
    |
{    + x14 + x15 + x16 + x17 + x18 + x19 + x20 + x21 + x22 + x23 + x24 + x\
25  |
{                                                                        \
    |
{    + x26 + x27 + x28 + x29 + x30                                       \
   --
```

Here, the string with added delimiter symbols is too wide for `TEXTWIDTH`. Also, `output::fence` does not let you control line breaks. Therefore, it is a good practice to avoid putting large delimiters to the left and right of long strings. For example, `abs` prints in functional notation for long arguments:

```
print(Plain, abs(_plus(x.i $ i = 0..30)))
```

```
abs(x0 + x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10 + x11 + x12

   + x13 + x14 + x15 + x16 + x17 + x18 + x19 + x20 + x21 + x22 + x23 + x24

   + x25 + x26 + x27 + x28 + x29 + x30)
```

Parameters

`left, right`

Strings indicating the type of delimiter: `"("`, `")"`, `"["`, `"]"`, `"[+"`, `"+]"`, `"{"`, `"}"`, `"|"`, `"|  "`, or `"  |"`.

`string`

The string to enclose

`width`

The width of the string to enclose. Defaults to the width of the widest line in `string`.

`base`

The baseline of the string, counted from the first line. Defaults to the bottom line of the string. If set to `-1`, the baseline is vertically centered.

Return Values

String

See Also

MuPAD Functions
`strprint`

output::mathText

Pretty output of text combined with formulas

Syntax

```
output::mathText(str1, expr1, <str2, <expr2>, …>)
```

Description

`output::mathText( str1, expr1, str2, … )` creates an object of combined strings and expressions. This object prints itself nicely in various output formats.

Examples

Example 1

`output::mathText` creates an object representing combined text and formulas:

```
messageWithMath := output::mathText("The integral ",
                            hold(int(sin(x)*cos(x), x)),
                            " is equal to ", int(sin(x)*cos(x), x))
```

$$\text{The integral } \int \cos(x)\,\sin(x)\,dx \text{ is equal to } \frac{\sin(x)^2}{2}$$

They can be printed with the ASCII pretty-printer as well:

```
print(Plain, messageWithMath)
```

```
                /                                      2
               |                                   sin(x)
The integral   |  sin(x) cos(x) dx is equal to     -------
              /                                      2
```

Same for the ASCII lineprint output:

```
PRETTYPRINT := FALSE:
print(Plain, messageWithMath):
delete PRETTYPRINT:
```

```
The integral int(sin(x)*cos(x), x) is equal to (1/2)*sin(x)^2
```

Parameters

str_1, str_2

Strings

$expr_1$, $expr_2$

Expressions

Return Values

Object of type `output::mathText`

See Also

MuPAD Functions
`strprint`

output::ordinal

Ordinal numbers

Syntax

```
output::ordinal(i)
```

Description

`output::ordinal` converts an integer to the corresponding english ordinal number. The return value is a string and can be used in messages.

Examples

Example 1

Convert some numbers to the corresponding english ordinal string:

```
map([0, 1, 2, 3, 4, 22, 134, 2001], output::ordinal)
```

```
["0th", "1st", "2nd", "3rd", "4th", "22nd", "134th", "2001st"]
```

Parameters

`i`

An integer number

Return Values

String with the English ordinal number

See Also

MuPAD Functions
`info` | `print`

output::roman

Roman numerals

Syntax

```
output::roman(n)
```

Description

`output::roman` converts an integer to the corresponding roman numeral. The return value is a string and can be used in messages.

Examples

Example 1

Convert some numbers to the corresponding roman numerals:

```
map([1, 2, 3, 4, 22, 134, 2001], output::roman)
```

```
["I", "II", "III", "IV", "XXII", "CXXXIV", "MMI"]
```

Now, thanks to alias backsubstitution, we can trick MuPAD into computing with roman numerals:

```
alias(`I`=1): // I is a reserved word in MuPAD, so we use `I` instead
for i from 2 to 1000 do
    eval(text2expr("alias(".output::roman(i)."=".expr2text(i).");"));
end_for:
II+II;
XIII*XXIV
```

```
IV
```

CCCXII

```
unalias(`I`):
for i from 2 to 1000 do
    eval(text2expr("unalias(".output::roman(i).");"));
end_for:
```

Parameters

n

Positive integer

Return Values

String containing the roman numeral

See Also

MuPAD Functions

`info` | `print`

output::subexpr

Abbreviates a expression

Syntax

```
output::subexpr(e, <varname>)
```

Description

`output::subexpr` rewrites symbolic expression in terms of common subexpressions.

If an expression `e` contains common subexpressions, `output::subexpr(e)` returns a list that contains the abbreviated expression and the abbreviations in a form of equations. With `output::subexpr` you get the same abbreviations as you see in the outputs. See “Example 1” on page 23-15.

An output of this command does not depend on the current setting of `Pref::abbreviateOutput`.

By default, MuPAD generates abbreviations using the # symbol followed by numbers. Using the argument `varname`, you can customize the names of abbreviated subexpressions. See “Example 2” on page 23-16.

Examples

Example 1

You can abbreviate an expression:

```
y := solve(x^3 + x + 1 = 0, x, MaxDegree = 3):
output::subexpr(y)
```

$$\left[\left\{\#_1 - \frac{1}{3\,\#_1},\ \frac{1}{6\,\#_1} - \frac{\#_1}{2} + \frac{\sqrt{3}\left(\frac{1}{3\,\#_1} + \#_1\right)i}{2},\ \frac{1}{6\,\#_1} - \frac{\#_1}{2} - \frac{\sqrt{3}\left(\frac{1}{3\,\#_1} + \#_1\right)i}{2}\right\},\ \#_1 = \left(\frac{\sqrt{31}\,\sqrt{108}}{108} - \frac{1}{2}\right)^{1/3}\right]$$

Example 2

You can customize the names of abbreviated subexpressions:

```
y := solve(x^3 + x + 1 = 0, x, MaxDegree = 3):
output::subexpr(y, t)
```

$$\left[\left\{t1 - \frac{1}{3\,t1},\ \frac{1}{6\,t1} - \frac{t1}{2} - \frac{\sqrt{3}\left(\frac{1}{3\,t1} + t1\right)i}{2},\ \frac{1}{6\,t1} - \frac{t1}{2} + \frac{\sqrt{3}\left(\frac{1}{3\,t1} + t1\right)i}{2}\right\},\ t1 = \left(\frac{\sqrt{31}\,\sqrt{108}}{108} - \frac{1}{2}\right)^{1/3}\right]$$

Parameters

e

A MuPAD expression

varname

A base name for the abbreviation variables

Return Values

List that contains the abbreviated expression and the abbreviations as equations

See Also

MuPAD Functions

`output::asciiAbbreviate` | `Pref::abbreviateOutput`

output::tableForm

Printing objects in table form

Syntax

```
output::tableForm(obj, <separator>, options)
```

Description

`output::tableForm(obj)` prints the elements of the given object `obj` in table form.

The width of the table and the number of columns depends on the size of `TEXTWIDTH` (see options `Width` and `Columns`). The width of a column depends on the widest entry in this column.

`output::tableForm` determines the number of columns, that the total width of the table fits into `TEXTWIDTH`.

The columns are separated by one space by default.

If `separator` is given, then it is printed between each column (instead of one space). Appending spaces to the separator results additionally space between columns. By default the separator is one space.

If the first argument `obj` is a table or a domain, `output::tableForm` uses the option `Columns = 2` (two columns) and the separator `"= "` as default.

Without the option `Sort` the objects are converted to strings and then sorted alphabetically. To avoid any sorting the option `Sort = FALSE` must be given.

Examples

Example 1

For all examples on this page we assume the `TEXTWIDTH` 75:

```
TEXTWIDTH := 75:
```

Print some random numbers in table form:

```
SEED := -1:
output::tableForm([random(100000)() $ k = 1..30])
```

```
11647 12826 26280 26292 28315 30908 36523 42073 4682  47334 52640 56426

5829 615   62580 65904 66223 6719  69451 69903 77904 78221 80528 81013

86068 89016 90516 91008 92791 9532
```

Some random strings are created. The columns should have all the same width (`Unique`) and be printed centered. The strings should not be sorted:

```
output::tableForm([_concat("*" $ random(10)()) $ k = 1..20],
                  Unique, Center, Sort = FALSE)
```

```
 "********"  "******"   "*********" "********"      "*"     "******"

"*********"   "*****"     "****"    "********"    "****"    "*****"

"*********"  "*******"   "*******"   "******"      "**"     "****"
```

The option `Unquoted` prevents printing of quotes (see `fprint`):

```
output::tableForm([_concat("*" $ random(10)()) $ k = 1..20],
                  Unique, Center, Sort = FALSE, Unquoted)
```

```
********    *      ******   ******    ****    ******     **    **

  *****     *       ****     ****      **    ******** *******  **

    *    *******
```

Example 2

The next object is a MuPAD table and should be printed as a table with two columns. The table contains some random numbers and their sum of the digits:

```
SEED := -1:
T :=  table(op(map([random(100000000)() $ k = 1..10],
                   proc(X)
                     local Xs, k;
```

```
                    begin
                      Xs := expr2text(X);
                      X = _plus(text2expr(substring(Xs, k))
                                $ k = 1..length(Xs))
                    end_proc))):
output::tableForm(T)

19962580 = 40

25878221 = 35

37777904 = 44

41281013 = 20

43856426 = 38

46169451 = 36

66926292 = 42

80330908 = 31

89306719 = 43

94386068 = 44
```

Domains are also printed in this form by default:

```
output::tableForm(newDomain("Test",
                            table("type" = "Test",
                                  "info" = "only a testdomain")))

"info" = "only a testdomain"

"key" = "Test"

"type" = "Test"
```

Example 3

The next table should consist of four columns:

```
SEED := -1:
output::tableForm([random(100000)() $ k = 1..30], Columns = 4)
```

```
11647  12826  26280  26292

28315  30908  36523  42073

4682  47334  52640  56426

5829  615    62580  65904

66223  6719   69451  69903

77904  78221  80528  81013

86068  89016  90516  91008

92791  9532
```

The next table should have a maximal width of 50 characters:

```
SEED := -1:
output::tableForm([random(100000)() $ k = 1..30], Width = 50)
```

```
11647  12826  26280  26292  28315  30908  36523  42073

4682  47334  52640  56426  5829   615    62580  65904

66223  6719   69451  69903  77904  78221  80528  81013

86068  89016  90516  91008  92791  9532
```

```
delete T:
```

Example 4

The next examples show different usage of separators. First one single separator:

```
SEED := -1:
output::tableForm([random(100000)() $ k = 1..30], "|")
```

```
11647|12826|26280|26292|28315|30908|36523|42073|4682 |47334|52640|56426

5829|615  |62580|65904|66223|6719 |69451|69903|77904|78221|80528|81013

86068|89016|90516|91008|92791|9532
```

Now a list with a separator character between each column. If the list is too short, the characters are used from beginning of the list again etc.:

```
SEED := -1:
output::tableForm([random(100000)() $ k = 1..30], ["|", " ", " "])
```

```
11647|12826 26280 26292|28315 30908 36523|42073 4682  47334|52640 56426

5829|615   62580 65904|66223 6719  69451|69903 77904 78221|80528 81013

86068|89016 90516 91008|92791 9532
```

Only the first both colunms should be separated by a vertical line:

```
SEED := -1:
output::tableForm([random(100000)() $ k = 1..30], ["|", " " $ 10])
```

```
11647|12826 26280 26292 28315 30908 36523 42073 4682  47334 52640 56426

5829|615   62580 65904 66223 6719  69451 69903 77904 78221 80528 81013

86068|89016 90516 91008 92791 9532
```

Additionally a character can be appended to each entry:

```
SEED := -1:
output::tableForm([random(100000)() $ k = 1..30],
                  ["|", " " $ 10], Append = ",")
```

```
11647,|12826, 26280, 26292, 28315, 30908, 36523, 42073, 4682,  47334, 52640,

56426,|5829,  615,   62580, 65904, 66223, 6719,  69451, 69903, 77904, 78221,

80528,|81013, 86068, 89016, 90516, 91008, 92791, 9532,
```

Example 5

The next examples show different usage of sorting. Without the option `Sort` the numbers are sorted as strings in lexicographical order:

```
SEED := -1:
output::tableForm([random(100000)() $ k = 1..30])
```

```
11647 12826 26280 26292 28315 30908 36523 42073 4682  47334 52640 56426

5829 615   62580 65904 66223 6719  69451 69903 77904 78221 80528 81013

86068 89016 90516 91008 92791 9532
```

`Sort = FALSE` avoids any sorting:

```
SEED := -1:
output::tableForm([random(100000)() $ k = 1..30], Sort = FALSE)
```

```
30908 6719  26292 56426 81013 69451 77904 78221 86068 62580 47334 12826

9532 5829  28315 65904 42073 80528 4682  52640 69903 92791 36523 26280

89016 91008 615   66223 90516 11647
```

Any sorting can be done with a special defined procedure, e.g., sort the numbers in reverse order:

```
SEED := -1:
output::tableForm([random(100000)() $ k = 1..30],
                  Sort = ((X,Y) -> Y < X))
```

```
92791 91008 90516 89016 86068 81013 80528 78221 77904 69903 69451 66223

65904 62580 56426 52640 47334 42073 36523 30908 28315 26292 26280 12826

11647 9532  6719  5829  4682  615
```

At last a user defined procedure is given that sorts the numbers by the sum of their digits ascending:

```
SEED := -1:
output::tableForm([random(100000)() $ k = 1..30],
                  Sort = proc(X,Y)
                           local crossfoot;
                         begin
                           crossfoot :=
                             proc(X)
                               local Xs, k;
                             begin
                               Xs := expr2text(X);
                               _plus(text2expr(substring(Xs, k))
                                     $ k = 1..length(Xs))
                             end_proc;
                           crossfoot(X) < crossfoot(Y)
                         end_proc)
```

```
615  81013 42073 52640 91008 26280 36523 11647 12826 66223 9532 28315

30908 4682  78221 26292 90516 47334 62580 56426 80528 6719  5829 65904
```

```
89016  69451  77904  69903  92791  86068
```

Parameters

`obj`

A list, set or table of any MuPAD objects or a domain

`separator`

A string between columns

Options

`Unquoted`

Strings are printed without quotes

The output function `fprint` is called with the option `Unquoted`.

`Unique`

All columns are of the same width

All columns are printed with the same width, the widest column determines the width of each column.

`Width`

Option, specified as `Width = w`

The maximal width of the table is set to `w` (instead of `TEXTWIDTH`). `w` must be a positive integer.

`Columns`

Option, specified as `Columns = c`

The number of columns is set to `c`. The width of the table depends on the width of any column.

Note: `output::tableForm` called with this option takes not care about the value of `TEXTWIDTH`.

`c` must be a positive integer.

`Center, Left, Right`

The entries of each column are aligned left-justified, centered or justified.

`Sort`

Option, specified as `Sort = procedure`

The entries are sorted with the given procedure. Entries can be printed unsorted, when `procedure` is the object `FALSE`.

`Output`

Option, specified as `Output = file`

Output into a file. If `file` is a string, a file named `file` is opened and overwritten and closed after writing. If `file` is a file descriptor (the return value of `fopen`), the table is appended to `file` without closing `file`.

`String`

Return as a string that can be printed

The string contains line breaks, and can be printed with `print` or `fprint` and option `Unquoted`.

`Append`

Option, specified as `Append = string`

Character string is appended to each entry of the list

Return Values

Void object `null()`

See Also

MuPAD Functions
`fclose` | `fopen` | `fprint` | `output::tree` | `print` | `sort`

output::tree

Display of trees

Syntax

```
output::tree(Tree, <indentdepth, <charlist>, <Small>>)
```

Description

`output::tree` displays trees given as specially MuPAD lists.

The first object of the list is the root of the tree. All further objects are nodes or subtrees of the tree. A subtree is again a special list (as described), and any other MuPAD object will be interpreted as node of the tree.

The elements of the tree will be printed by MuPAD, when the tree will be displayed, so it's recommended to use strings as objects or objects with a well defined display.

The return value is a string that contains all chars to display the tree. With functions `print` and `fprint`*and* the option `Unquoted` the tree can be displayed.

The parameter `charlist` is a list with five characters. The default value is `["|", "+", "-", „", " "]`. The characters have the following meaning (described in the order of the list).

The vertical lines of the tree, the connection between vertical and horizontal line (i.e., an arm, but not the last arm), an arm (vertical line), the last connection to an arm in a subtree, a char between an arm and the description of the arm.

Examples

Example 1

`output::tree` displayes special nested lists as trees:

```
TREE := ["a1", "a2", ["b1", "b2", ["c1", "c2"], "b3"],
                ["d1", "d2", "d3"]]:
```

```
print(Unquoted, output::tree(TREE))
```

```
a1
|
+-- a2
|
+-- b1
|   |
|   +-- b2
|   |
|   +-- c1
|   |   |
|   |   `-- c2
|   |
|   `-- b3
|
`-- d1
    |
    +-- d2
    |
    `-- d3
```

```
print(Unquoted, output::tree(TREE, 3, Small)):
```

```
a1
+- a2
+- b1
|  +- b2
|  +- c1
|  |  `- c2
|  `- b3
`- d1
   +- d2
   `- d3
```

The chars can be defined by the user:

```
print(Unquoted, output::tree(TREE, 6, ["|", "|", ".", "\\", " "])):
```

```
a1
|
|.... a2
|
```

```
|.... b1
|      |
|      |.... b2
|      |
|      |.... c1
|      |      |
|      |      \.... c2
|      |
|      \.... b3
|
\.... d1
       |
       |.... d2
       |
       \.... d3
```

Parameters

`Tree`

The tree, given as a special list

`indentdepth`

Indent depth for each subtree

`charlist`

The chars that illustrate the tree structure

Options

`Small`

Suppresses the display of a space line between every tree entry to reduce the height of the tree

Return Values

String object to display

See Also

MuPAD Functions
`adt::Tree` | `prog::exprlist` | `prog::exprtree`

Graphics and Animations

RGB::ColorNames
RGB::plotColorPalette
RGB::colorName
RGB::fromWaveLength
RGB::random
RGB::toHSV
RGB::fromHSV
plot::easy
plot::getDefault
plot::setDefault
plot::copy
plot::modify
plot::delaunay
plot::hull
plot::Arc2d
plot::Arc3d
plot::Arrow2d
plot::Arrow3d
plot::Bars2d
plot::Bars3d
plot::Box
plot::Boxplot
plot::Circle2d
plot::Circle3d
plot::Cone
plot::Conformal
plot::Curve2d
plot::Curve3d
plot::Cylinder
plot::Cylindrical
plot::Density
plot::Ellipse2d

plot::Ellipse3d
plot::Function2d
plot::Function3d
plot::Hatch
plot::Histogram2d
plot::Implicit2d
plot::Implicit3d
plot::Inequality
plot::Integral
plot::Iteration
plot::Line2d
plot::Line3d
plot::Listplot
plot::Lsys
plot::Matrixplot
plot::MuPADCube
plot::Ode2d
plot::Ode3d
plot::Parallelogram2d
plot::Parallelogram3d
plot::Piechart2d
plot::Piechart3d
plot::Plane
plot::Point2d
plot::Point3d
plot::PointList2d
plot::PointList3d
plot::Polar
plot::Polygon2d
plot::Polygon3d
plot::Prism
plot::Pyramid
plot::QQplot
plot::Raster
plot::Rectangle
plot::Rootlocus
plot::Scatterplot
plot::Sequence
plot::SparseMatrixplot
plot::Sphere

plot::Ellipsoid
plot::Spherical
plot::Streamlines2d
plot::Sum
plot::Surface
plot::SurfaceSet
plot::SurfaceSTL
plot::Sweep
plot::Tetrahedron
plot::Hexahedron
plot::Octahedron
plot::Dodecahedron
plot::Icosahedron
plot::Text2d
plot::Text3d
plot::Tube
plot::Turtle
plot::VectorField2d
plot::VectorField3d
plot::Waterman
plot::XRotate
plot::ZRotate
plot::Canvas
plot::CoordinateSystem2d
plot::CoordinateSystem3d
plot::Group2d
plot::Group3d
plot::Scene2d
plot::Scene3d
plot::ClippingBox
plot::Reflect2d
plot::Reflect3d
plot::Rotate2d
plot::Rotate3d
plot::Scale2d
plot::Scale3d
plot::Transform2d
plot::Transform3d
plot::Translate2d
plot::Translate3d

plot::AmbientLight
plot::Camera
plot::DistantLight
plot::PointLight
plot::SpotLight
OutputFile, OutputOptions
AffectViewingBox
Angle
AngleRange, AngleBegin, AngleEnd
Area
Averaged
Axis, AxisX, AxisY, AxisZ
Base, Top, BaseX, TopX, BaseY, TopY, BaseZ, TopZ
BaseRadius, TopRadius
Cells
CellsClosed, ClassesClosed
Center, CenterX, CenterY, CenterZ
Closed
ColorData
CommandList
Contours
CoordinateType
Data
DensityData, DensityFunction
Edges
Extension
From, To, FromX, FromY, FromZ, ToX, ToY, ToZ
Function, XFunction, YFunction, ZFunction
Function1, Function2, Baseline
InitialConditions, TimeMesh
IntMethod
Generations, RotationAngle, IterationRules, StartRule, StepLength, TurtleRules
Ground
Heights, Moves
Inequalities
InputFile
Iterations, StartingPoint
LineColorFunction, FillColorFunction
Matrix2d, Matrix3d
MeshList, MeshListType, MeshListNormals

Name
Nodes
Normal, NormalX, NormalY, NormalZ
ParameterName, ParameterBegin, ParameterEnd, ParameterRange
Points2d, Points3d
Position, PositionX, PositionY, PositionZ
Radius
RadiusFunction
RationalExpression
Scale, ScaleX, ScaleY, ScaleZ
SemiAxes, SemiAxisX, SemiAxisY, SemiAxisZ
Shift, ShiftX, ShiftY, ShiftZ
Size
Tangent1, Tangent1X, Tangent1Y, Tangent1Z, Tangent2, Tangent2X, Tangent2Y, Tangent2Z
Text
TextOrientation
TextRotation
UName, URange, UMin, UMax, VName, VRange, VMin, VMax, XName, XRange, XMin, XMax, YName, YRange, YMin, YMax, ZName, ZRange, ZMin, ZMax
ViewingBox, ViewingBoxXMin, ViewingBoxXMax, ViewingBoxXRange, ViewingBoxYMin, ViewingBoxYMax, ViewingBoxYRange, ViewingBoxZMin, ViewingBoxZMax, ViewingBoxZRange
Visible
XFunction1, YFunction1, ZFunction1, XFunction2, YFunction2, ZFunction2
Axes
AxesInFront
AxesLineColor
AxesLineWidth
AxesOrigin, AxesOriginX, AxesOriginY, AxesOriginZ
AxesTips
AxesTitleAlignment, XAxisTitleAlignment, YAxisTitleAlignment, ZAxisTitleAlignment
AxesTitles, XAxisTitle, YAxisTitle, ZAxisTitle
AxesVisible, XAxisVisible, YAxisVisible, ZAxisVisible
YAxisTitleOrientation
TicksAnchor, XTicksAnchor, YTicksAnchor, ZTicksAnchor
TicksAt, XTicksAt, YTicksAt, ZTicksAt
TicksBetween, XTicksBetween, YTicksBetween, ZTicksBetween
TicksDistance, XTicksDistance, YTicksDistance, ZTicksDistance
TicksLabelStyle, XTicksLabelStyle, YTicksLabelStyle, ZTicksLabelStyle

TicksLength
TicksNumber, XTicksNumber, YTicksNumber, ZTicksNumber
TicksVisible, XTicksVisible, YTicksVisible, ZTicksVisible
TicksLabelsVisible, XTicksLabelsVisible, YTicksLabelsVisible, ZTicksLabelsVisible
GridInFront
GridLineColor, SubgridLineColor
GridLineStyle, SubgridLineStyle
GridLineWidth, SubgridLineWidth
GridVisible, SubgridVisible, XGridVisible, XSubgridVisible, YGridVisible, YSubgridVisible, ZGridVisible, ZSubgridVisible
AnimationStyle
AutoPlay
Frames
TimeBegin, TimeEnd, TimeRange, InitialTime
VisibleAfter, VisibleBefore, VisibleFromTo
VisibleBeforeBegin, VisibleAfterEnd
Footer, Header
FooterAlignment, HeaderAlignment
HorizontalAlignment, TitleAlignment, VerticalAlignment
Legend
LegendEntry
LegendAlignment, LegendPlacement, LegendVisible
LegendText
ShowInfo
Title, Titles
TitlePosition, TitlePositionX, TitlePositionY, TitlePositionZ
Bottom, Left
Height, Width
Layout, Rows, Columns
Margin, BottomMargin, TopMargin, LeftMargin, RightMargin
OutputUnits
Spacing
AbsoluteError, RelativeError
AdaptiveMesh
DiscontinuitySearch
Mesh, Submesh
MinimumDistance
ODEMethod, Stepsize
UMesh, VMesh, USubmesh, VSubmesh
XMesh, XSubmesh, YMesh, YSubmesh, ZMesh

CameraCoordinates
CameraDirection, CameraDirectionX, CameraDirectionY, CameraDirectionZ
FocalPoint, FocalPointX, FocalPointY, FocalPointZ
LightColor
Lighting
LightIntensity
OrthogonalProjection
SpotAngle
Target, TargetX, TargetY, TargetZ
UpVector, UpVectorX, UpVectorY, UpVectorZ, KeepUpVector
ViewingAngle
AntiAliased
ArrowLength
AxesTitleFont, FooterFont, HeaderFont, LegendFont, TextFont, TicksLabelFont, TitleFont
BackgroundColor, BackgroundColor2
BackgroundStyle
BackgroundTransparent
Billboarding
BorderColor, BorderWidth
BoxCenters, BoxWidths
DrawMode
Gap, XGap, YGap
Notched, NotchWidth
Projectors
Scaling, YXRatio, ZXRatio
VerticalAsymptotesVisible, VerticalAsymptotesStyle, VerticalAsymptotesColor, VerticalAsymptotesWidth
LineColor, LineColor2
LineColorDirection, LineColorDirectionX, LineColorDirectionY, LineColorDirectionZ
LineColorType
LineStyle
LinesVisible, ULinesVisible, VLinesVisible, XLinesVisible, YLinesVisible
LineWidth
MeshVisible
XContours, YContours, ZContours
PointColor
PointColor2
PointColorType
PointSize

PointStyle
PointsVisible
BarCenters, BarWidths
BarStyle, Shadows
Color
Colors
FillColor, FillColor2
FillColorDirection, FillColorDirectionX, FillColorDirectionY, FillColorDirectionZ
FillColorTrue, FillColorFalse, FillColorUnknown
FillColorType
Filled
FillPattern, FillPatterns
FillStyle
GroupStyle
InterpolationStyle
Shading
UseNormals
TipAngle
TipLength
TipStyle
TubeDiameter
Tubular

RGB::ColorNames

Find predefined colors by name

Syntax

```
RGB::ColorNames()

RGB::ColorNames(subname)
```

Description

`RGB::ColorNames(str)` returns a list of names of colors in the `RGB` name space whose names contain str.

`RGB::ColorNames` goes through the list of predefined color names and returns those whose names contain the string or identifier given as input, if any.

Environment Interactions

`RGB::plotColorPalette` plots a list of color samples with names.

Examples

Example 1

The following call returns all predefined color names containing "`Olive`":

```
RGB::ColorNames("Olive")
```

```
[Olive, OliveDrab, OliveGreen, OliveGreenDark]
```

The RGB values of these colors are:

```
RGB::Olive, RGB::OliveDrab, RGB::OliveGreen,
RGB::OliveGreenDark
```

```
[0.230003, 0.370006, 0.170003], [0.419599, 0.556902, 0.137303], [0.2, 0.2, 0.0],
  [0.333293, 0.419599, 0.184301]
```

Example 2

The following call plots all predefined colors containing "`Olive`":

```
RGB::plotColorPalette("Olive")
```

When the list of colors found gets larger, they are distributed over more lines:

```
RGB::plotColorPalette("Blue")
```


Colors containing 'Blue'
AliceBlue
AirForceBlue
BlueMedium
BlueLight
BlueGrey
Blue
AzureBlueDark
CornflowerBlue
CeruleanBlue
BondiBlue
BlueViolet
BluePale
ManganeseBlue
LightBlue
InternationalKleinBlue
DodgerBlue
DarkBlue
PersianBlue
PaleCornflowerBlue
PaleBlue
NavyBlue
MidnightBlue
RoyalBlue
RobinEggBlue
PrussianBlue
PowderBlueLight
PowderBlue
SlateBlueDark
SlateBlue
SkyBlueLight
SkyBlueDeep
SkyBlue
UnitedNationsBlue
SteelBlueLight
SteelBlue
SlateBlueMedium
SlateBlueLight

Parameters

subname

A part of a color name: a string or an identifier

Return Values

`RGB::ColorNames` returns a list of predefined color names. `RGB::plotColorPalette` returns the empty object, `null()`.

See Also

MuPAD Functions
`RGB::plotColorPalette`

RGB::plotColorPalette

Display predefined colors

Syntax

```
RGB::plotColorPalette(subname)
```

Description

`RGB::plotColorPalette(str)` displays the colors in the `RGB` name space whose names contain str.

`RGB::plotColorPalette` uses `RGB::ColorNames` and plots samples of the colors found by this routine, in tabular fashion.

Environment Interactions

`RGB::plotColorPalette` plots a list of color samples with names.

Examples

Example 1

The following call returns all predefined color names containing "`Olive`":

```
RGB::ColorNames("Olive")
```

```
[Olive, OliveDrab, OliveGreen, OliveGreenDark]
```

The RGB values of these colors are:

```
RGB::Olive, RGB::OliveDrab, RGB::OliveGreen,
RGB::OliveGreenDark
```

```
[0.230003, 0.370006, 0.170003], [0.419599, 0.556902, 0.137303], [0.2, 0.2, 0.0],
  [0.333293, 0.419599, 0.184301]
```

Example 2

The following call plots all predefined colors containing "`Olive`":

```
RGB::plotColorPalette("Olive")
```

When the list of colors found gets larger, they are distributed over more lines:

```
RGB::plotColorPalette("Blue")
```


Colors containing 'Blue'
AliceBlue
AirForceBlue
BlueMedium
BlueLight
BlueGrey
Blue
AzureBlueDark
CornflowerBlue
CeruleanBlue
BondiBlue
BlueViolet
BluePale
ManganeseBlue
LightBlue
InternationalKleinBlue
DodgerBlue
DarkBlue
PersianBlue
PaleCornflowerBlue
PaleBlue
NavyBlue
MidnightBlue
RoyalBlue
RobinEggBlue
PrussianBlue
PowderBlueLight
PowderBlue
SlateBlueDark
SlateBlue
SkyBlueLight
SkyBlueDeep
SkyBlue
UnitedNationsBlue
SteelBlueLight
SteelBlue
SlateBlueMedium
SlateBlueLight

Parameters

subname

A part of a color name: a string or an identifier

Return Values

`RGB::ColorNames` returns a list of predefined color names. `RGB::plotColorPalette` returns the empty object, `null()`.

See Also

MuPAD Functions
`RGB::ColorNames`

RGB::colorName

Find names of predefined colors

Syntax

```
RGB::colorName(rgb, <Exact>)
```

Description

`RGB::colorName([r, g, b])` looks for the predefined color with values closest to `[r, g, b]` and returns its name.

`RGB::colorName([r, g, b], Exact)` looks for a predefined color with values exactly `[r, g, b]` and returns its name.

The `RGB` namespace contains predefined color names, accessible as `RGB::Blue` etc. `RGB::colorName` performs a reverse lookup, finding the name of a color given as RGB values.

Since rather often, colors will stem from calculations with floating-point numbers, no exact matches can be expected in this reverse lookup. Therefore, by default, `RGB::colorName` will perform a "fuzzy" search, returning the predefined color which is closest (in Euclidean distance in RGB space) to the input. Cf. "Example 2" on page 24-18.

Examples

Example 1

`RGB::colorName` returns the symbolic name of predefined colors:

```
RGB::colorName([0, 1, 0])
```

```
RGB::Green
```

```
RGB::colorName([0, 1, 0, 0.5])
```

```
RGB::Green.[0.5]
```

Example 2

When performing calculations on color values, the results will rarely be exact, even if the unavoidable round-off errors are too small to be displayed on the screen:

```
a := RGB::Olive;
b := RGB::fromHSV(RGB::toHSV(RGB::Olive))
```

```
[0.230003, 0.370006, 0.170003]
```

```
[0.230003, 0.370006, 0.170003]
```

```
bool(a = b)
```

```
FALSE
```

Therefore, `RGB::colorName` by default searches in a “fuzzy” fashion:

```
RGB::colorName(a);
RGB::colorName(b)
```

```
RGB::Olive
```

```
RGB::Olive
```

In cases where this is undesirable, the option `Exact` can be used to switch to exact searching:

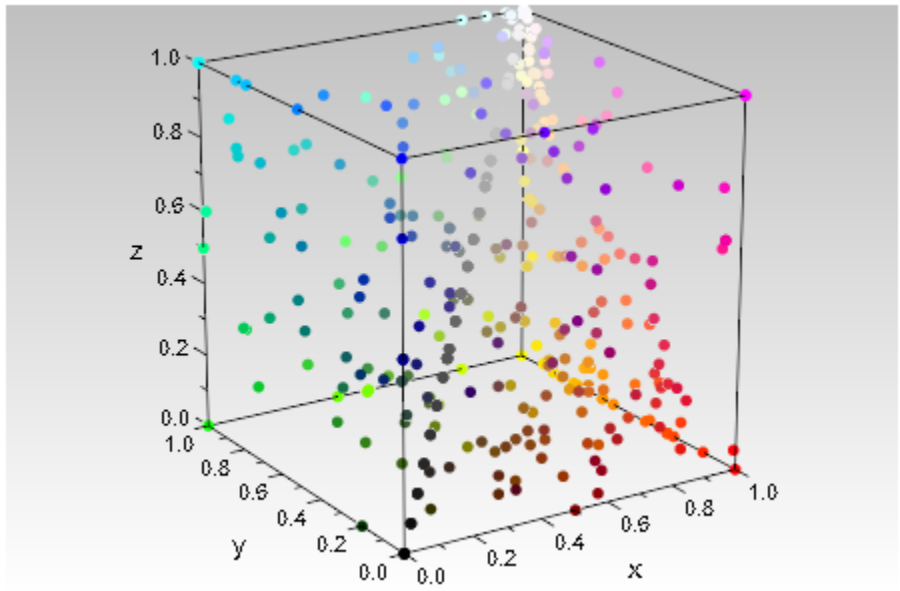
```
RGB::colorName(a, Exact);
RGB::colorName(b, Exact)
```

```
RGB::Olive
```

FAIL

The predefined color names do not fill RGB space uniformly, therefore, the color found by `RGB::colorName` may be quite different from the one entered. The following plot shows the predefined colors in RGB space:

```
plot(plot::Scene3d(
  plot::PointList3d([c.[c] $ c in RGB::ColorList]),
  ZXRatio = 1, BackgroundStyle = TopBottom,
  BackgroundColor = RGB::Grey,
  BackgroundColor2 = RGB::White,
  Margin=0))
```



Parameters

`rgb`

An RGB or RGBa color specification: A list of three or four real numbers in the interval [0, 1].

Options

`Exact`

Only return an exact match, FAIL if none exists.

Return Values

If a color was found, `RGB::colorName` returns an expression of the form `RGB::Name` or `RGB::Name.[a]`. If given `Exact` and no match was found, `FAIL` is returned. If given symbolic input parameters, an unevaluated call is returned.

See Also

MuPAD Functions
RGB

RGB::fromWaveLength

Get the RGB color of monochromatic light

Syntax

```
RGB::fromWaveLength(λ, <γ>)
```

Description

`RGB::fromWaveLength( λ )` returns an approximative RGB specification for light of wavelength $\lambda\, nm$.

Light consists of photons, each of which has a distinct wavelength. These different wavelengths cause color perception. `RGB::fromWaveLength` calculates an RGB triple corresponding to a given wave length.

Different displays show the same RGB color in slightly different ways. For this reason, the so-called "gamma correction" has been invented. `RGB::fromWaveLength` accepts a second argument, for fine-tuning the assumed gamma correction that enters the calculation.

Color perception depends on a number of factors, including individual differences. Therefore, such a calculation can only return an approximation. `RGB::fromWaveLength` uses the model published by Dan Bruton for the conversion.

For wavelengths outside the visible spectrum (which ranges from 380 nm to 780 nm), `RGB::fromWaveLength` returns black.

Examples

Example 1

White light, when sent through a prism, is split into the commonly known spectrum, because the prism refracts different wavelengths differently. This spectrum can easily be reproduced by `RGB::fromWaveLength`:

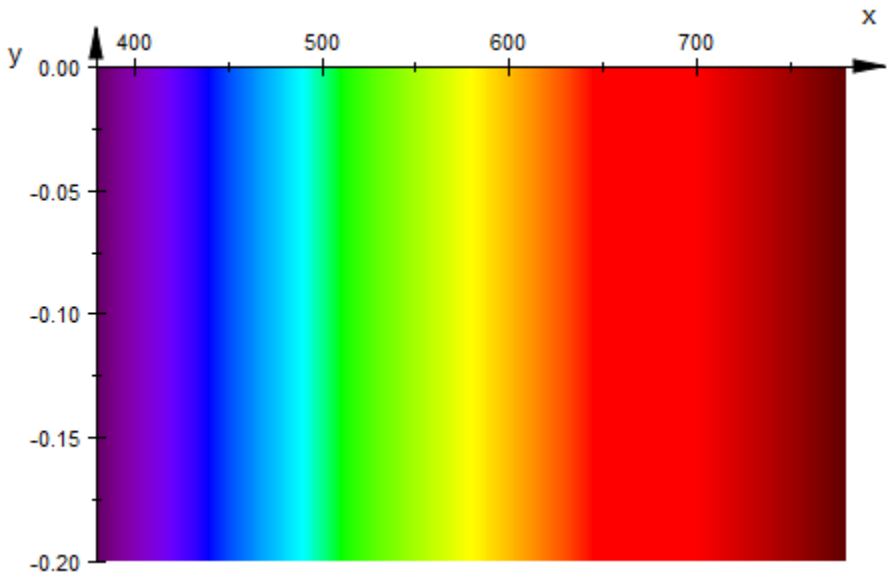
```
plot(plot::Raster([[RGB::fromWaveLength(i) $ i=380..780]]),
```

```
  Scaling = Unconstrained, Height = 20)
```

Example 2

Bruton's conversion model looks like this:

```
plotfunc2d(
  plot::Raster([[RGB::fromWaveLength(i) $ i = 380..780]],
    x = 380..780, y = -0.2..0),
  (x -> RGB::fromWaveLength(x)[i]) $ i = 1..3,
  x = 380..780,
  Colors = [RGB::Red, RGB::Green, RGB::Blue],
  LegendVisible = FALSE,
  XTicksNumber = Low,
  Scaling = Unconstrained,
  Axes = Automatic)
```



Parameters

λ

The wavelength: a real-valued constant (interpreted as nanometers) or a length expression

γ

The “gamma correcture” for the display, defaults to 0.8

Return Values

RGB color: a list of three floating-point values

See Also

MuPAD Functions
`RGB` | `RGB::fromHSV`

RGB::random

Pick a color uniformly at random in RGB color space

Syntax

```
RGB::random()
```

Description

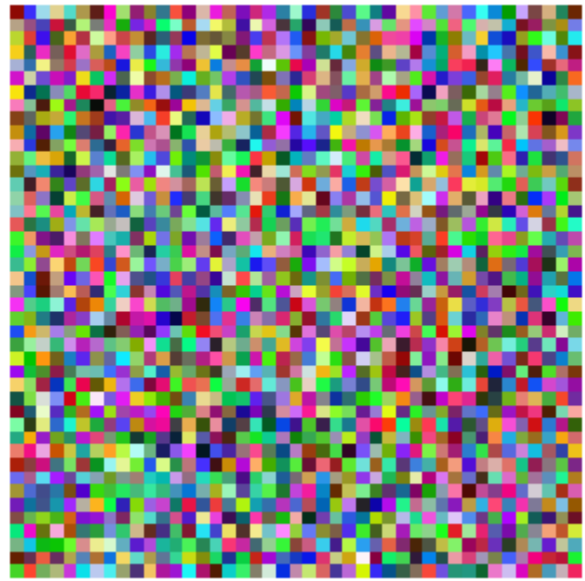
`RGB::random()` returns a random color. The colors returned are distributed independently and uniformly in the RGB color space.

Examples

Example 1

`RGB::random` can be used to produce high-frequency noise:

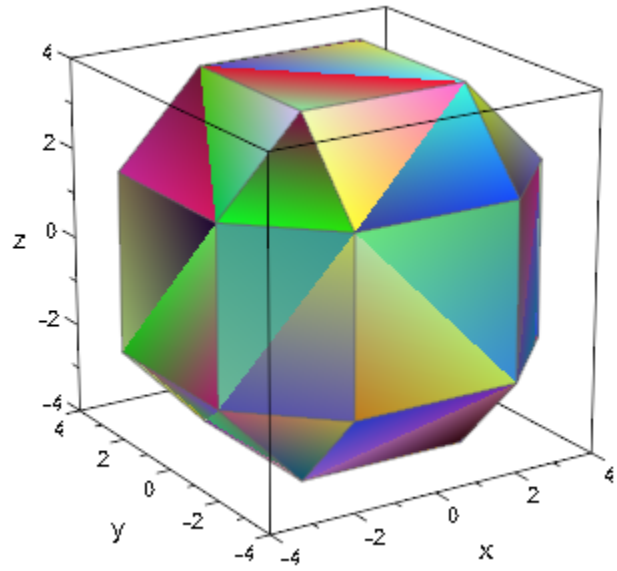
```
plot(plot::Raster([[RGB::random() $ x = 0..42] $ y = 0..42]))
```



Example 2

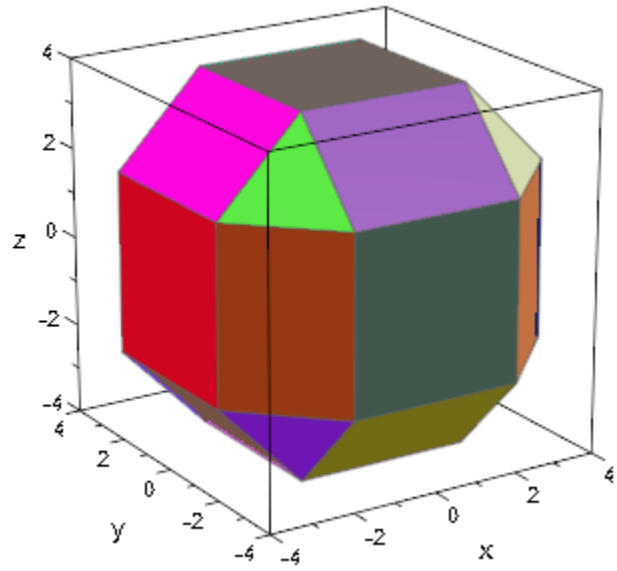
It is possible to use `RGB::random` directly as a color function, but the result may be unexpected:

```
plot(plot::Waterman(5, FillColorFunction = RGB::random))
```



The reason is that the color function will be called repeatedly if the same point is met again. It is a better idea to create a bunch of random colors and then use the parameters passed into the color functions to get some consistency into the choice of colors:

```
colors := [RGB::random() $ i = 1..42]:
plot(plot::Waterman(5, FillColorFunction=((x,y,z,i) -> colors[i])))
```



Return Values

RGB color: A list of three floating-point values.

See Also

MuPAD Functions
`frandom | RGB`

RGB::toHSV

Convert RGB colors to HSV

Syntax

```
RGB::toHSV([r, g, b, <a>])
```

Description

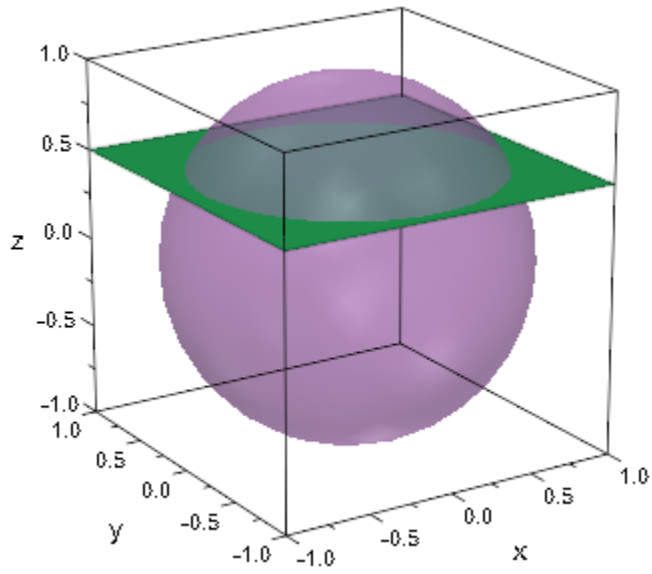
`RGB::toHSV(RGBcolor)` returns `RGBcolor` expressed in HSV values, with hue in the range `0..360` (i.e., in degrees) and saturation and value in the range `0..1`.

Examples

Example 1

With the `RGB::fromHSV` utility, all colors in a MuPAD graphics can be specified easily as HSV colors. For example, the color "violet" is given by the HSV values `[290, 0.4, 0.6]`, whereas "dark green" is given by the HSV specification `[120, 1, 0.4]`. Hence, a semi-transparent violet sphere intersected by an opaque dark green plain may be specified as follows:

```
plot(plot::Sphere(1, [0, 0, 0],
                  Color = RGB::fromHSV([290, 0.4, 0.6]).[0.5]),
     plot::Surface([x, y, 0.5], x = -1 .. 1, y = -1 .. 1,
                   Mesh = [2, 2],
                   Color = RGB::fromHSV([120, 1, 0.4]))
):
```



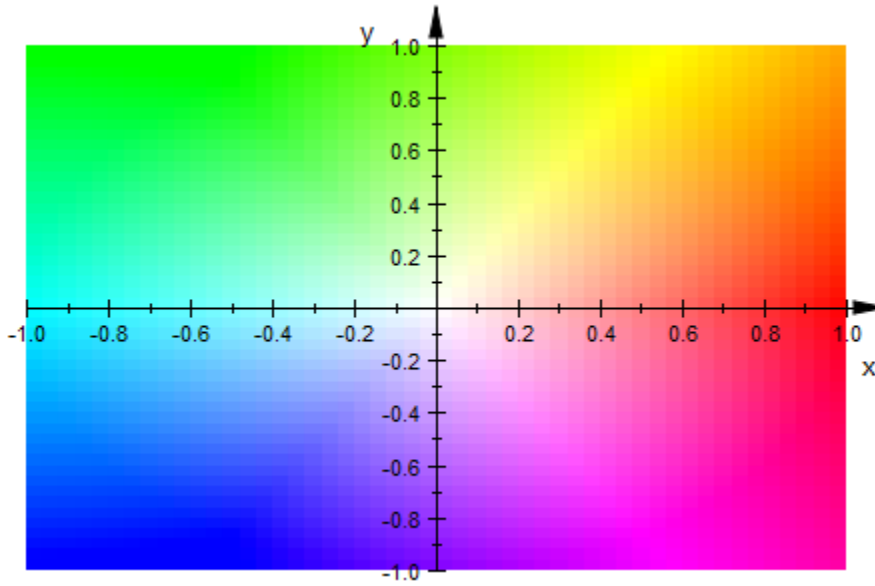
Example 2

There are numerous ways of displaying complex-valued functions of a complex argument, see for example `plot::Conformal`. One of these is to use a color scheme that interprets the complex plane as a section through HSV color space at a fixed value, say, 1. To plot this scheme in MuPAD, we use `plot::Density`, providing the following color function:

```
f_color := (x, y, fz, a) ->
           RGB::fromHSV([180/float(PI)*arg(fz), abs(fz), 1]):
```

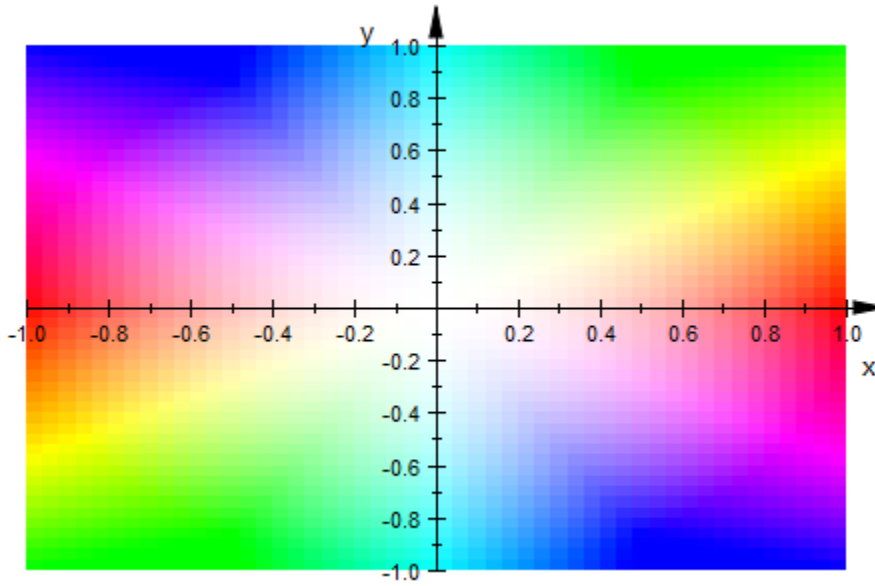
The identity function is thus shown as follows:

```
plot(plot::Density((x, y) -> x + I*y, x = -1..1, y = -1..1,
                   XMesh = 50, YMesh = 50,
                   FillColorFunction = f_color))
```



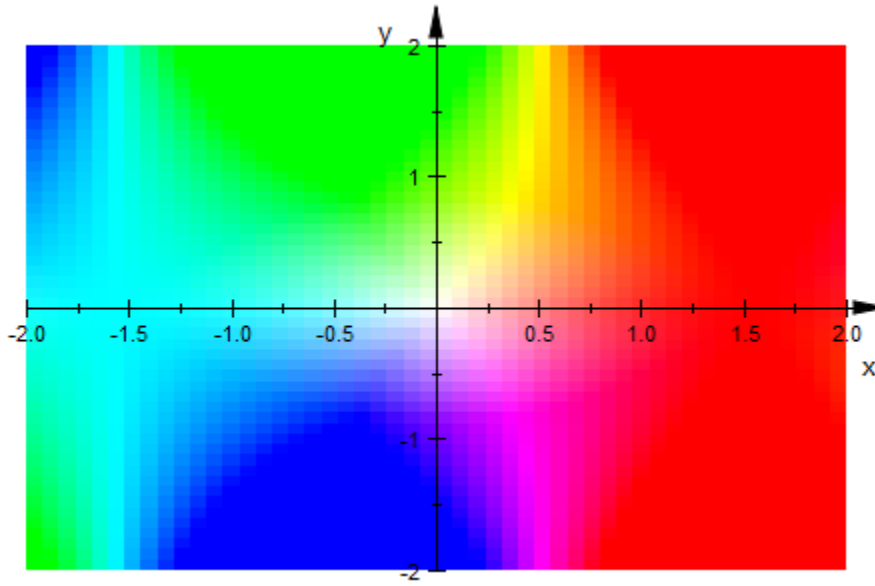
$z \to z^2$ doubles the argument of a complex function, resulting in the following picture:

```
plot(plot::Density((x, y) -> (x + I*y)^2, x = -1..1, y = -1..1,
                   XMesh = 50, YMesh = 50,
                   FillColorFunction = f_color))
```



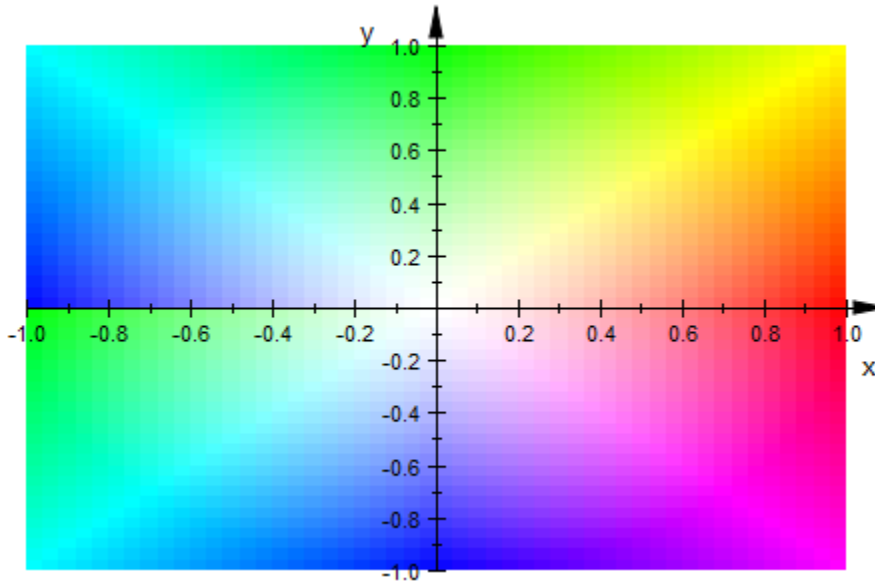
To plot the complex sine function, we choose a larger rectangle, since the sine is too similar to the identity in small neighborhoods of th origin to be of interest:

```
plot(plot::Density((x, y) -> sin(x + I*y), x = -2..2, y = -2..2,
                   XMesh = 50, YMesh = 50,
                   FillColorFunction = f_color))
```



$z \to z^{4/3}$ is clearly discontinuous along the negative real axis:

```
plot(plot::Density((x, y) -> (x + I*y)^(4/3),
                   x = -1..1, y = -1..1,
                   XMesh = 50, YMesh = 50,
                   FillColorFunction = f_color))
```



Parameters

r, g, b

The red, green, and blue contributions of an RGB color: numerical values between 0 and 1.

a

The translucency (alpha) value: a numerical value between 0 and 1.

Return Values

a list with three or four floating-point values, depending on whether a was given in the input.

See Also

MuPAD Functions
`RGB::fromHSV`

RGB::fromHSV

Convert HSV colors to RGB

Syntax

```
RGB::fromHSV([h, s, v, <a>])
```

Description

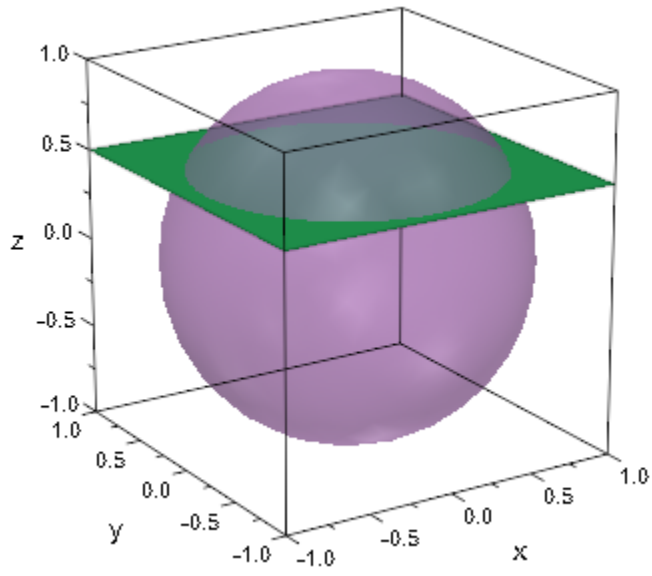
`RGB::fromHSV(HSVcolor)` is the inverse of `RGB::toHSV`: Given color coordinates in HSV, this function returns the corresponding RGB color. Cf. “Example 2” on page 24-36.

Examples

Example 1

With the `RGB::fromHSV` utility, all colors in a MuPAD graphics can be specified easily as HSV colors. For example, the color “violet” is given by the HSV values `[290, 0.4, 0.6]`, whereas “dark green” is given by the HSV specification `[120, 1, 0.4]`. Hence, a semi-transparent violet sphere intersected by an opaque dark green plain may be specified as follows:

```
plot(plot::Sphere(1, [0, 0, 0],
                  Color = RGB::fromHSV([290, 0.4, 0.6]).[0.5]),
     plot::Surface([x, y, 0.5], x = -1 .. 1, y = -1 .. 1,
                   Mesh = [2, 2],
                   Color = RGB::fromHSV([120, 1, 0.4]))
):
```



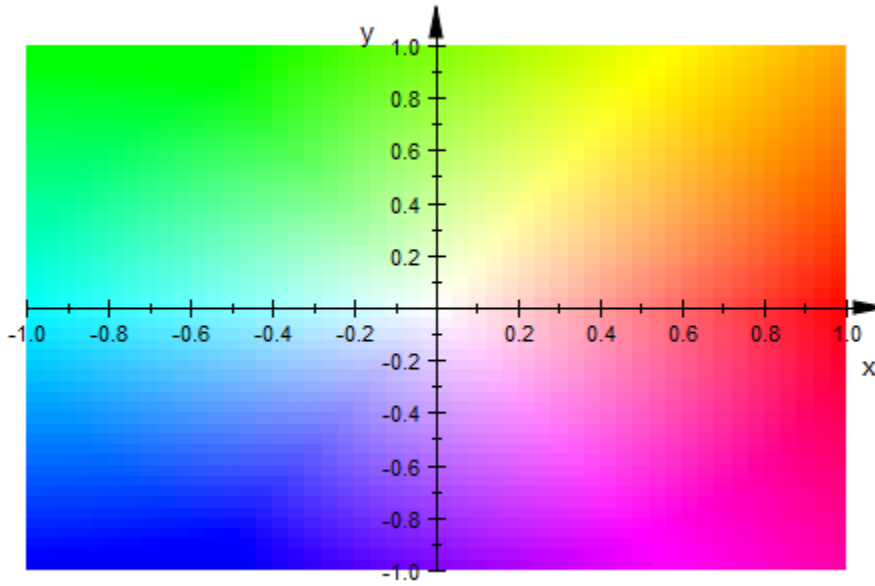
Example 2

There are numerous ways of displaying complex-valued functions of a complex argument, see for example `plot::Conformal`. One of these is to use a color scheme that interprets the complex plane as a section through HSV color space at a fixed value, say, 1. To plot this scheme in MuPAD, we use `plot::Density`, providing the following color function:

```
f_color := (x, y, fz, a) ->
           RGB::fromHSV([180/float(PI)*arg(fz), abs(fz), 1]):
```

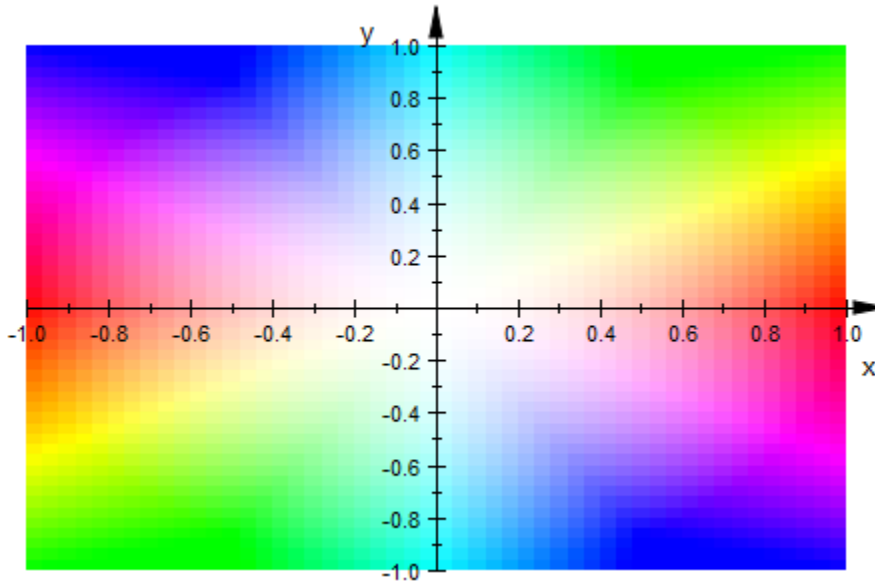
The identity function is thus shown as follows:

```
plot(plot::Density((x, y) -> x + I*y, x = -1..1, y = -1..1,
                   XMesh = 50, YMesh = 50,
                   FillColorFunction = f_color))
```



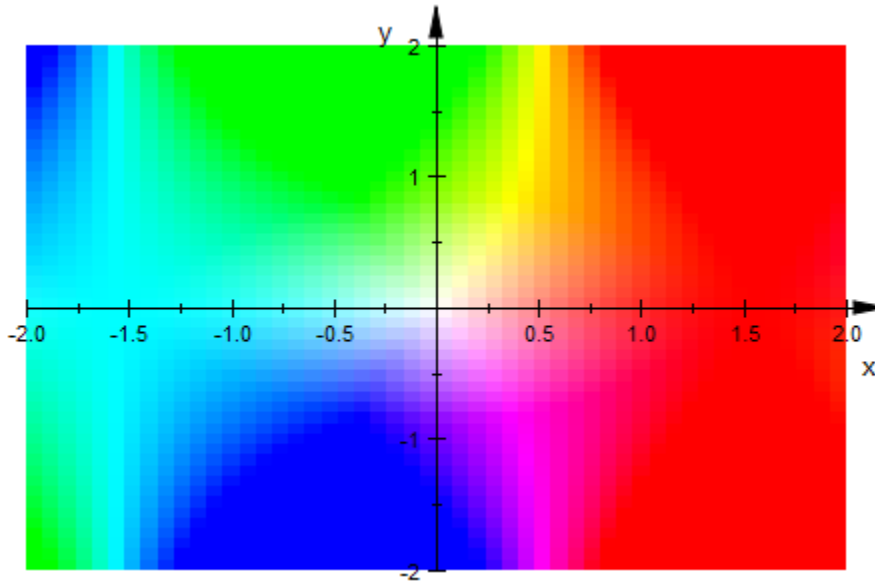
$z \to z^2$ doubles the argument of a complex function, resulting in the following picture:

```
plot(plot::Density((x, y) -> (x + I*y)^2, x = -1..1, y = -1..1,
                   XMesh = 50, YMesh = 50,
                   FillColorFunction = f_color))
```



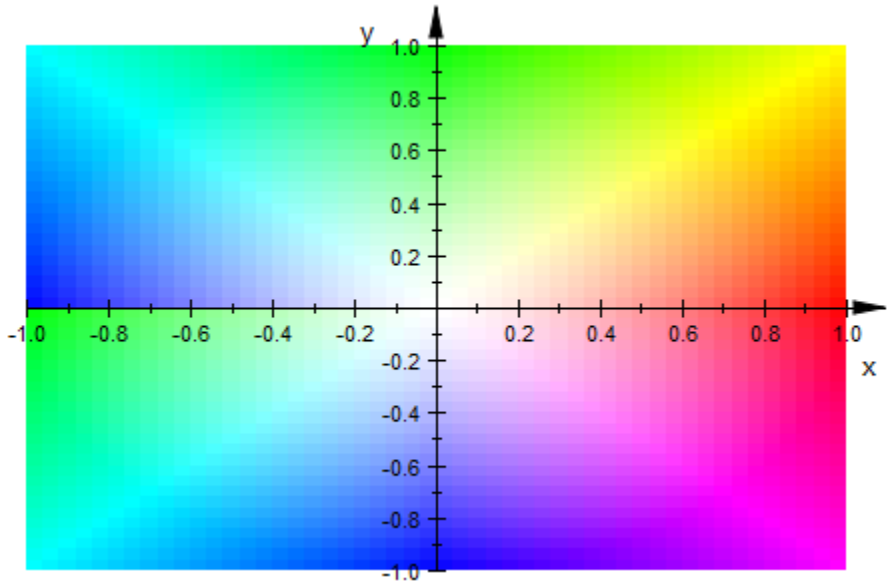
To plot the complex sine function, we choose a larger rectangle, since the sine is too similar to the identity in small neighborhoods of th origin to be of interest:

```
plot(plot::Density((x, y) -> sin(x + I*y), x = -2..2, y = -2..2,
                   XMesh = 50, YMesh = 50,
                   FillColorFunction = f_color))
```



$z \to z^{4/3}$ is clearly discontinuous along the negative real axis:

```
plot(plot::Density((x, y) -> (x + I*y)^(4/3),
                   x = -1..1, y = -1..1,
                   XMesh = 50, YMesh = 50,
                   FillColorFunction = f_color))
```



Parameters

a

The translucency (alpha) value: a numerical value between 0 and 1.

h

The “hue” in an HSV specification: a numerical value between 0 and 360

s

The “saturation” in an HSV specification: a numerical value between 0 and 1

v

The “value” in an HSV specification: a numerical value between 0 and 1

Return Values

a list with three or four floating-point values, depending on whether a was given in the input.

See Also

MuPAD Functions
`RGB::toHSV`

plot::easy

Easy plotting

Syntax

```
plot::easy(<arg, …>, options)
```

Description

`plot::easy( arg , <options>, …)` transforms data and expressions into graphical objects.

`plot::easy` accepts graphical objects and graphical attributes as input and returns them unchanged.

`plot::easy` supports the options listed above. Additionally, it accepts arbitrary data and expressions and tries to transform them into valid graphical objects.

`plot::easy` supports the option `Colors`=[c_1,...c_2] for automatically coloring newly generated graphical objects. The given list is used instead of the internally defined default color list.

`plot::easy` accepts the options `Mesh` and `Submesh` and uses them for each newly generated graphical object.

The function `plot` calls `plot::easy` for preprocessing its input before plotting.

`plot::easy` tries to handle standard situations intuitively in order to make plotting as easy as possible. However, it supports only a small subset of the graphical objects, attributes and expressions available in MuPAD and thus does not claim to be complete.

Users that want to plot other objects or control specific details of their graphics explicitly, still have to create their graphical objects manually, e.g. using `plot::Function2d` and `plot::Point2d`, and to use graphical attributes like `LineStyle =Dashed` directly.

`plot::easy` sets a new color for each object that it creates, if no color is predefined in the given context.

`plot::easy` accepts sets {...} as group definition and transforms them into graphical objects of type `plot::Group2d(...)` or `plot::Group3d(...)`, respectively. All elements of a group share the same color, if colors were not specified explicitly for single objects.Note that, regular graphical objects usually have a predefined color.

`plot::easy` accepts a nested set {{...}} as scene definition and transforms it into the graphical object `plot::Scene2d(...)` or `plot::Scene3d(...)`, respectively.

Creating a graphical object may require the specification of value ranges for variables. If they are not specified explicitly then `plot::easy` tries use ranges specified for other variables and/or uses the default value range `-5..5`.

Examples

Example 1

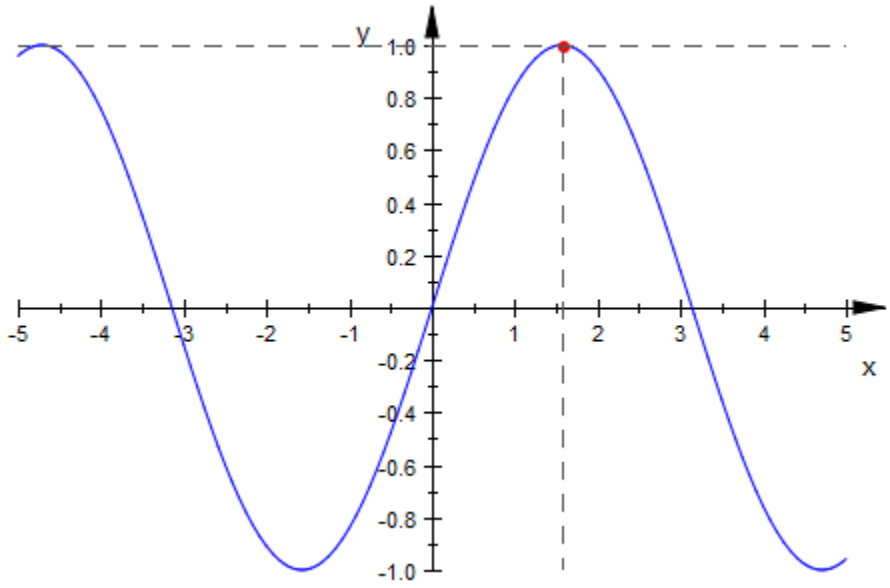
`plot::easy` tries to transform all given data and expressions into valid graphical objects and attributes:

```
plot::easy(sin(x), [PI/2,1])
```

$$\text{plot::Function2d}(\sin(x),\ x = -5..5),\ \text{plot::Point2d}\left(\frac{\pi}{2},\ 1,\ \text{PointColor} = \text{RGB::Red},\ \text{LegendText} = \text{"[PI/2, 1]"}\right)$$

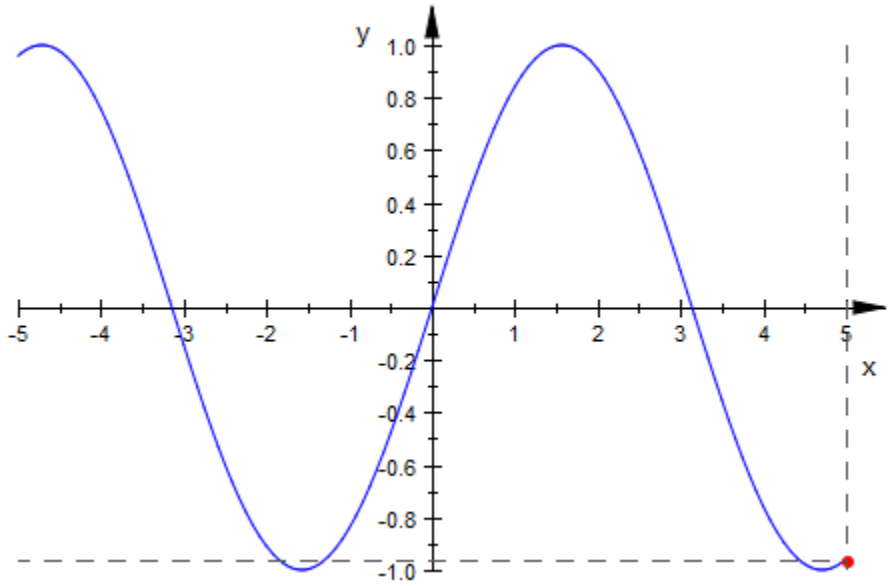
Since the function `plot` calls the function `plot::easy` for preprocessing its input data, scenes like above can directly be plotted using `plot`:

```
plot(sin(x), [PI/2,1], #x = PI/2, #y = 1)
```



Now, it is only a small step to animate this scene:

```
plot(sin(x), {[x,sin(x)], #Points}, #x = x, #y = sin(x))
```



Note: Graphical objects and attributes, as well as data that `plot::easy` cannot transform, are returned unchanged:

```
plot::easy(x, plot::Point2d(1,1), LineStyle=Dashed, "UnknownObject")
```

plot::Function2d(x, x = −5..5), plot::Point2d(1, 1), LineStyle = Dashed, "UnknownObject"

This is why `plot` returns the following error message when it is executed with the above arguments:

```
plot(x, plot::Point2d(1,1), LineStyle=Dashed, "UnknownObject");
```

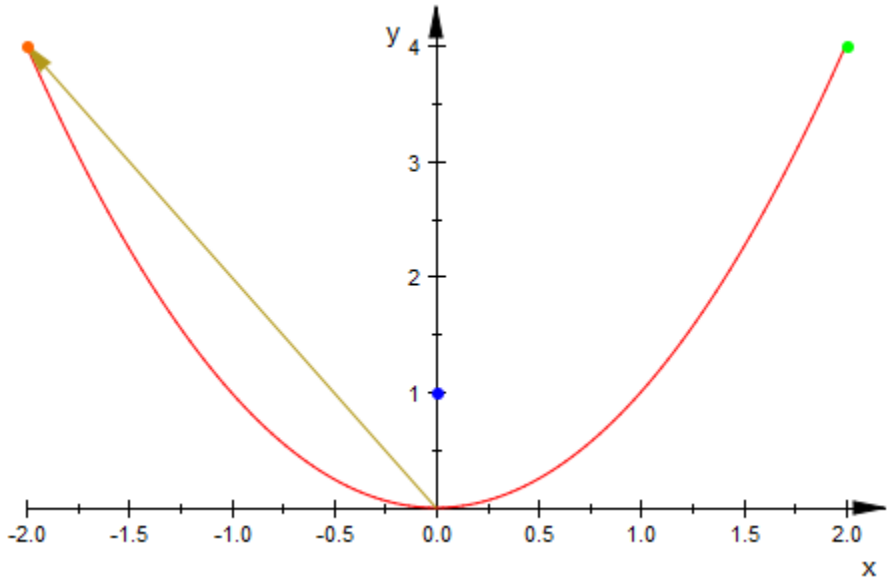
```
Error: The arguments 'UnknownObject' are unexpected. [plot::Canvas::new]
```

Example 2

Points can be entered as lists with two or three values. Alternatively, a corresponding column vector in combination with the option `#Points` (alias `#P`, see “Example 18” on page 24-75) can be used.

Note that for plotting animated points the option `#Points` is required. Otherwise a curve (when entered a list) or an arrow (when entered a vector) is plotted:

```
plot([0,1], [s,s^2], {[s,s^2], #Points}, s = -2.. 2,
     matrix([t,t^2]), {matrix([t,t^2]), #Points}, t = 2..-2
)
```

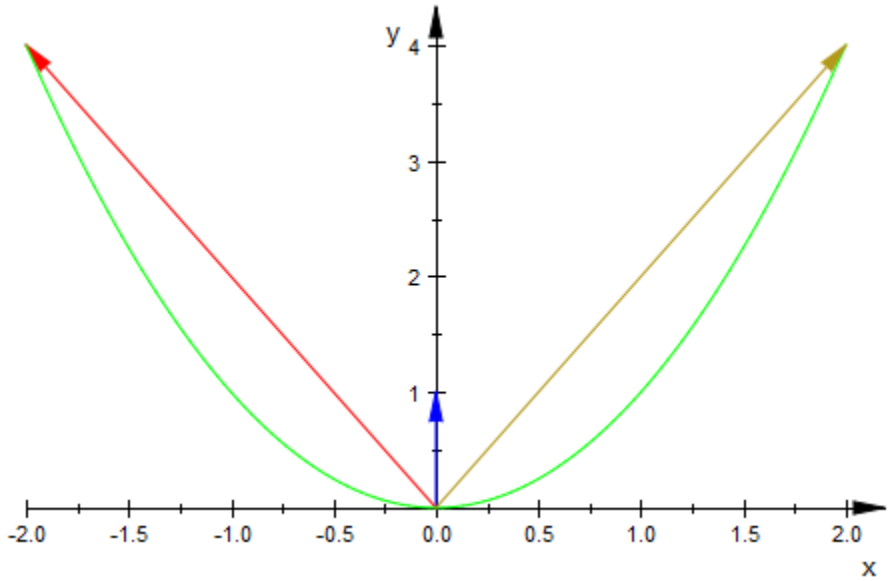


See also: `plot::Point2d`, `plot::Point3d`.

Example 3

Arrows can be specified as column vectors with two or three elements. Alternatively, a list in combination with the option `#Arrows` (alias `#A`, see “Example 13” on page 24-70) can be used:

```
plot(matrix([0,1]), matrix([s,s^2]), s = 2..-2,
     [t,t^2], {[t,t^2], #Arrows}, t = -2..2
)
```

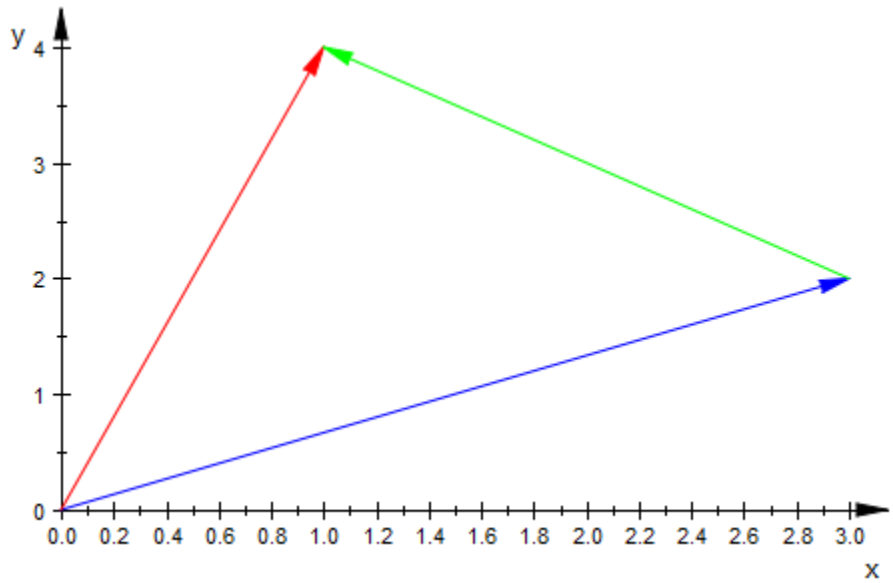


If an arrow should start at coordinates other than (0,0) or (0,0,0), respectively, then a list of two column vectors or a corresponding list of lists in combination with the option `#Arrows` (alias `#A`, see "Example 13" on page 24-70) can be used:

```
u := matrix([3,2]):
v := matrix([1,4]):
w := (1-a)*u + a*v:
u, v, w;
```

$$\begin{pmatrix} 3 \\ 2 \end{pmatrix}, \begin{pmatrix} 1 \\ 4 \end{pmatrix}, \begin{pmatrix} 3-2\,a \\ 2\,a+2 \end{pmatrix}$$

```
plot(u, v, [u,w], a = 0.1..1, #Arrows)
```



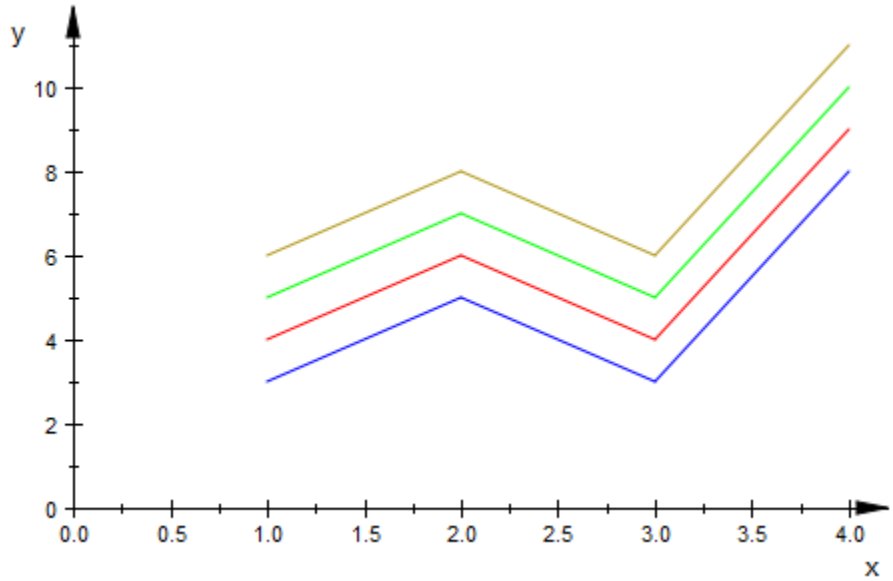

```
delete u, v, w:
```

See also: `plot::Arrow2d`, `plot::Arrow3d`.

Example 4

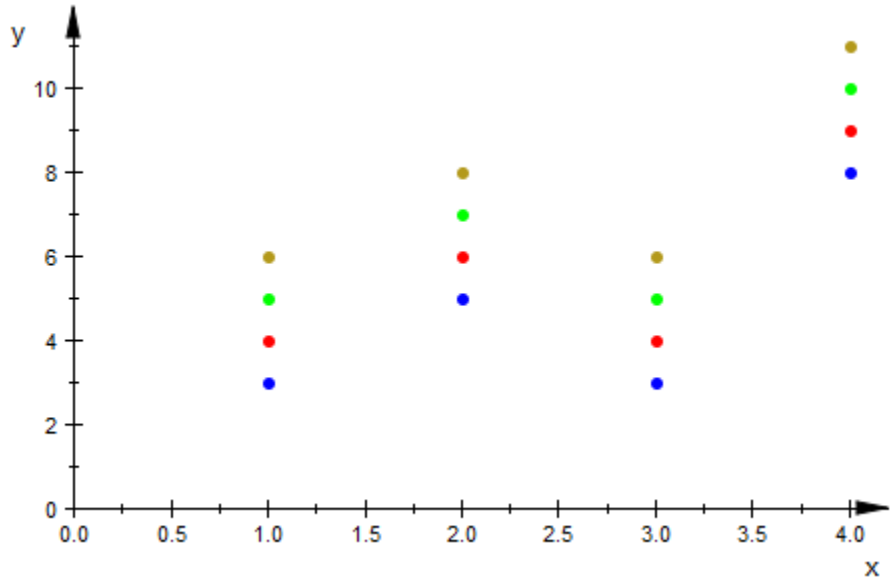
Polygons can be specified as lists, tables or matrices. In the following example, polygons are plotted using different input styles. The option `#Origin` (alias `#0`, see "Example 17" on page 24-74) ensures that the origin of the coordinate system is visible in the scene as well:

```
plot([[1,3],[2,5],[3,3],[4,8]],
     table(1=4,2=6,3=4,4=9),
     [matrix([1,5]),matrix([2,7]),matrix([3,5]),matrix([4,10])],
     matrix([[1,6],[2,8],[3,6],[4,11]]),
     #Origin)
```



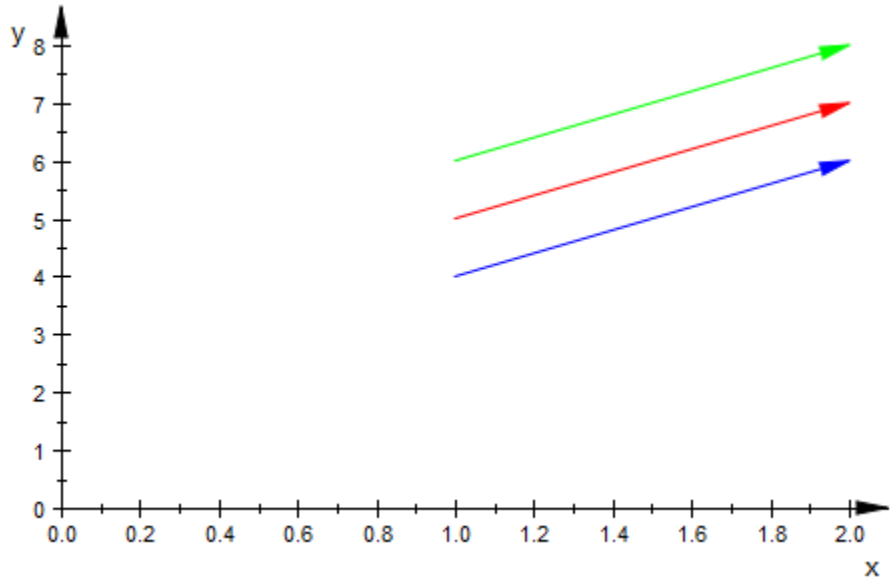
Note that polygons are displayed as points when option `#Points` (alias `#P`, see "Example 18" on page 24-75) is used:

```
plot([[1,3],[2,5],[3,3],[4,8]],
     table(1 = 4,2 = 6,3 = 4,4 = 9),
     [matrix([1,5]),matrix([2,7]),matrix([3,5]),matrix([4,10])],
     matrix([[1,6],[2,8],[3,6],[4,11]]),
     #Points, #Origin)
```



Note that the following polygons with two elements are displayed as arrows when option `#Arrows` (alias `#A`, see "Example 13" on page 24-70) is used:

```
plot([[1,4],[2,6]],
     [matrix([1,5]),matrix([2,7])],
     matrix([[1,6],[2,8]]),
     #Arrows, #Origin)
```

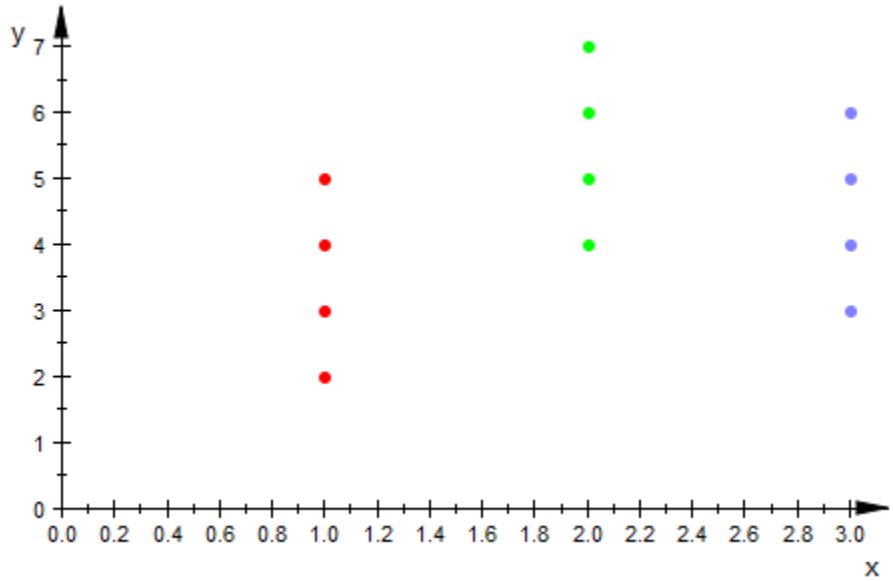


See also: `plot::Polygon2d`, `plot::Polygon3d`.

Example 5

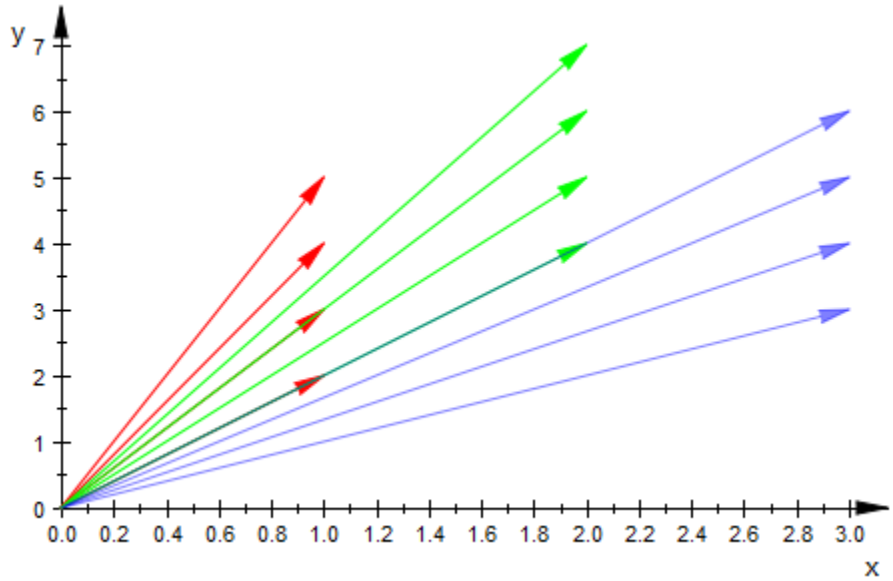
Point lists can be specified as lists, tables or matrices. For each point an RGBa color has to be specified. In the following example, point lists are plotted using different input styles. The option `#Origin` (alias `#0`, see "Example 17" on page 24-74) ensures that the origin of the coordinate system is visible in the scene as well:

```
plot([[1,2,RGB::Red], [2,4,[0,1,0]], [3,3,[0,0,1,0.5]]],
     [matrix([1,3,RGB::Red]), matrix([2,5,[0,1,0]]),
                              matrix([3,4,[0,0,1,0.5]])],
     matrix([[1,4,RGB::Red], [2,6,[0,1,0]],[3,5,[0,0,1,0.5]]]),
     table(1=[5,RGB::Red], 2=[7,[0,1,0]], 3=[6,[0,0,1,0.5]]),
     #Origin)
```



Note that the following point lists are displayed as arrows when option `#Arrows` (alias `#A`, see "Example 13" on page 24-70) is used:

```
plot([[1,2,RGB::Red], [2,4,[0,1,0]] ,[3,3,[0,0,1,0.5]]],
     [matrix([1,3,RGB::Red]), matrix([2,5,[0,1,0]]),
                              matrix([3,4,[0,0,1,0.5]])],
     matrix([[1,4,RGB::Red], [2,6,[0,1,0]], [3,5,[0,0,1,0.5]]]),
     table(1=[5,RGB::Red], 2=[7,[0,1,0]] ,3=[6,[0,0,1,0.5]]),
     #Arrows, #Origin)
```

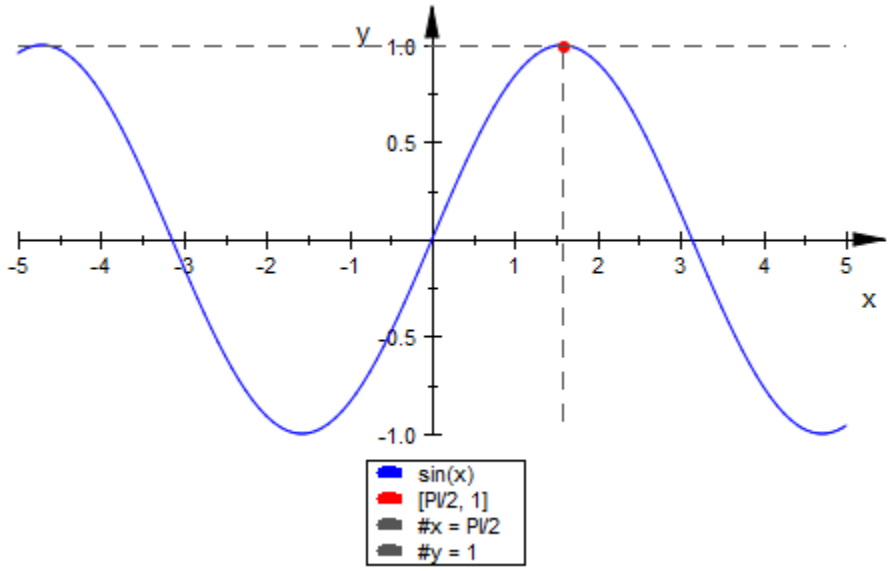


See also: `plot::PointList2d`, `plot::PointList2d`.

Example 6

For drawing horizontal and vertical infinite lines, the short syntax `#x = e` and `#y = e` with `e` is a real expression,can be used:

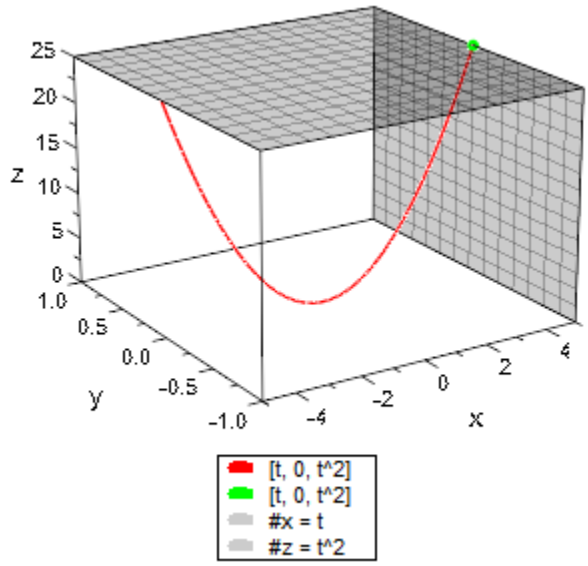
```
plot(sin(x), [PI/2,1], #x=PI/2, #y=1, #Legend)
```



For drawing horizontal and vertical infinite planes, the short syntax`#x = e`, `#y = e` and `#z = e`, with `e` is a real expression, can be used.

Both, lines and planes can also be animated:

```
plot([t,0,t^2], {[t,0,t^2], #Points}, #x=t, #z=t^2, #Legend)
```

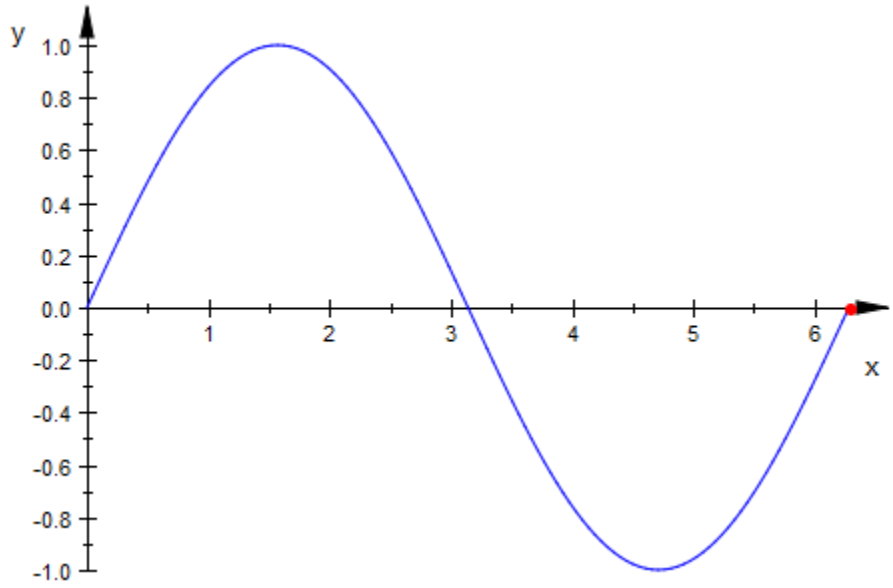


See also: `plot::Line2d`, `plot::Plane`.

Example 7

A curve can be specified as list with two or three elements, where at least one element depends on a free variable. If option `#Points` (alias `#P`, see “Example 18” on page 24-75) is set, then instead of a curve, an animated point is plotted that moves along the curve.

```
plot([t,sin(t)], {[t,sin(t)], #Points},  t=0..2*PI)
```

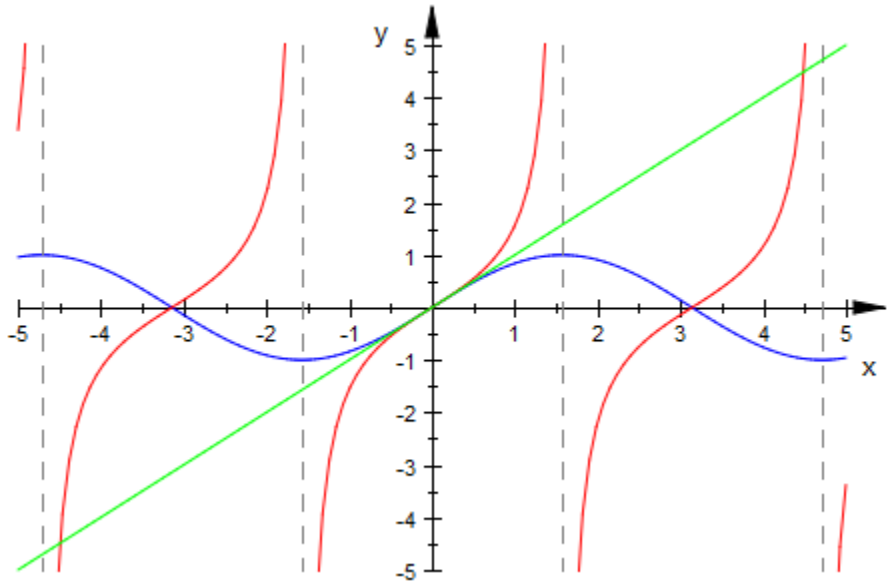


See also: `plot::Curve2d`, `plot::Curve3d`.

Example 8

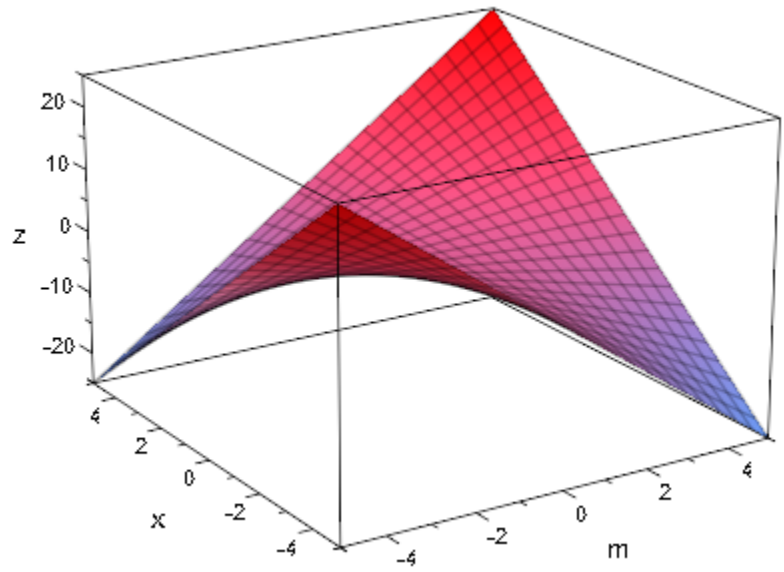
`plot::easy` tries to transform expressions that are no lists, sets, matrices, equations or inequalities into graphs of 2D or 3D functions. We plot some graphs of 2D functions:

```
plot(sin(x), tan(x), x)
```



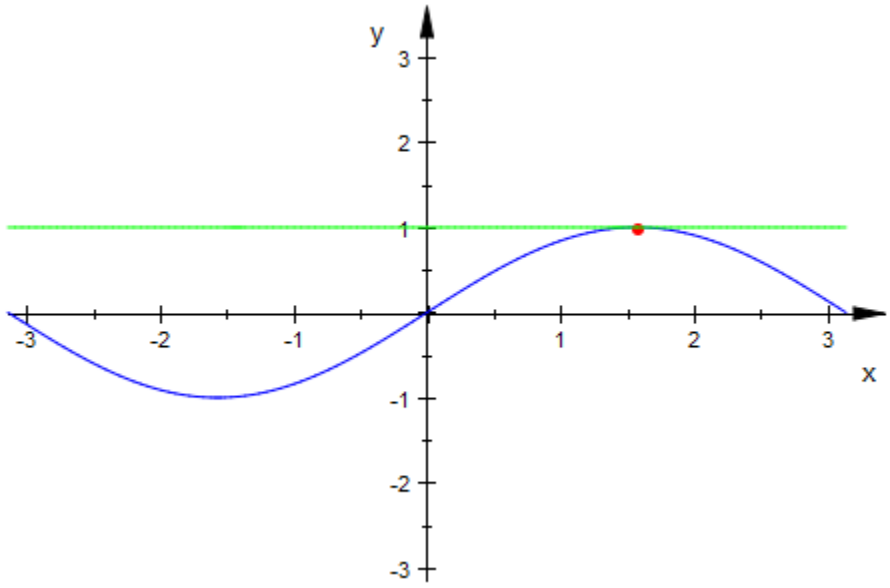
We plot a simple 3D function. Note that the option #3D (alias #3, see "Example 12" on page 24-67) is required in the following example for plotting a 3D function. Otherwise, an animated 2D function is created.

```
plot(m*x, #3D)
```



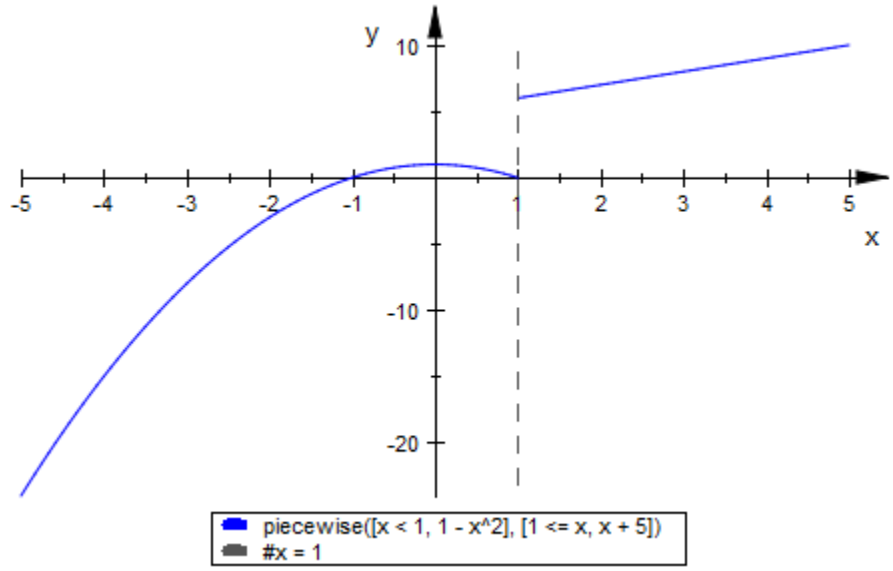
We plot a simple 2D animation: a point and the corresponding tangent of the sine function move along the sine function graph:

```
f:= x -> sin(x):
plot(f(x), x = -PI..PI,
     {[a, f(a)], #Points}, f'(a)*(x-a) + f(a), a = -PI/2..PI/2)
```



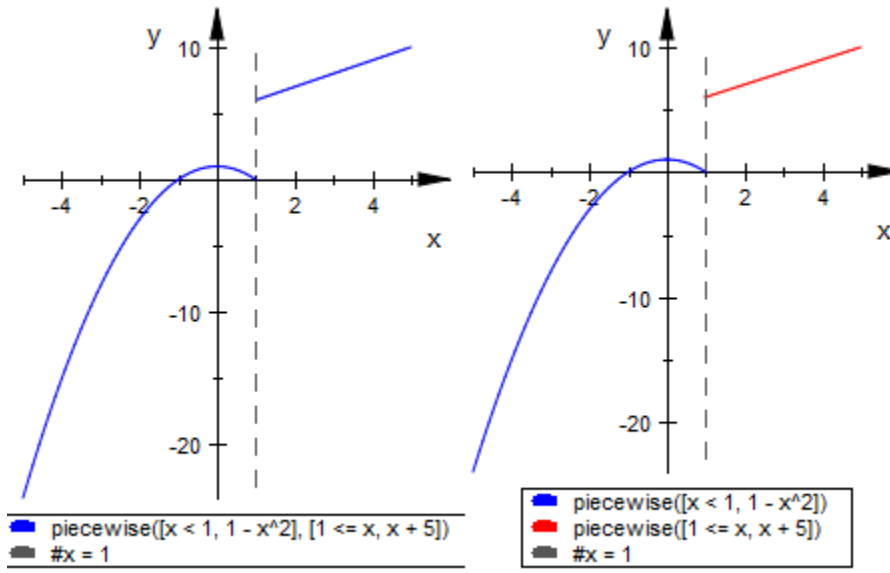
We plot a piecewise defined function:

```
plot(piecewise([x < 1, -x^2 + 1], [x >= 1, x + 5]), #x=1, #Legend)
```



The same piecewise defined function is now written in a shorter syntax. Note the difference between defining one function with two branches (left) and defining two functions (right):

```
plot({{[[x < 1, -x^2 + 1], [x >= 1, x + 5]], #x=1}},
     {{ [x < 1, -x^2 + 1], [x >= 1, x + 5] , #x=1}},
     #Legend)
```

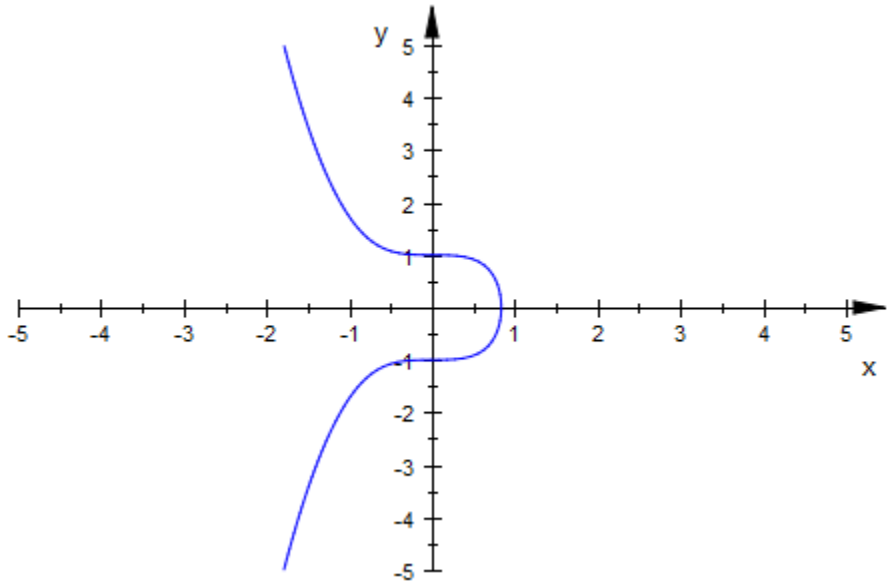


See also: `plot::Function2d`, `plot::Function3d`.

Example 9

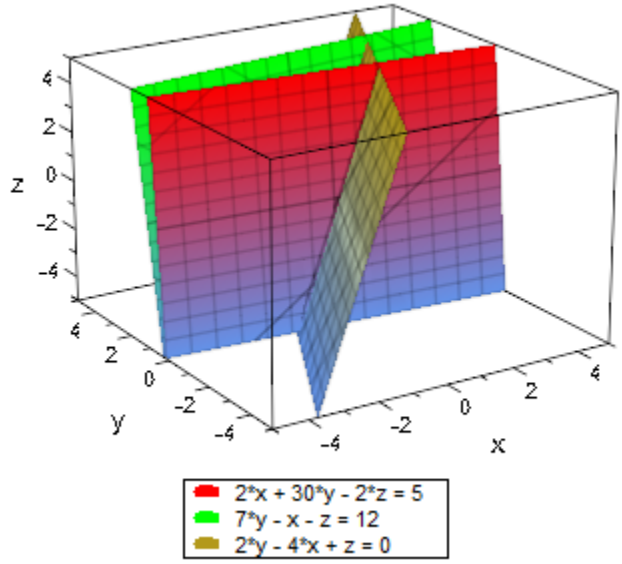
An implicit function can be specified as an equation:

```
plot(u^5 + x^2 = 1 - u^3)
```



Note that the option #3D (alias #3, see “Example 12” on page 24-67) may be required to plot planes given as cartesian equations. Otherwise, an animated 2D graph might be created. This depends on the number of variable of the equation.

```
E1:= 2*x + 30*y - 2*z = 5:
E2:= -x + 7*y - z = 12:
E3:= -4*x + 2*y + z = 0:
plot(E1, E2, E3, #3D, #Legend)
```



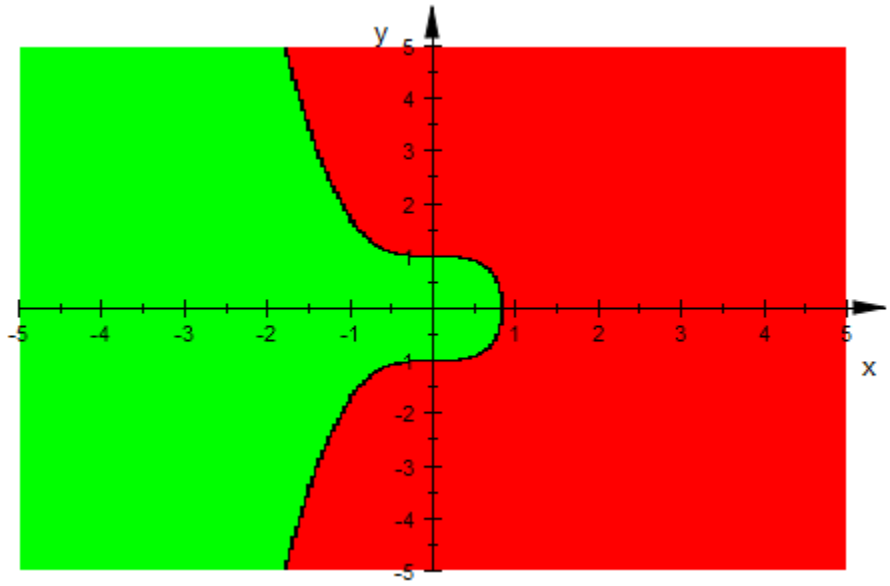

```
delete E1, E2, E3:
```

See also: `plot::Implicit2d`, `plot::Implicit3d`.

Example 10

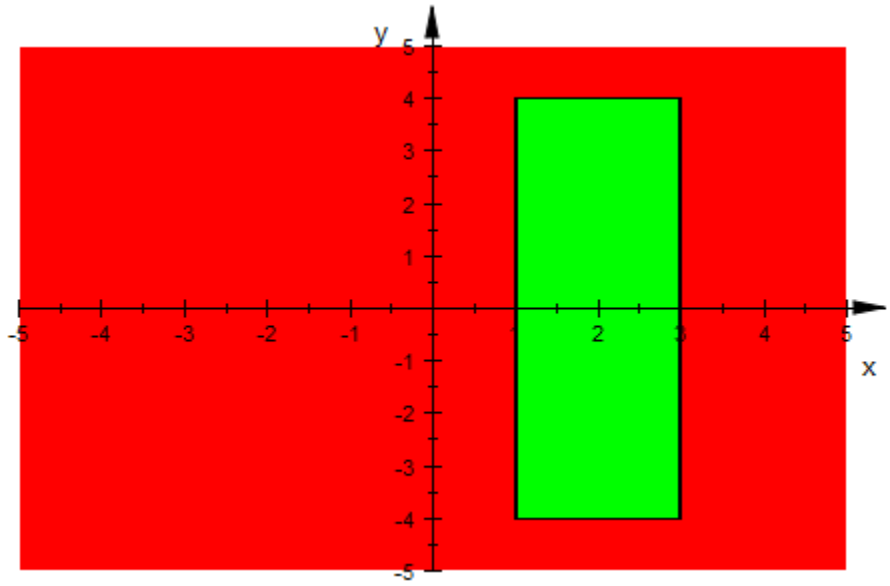
An inequality can be displayed directly:

```
plot(u^5+x^2 < 1-u^3)
```



The same is true for a list of inequalities and equations:

```
plot([x < 3, x > 1, y <  4, y > -4])
```

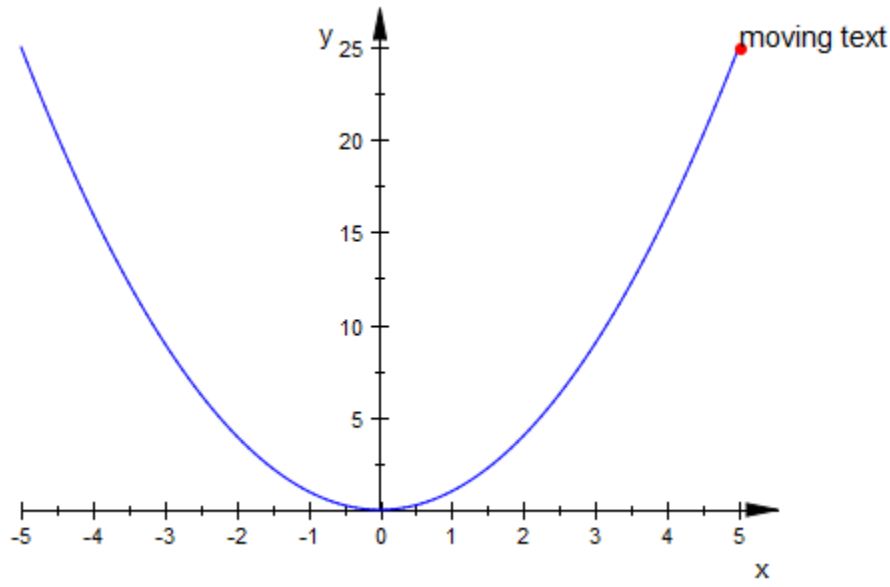


See also: `plot::Inequality`.

Example 11

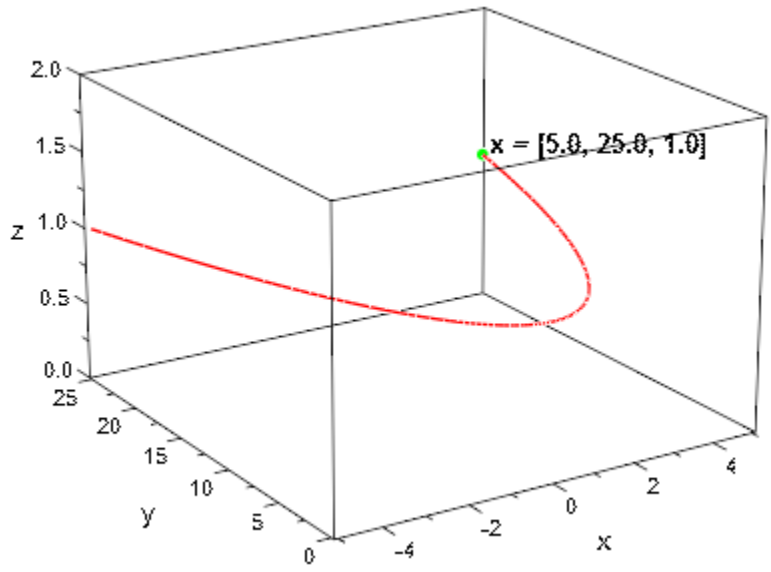
To display a 2D text at a certain position, an equation of a coordinate tuple and a character string or a procedure can be entered:

```
plot([t, t^2],
    {[t, t^2], #Points},
     [t, t^2] = "moving text")
```



We display a 3D text. As in any other context of `plot::easy`, we can use regular graphical attributes like `TextFont =Center` as well:

```
DIGITS := 2:
plot([t, t^2, 1],
    {[t, t^2, 1], #Points},
     [t, t^2, 1] = (t->" x = ".[t, t^2, 1.0]),
     TextFont=[Bold]):
delete DIGITS:
```

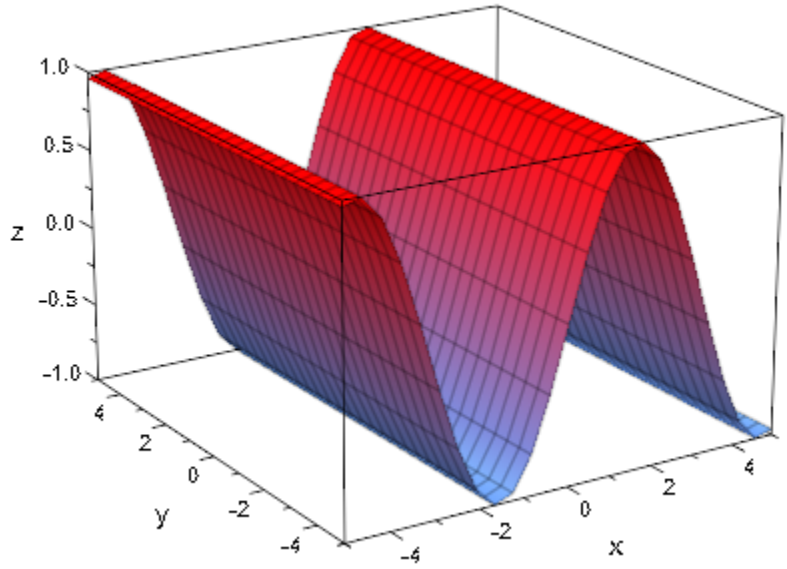


See also: `plot::Text2d`, `plot::Text3d`.

Example 12

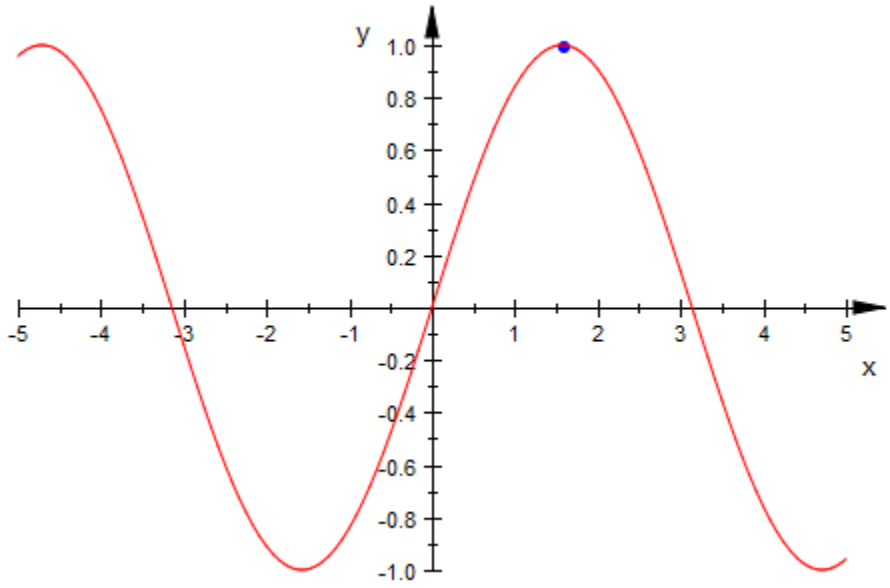
Usually `plot::easy` creates a 2D scene, unless one of the arguments is a 3D object or can only be transformed to a 3D object or the option `#3D` is used.

```
plot(sin(x), #3)
```



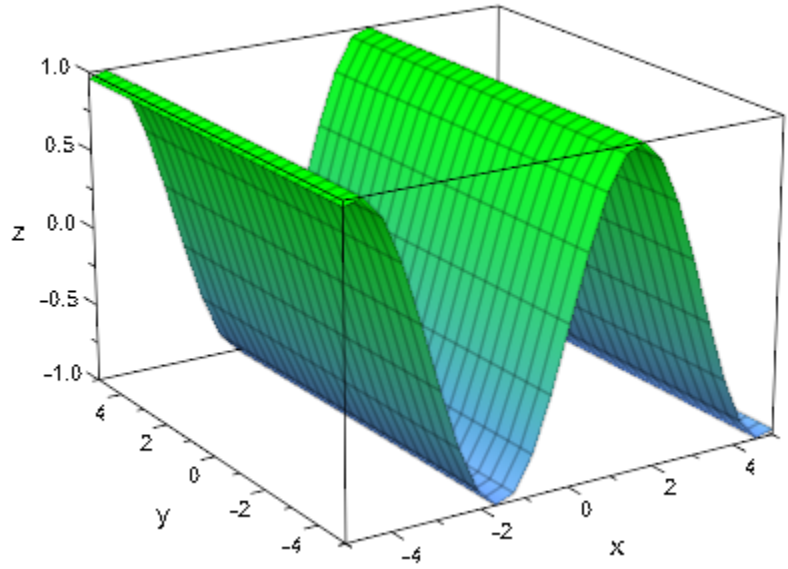
However, the option `#3D` is only a hint. It is ignored if the current scene can only be a 2D scene. In the following example, the 2D point determines the dimension of the scene:

```
plot([PI/2,1], sin(x), #3)
```



In the following example, the 3D point determines the dimension of the scene. There is no need to use option #3D in order to create a 3d scene:

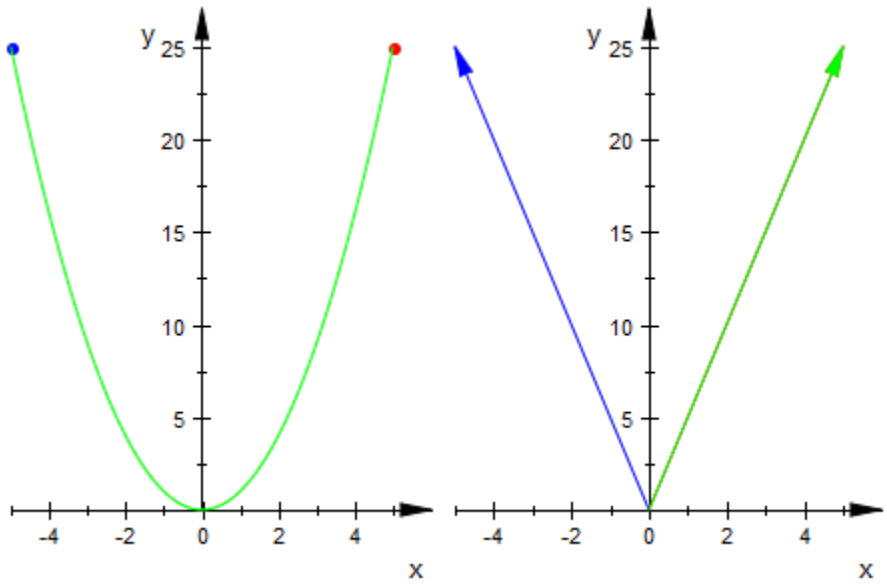
```
plot([PI/2,1,0], sin(x))
```



Example 13

With option `#Arrows`, arrows instead of points or curves are created. When used within a {...}-group, it affects the elements of this group or scene only.

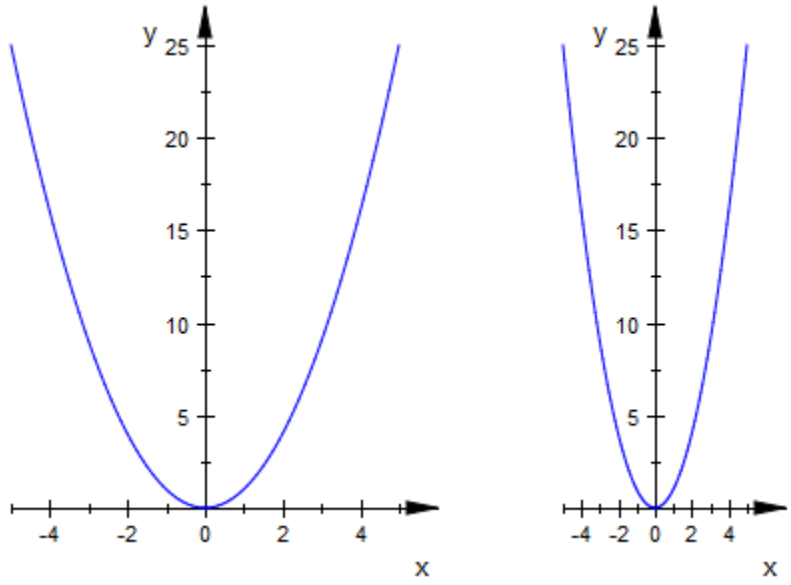
```
plot({{ [-5,25], [5,25], [x,x^2]     }},
     {{ [-5,25], [5,25], [x,x^2], #A }})
```



Example 14

Option `#Constrained` creates a coordinate system with constrained scaled axes. This is a shortcut for `Scaling = Constrained`.

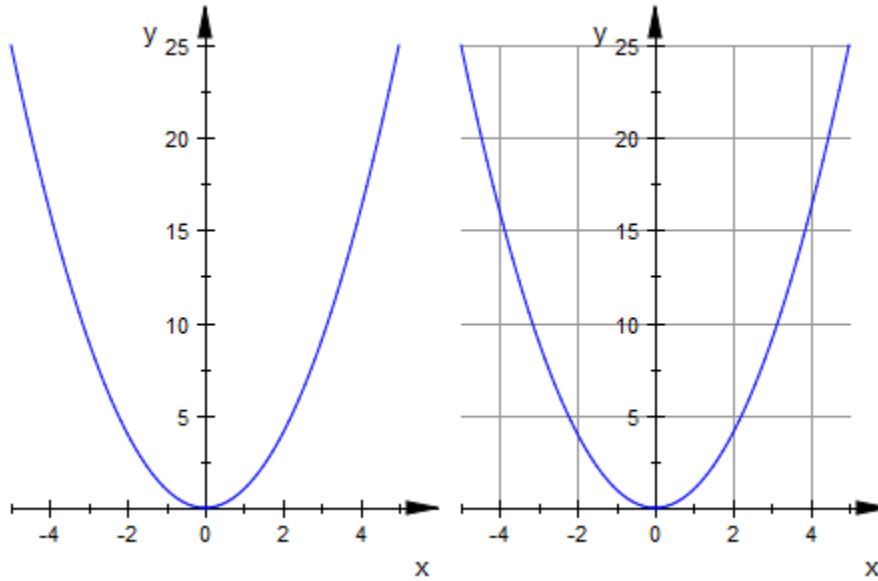
```
plot({{ x^2 }}, {{ x^2, #C }})
```



Example 15

Option `#Grid` creates a coordinate system with grid lines. This is a shortcut for `GridVisible = TRUE`.

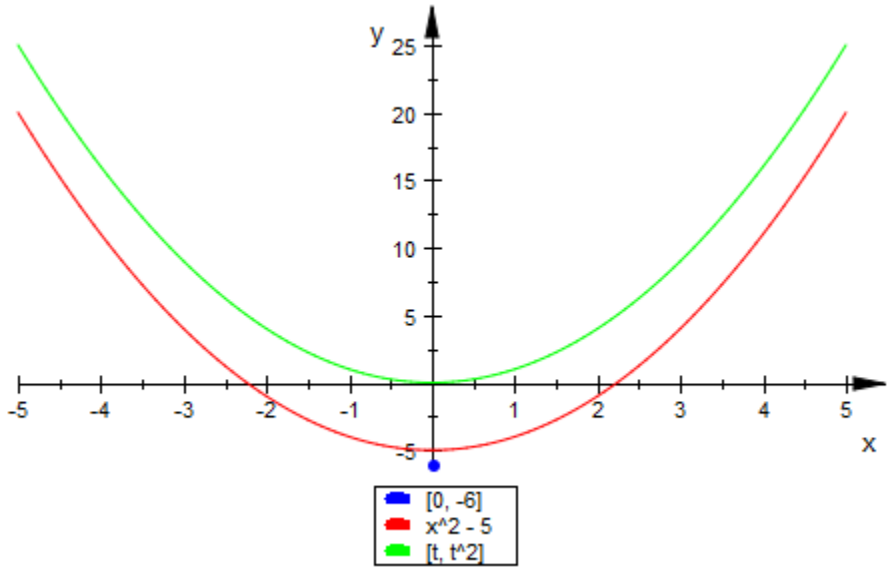
```
plot({{ x^2 }}, {{ x^2, #G }})
```



Example 16

Option `#Legend` creates a legend. This is a shortcut for `LegendVisible = TRUE` in combination with `LegendEntry = TRUE`. Note that `plot::easy` explicitly sets a legend text for each graphical object it creates.

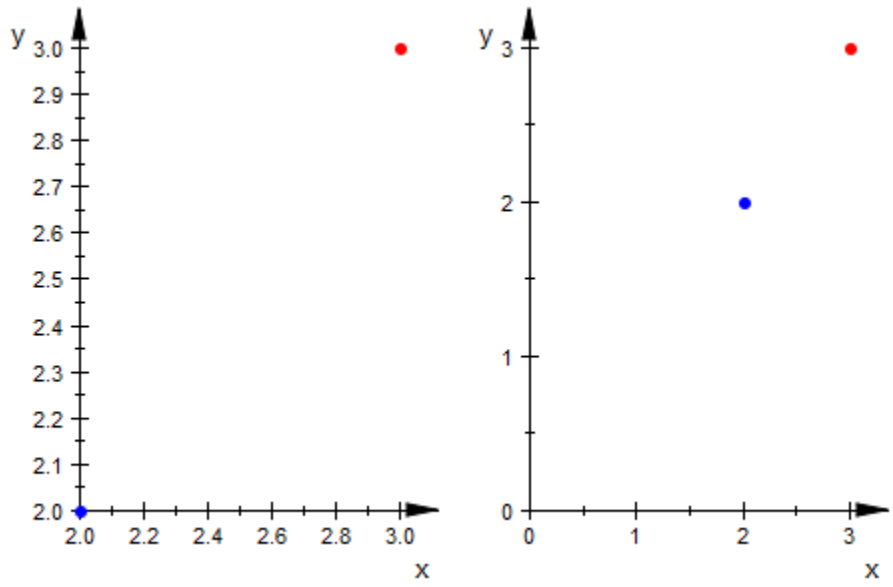
```
plot([0,-6], x^2-5, [t,t^2], #L)
```



Example 17

Option `#Origin` includes the coordinates (0,0) or (0,0,0), respectively, into the viewing box of the current scene.

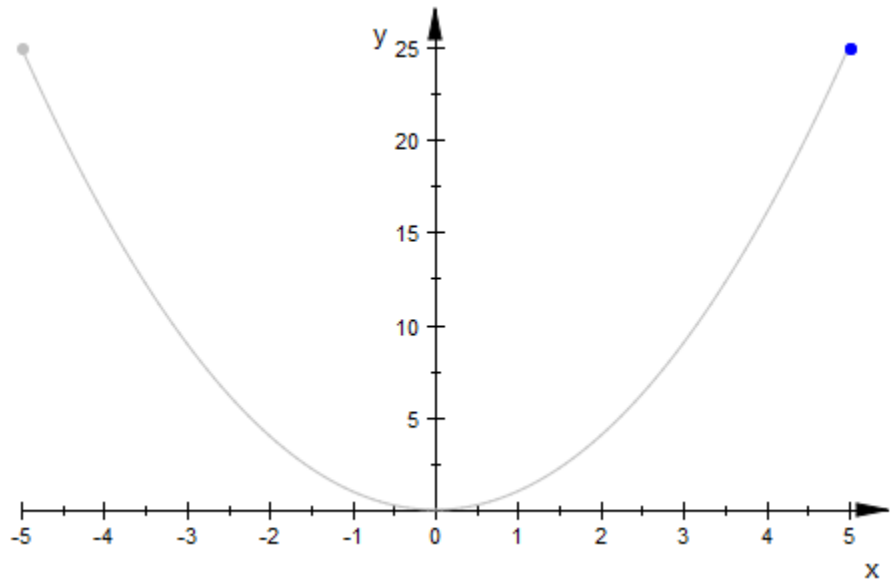
```
plot({{[2,2], [3,3]}}, {{[2,2], [3,3], #0}})
```



Example 18

Using option `#Points`, points instead of arrows or curves are created. Furthermore, this option sets the attributes `PointsVisible =TRUE` and `LinesVisible =FALSE`. When used within a {...}-group, it affects the elements of this group or scene only.

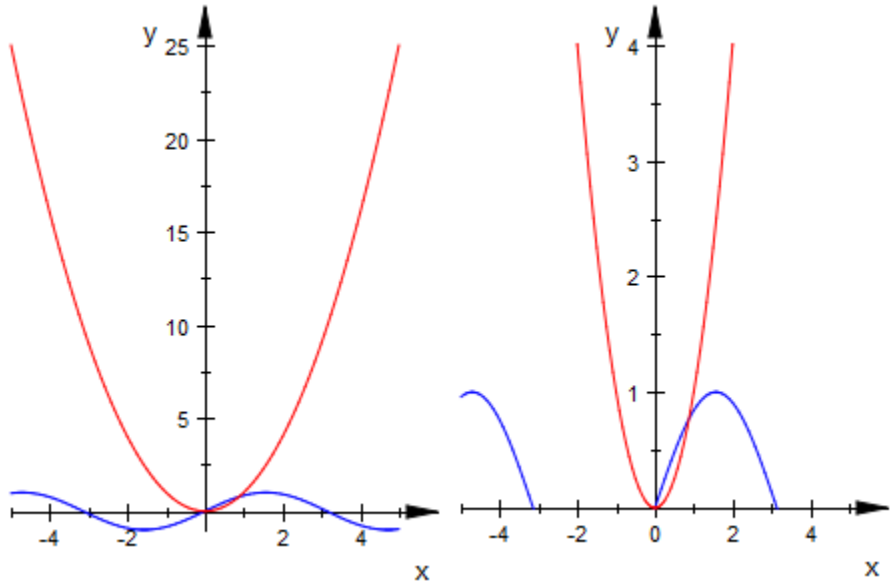
```
plot({[-5,25], [x,x^2], #Gray}, {matrix([5,25]), [x,x^2], #P})
```



Example 19

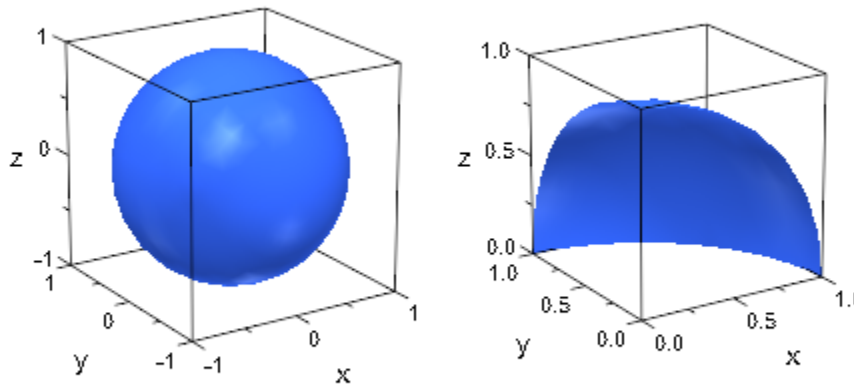
`#X / #Y / #Z = a..b` sets the x- / y- / z-range of the viewing box of the scene to `a..b`. This is a shortcut for `ViewingBoxXRange / ViewingBoxYRange / ViewingBoxZRange = a..b`.

```
plot({{x^2, sin(x)}}, {{x^2, sin(x), #Y=0..4}})
```



We draw a 3D scene with a restricted viewing box. Therefore, only a quater of the shere is visible:

```
plot({{plot::Sphere(1)}},
     {{plot::Sphere(1), #X=0..1, #Y=0..1, #Z=0..1}})
```



Parameters

`arg`

Any object

Options

`#3D`

Alias #3. Creates a 3D instead of a 2D scene, if possible. Usually, a 2D scene is created unless one of the arguments is a 3D object or can only be transformed into one.

`#Arrows`

Alias #A. Creates arrows instead of points. When used within a {...}-group, it affects the elements of this group or scene only.

#Constrained

Alias #C. Creates a coordinate system with axes having the same scaling. This is a shortcut for `Scaling =Constrained`.

#Grid

Alias #G. Creates a coordinate system with grid lines. This is a shortcut for `GridVisible =TRUE`.

#Legend

Alias #L. Creates a legend. This is a shortcut for `LegendEntry =TRUE` and `LegendVisible =TRUE`. When used within a {...}-group, it affects the elements of this group or scene only. Note that `plot::easy` explicitly sets a legend text for each graphical object it creates.

#Origin

Alias #O. Includes the coordinates (0,0) or (0,0,0), respectively, into the viewing box of the current scene.

#Points

Alias #P. Creates points instead arrows or curves. Furthermore, it sets the attributes `PointsVisible =TRUE` and `LinesVisible =FALSE`. When used within a {...}-group, it affects the elements of this group or scene only.

#XRange

Option, specified as `#XRange = a .. b`

Alias #X = a .. b. Sets the x-range of the viewing box of the scene to `a..b`. This is a shortcut for `ViewingBoxXRange =a..b`.

#YRange

Option, specified as `#YRange = a .. b`

Alias #Y = a .. b. Sets the y-range of the viewing box of the scene to `a..b`. This is a shortcut for `ViewingBoxYRange =a..b`.

#ZRange

Option, specified as `#ZRange = a .. b`

Alias `#Z` = a .. b. Sets the z-range of the viewing box of the scene to `a..b`. This is a shortcut for `ViewingBoxZRange =a..b`.

`#<Colorname>`

If `RGB ::<Colorname>` is a valid color name in MuPAD, `#<Colorname>` is transformed to: `Color = RGB ::<Colorname>`, `LineColorType =Flat` and for 3D objects additionally `FillColorType =Flat`. Transparent `RGB` colors can be specified as `#<Colorname>.[t]`, with t is in 0..1. If `#<Colorname>` is a valid color following the html conventions then instead of `RGB ::<Colorname>`, the corresponding RGB or RGBa color value is inserted. When used within a {...}-group, it affects the elements of this group or scene only.

`#<Colorname1> .. #<Colorname2>`

If `RGB ::<Colorname1>` and `RGB ::<Colorname2>` are valid color names in MuPAD, this option is transformed to:

`Color = RGB ::<Colorname1>`, `LineColorType =Dichromatic`, `LineColor2 = RGB ::<Colorname2>` in 2D scenes and to:

`Color = RGB ::<Colorname1>`, `FillColorType =Dichromatic`, `FillColor2 = RGB ::<Colorname2>` in 3D scenes.

Transparent `RGB` colors can be specified as `#<Colorname>.[t]`, with t is in 0..1. If `#<Colorname1>` and/or `#<Colorname2>` are valid colors following the html conventions then instead of `RGB ::<Colorname1>` and/or, `RGB ::<Colorname2>` the corresponding RGB or RGBa color values are inserted. When used within a {...}-group, it affects the elements of this group or scene only.

Return Values

A sequence of graphical objects and graphical attributes as well as objects that could not be transformed by `plot::easy`.

Overloaded By

arg

Algorithms

Let c_i be real constants and f and f_i be real functions. `plot::easy` automatically carries out the following transformations:

Graphical object	Data or mathematical expression
`plot::Point2d`:	$[c_1, c_2]$, {$[f_1(x), f_2(x)]$, #Points}, {matrix($[f_1(x), f_2(x)]$), #Points}.
`plot::Point3d`:	$[c_1, c_2, c_3]$, {$[f_1(x), f_2(x), f_3(x)]$, #Points}, {matrix($[f_1(x), f_2(x), f_3(x)]$), #Points}.
`plot::Arrow2d`:	matrix($[c_1, c_2]$), {$[c_1, c_2]$, #Arrows}, {[matrix($[f_1(x), f_2(x)]$), matrix($[f_3(x), f_4(x)]$)], #Arrows}, {matrix($[[f_1(x), f_2(x)], [f_3(x), f_4(x)]]$), #Arrows}, {$[[f_1(x), f_2(x)], [f_3(x), f_4(x)]]$, #Arrows}.
`plot::Arrow3d`:	matrix($[c_1, c_2, c_3]$), {$[c_1, c_2, c_3]$, #Arrows}, {[matrix($[f_1(x), f_2(x), f_3(x)]$), matrix($[f_4(x), f_5(x), f_6(x)]$)], #Arrows}, {matrix($[[f_1(x), f_2(x), f_3(x)], [f_4(x), f_5(x), f_6(x)]]$), #Arrows}, {$[[f_1(x), f_2(x), f_3(x)], [f_4(x), f_5(x), f_6(x)]]$, #Arrows}.
`plot::Polygon2d`:	$[[f_1(x), f_2(x)],...]$, [matrix($[f_1(x), f_2(x)]$),...], matrix($[[f_1(x), f_2(x)],...]$), table($f_1(x)=f_2(x)$,...).
`plot::Polygon3d`:	$[[f_1(x), f_2(x), f_3(x)],...]$, [matrix($[f_1(x), f_2(x), f_3(x)]$),...], matrix($[[f_1(x), f_2(x), f_3(x)],...]$). table($f_1(x)=[f_2(x), f_3(x)]$,...).
`plot::PointList2d`:	$[[f_1(x), f_2(x),$ RGBa$],...]$, [matrix($[f_1(x), f_2(x),$ RGBa$]$),...], matrix($[[f_1(x), f_2(x),$ RGBa$],...]$), table($f_1(x)=[f_2(x),$ RGBa$],...$).

Graphical object	Data or mathematical expression
plot::PointList3d:	[[f_1(x), f_2(x), f_3(x), RGBa],...], [matrix([f_1(x), f_2(x), f_3(x), RGBa]),...], matrix([[f_1(x), f_2(x), f_3(x), RGBa],...]). table(f_1(x)=[f_2(x), f_3(x), RGBa],...).
plot::Line2d:	#x= f(x), #y= f(x).
plot::Plane:	#z= f(x), {#x= f(x) , #3D}, {#y= f(x) , #3D}, {#z= f(x) , #3D}.
plot::Curve2d:	[f_1(x), f_2(x)].
plot::Curve3d:	[f_1(x), f_2(x), f_3(x)].
plot::Function2d:	f(x), f(x, a), [$cond_1$, f(x)], [[$cond_1$, f(x)],...].
plot::Function3d:	f(x, y, a), [$cond_1$, f(x, y, a)], [[$cond_1$, f(x, y, a)],...], {f(x), #3D}, {f(x, a), #3D}.
plot::Implicit2d:	f_1(x, y, a)=f_2(x, y, a).
plot::Implicit3d:	f_1(x, y, z, a)=f_2(x, y, z, a), {f_1(x, y, a)=f_2(x, y, a), #3D}.
plot::Inequality:	f_1(x, a) < f_2(x, a), f_1(x, a) ≤ f_2(x, a), f_1(x, a) > f_2(x, a), f_1(x, a) ≥ f_2(x, a), [f_1(x, a)<f_2(x, a), f_3(x, a)>f_4(x, a), f_5(x, a)=f_6(x, a),...].
plot::Text2d:	[f_1(x), f_2(x)]=text, matrix([f_1(x), f_2(x)])=text, [f_1(x), f_2(x)]=procedure, matrix([f_1(x), f_2(x)])=procedure.
plot::Text3d:	[f_1(x), f_2(x), f_3(x)]=text, matrix([f_1(x), f_2(x), f_3(x)])=text, [f_1(x), f_2(x), f_3(x)]=procedure, matrix([f_1(x), f_2(x), f_3(x)])=procedure.

See Also

MuPAD Functions
`plot`

plot::getDefault

Get current default setting of attributes

Syntax

```
plot::getDefault(type::attr)
```

Description

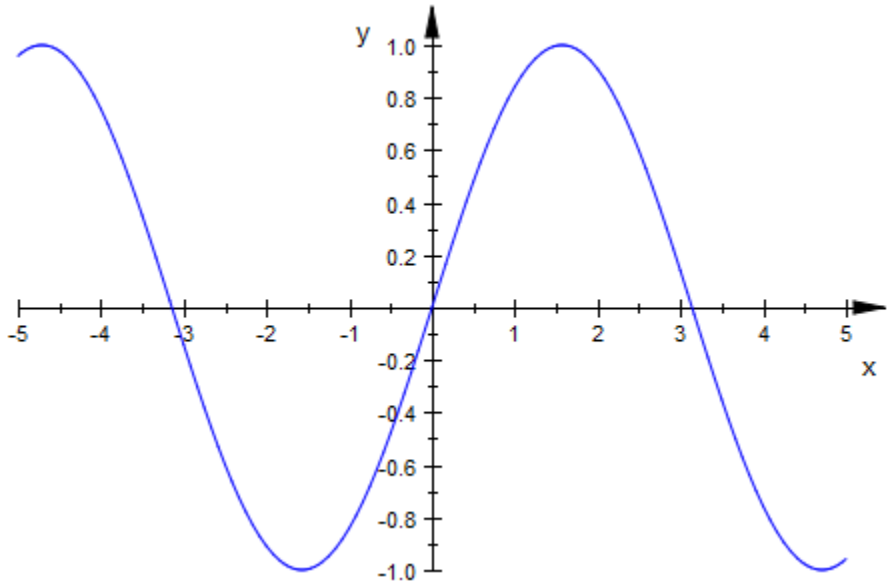
`plot::getDefault(plot::Object::Attribute)` enquires the current default.

Examples

Example 1

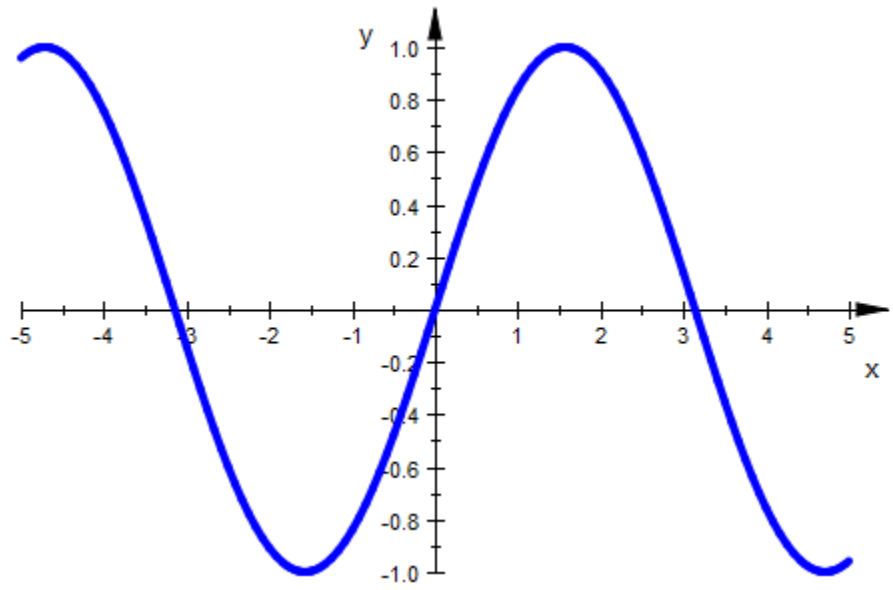
By default, function plots use relatively thin lines:

```
plotfunc2d(sin(x))
```



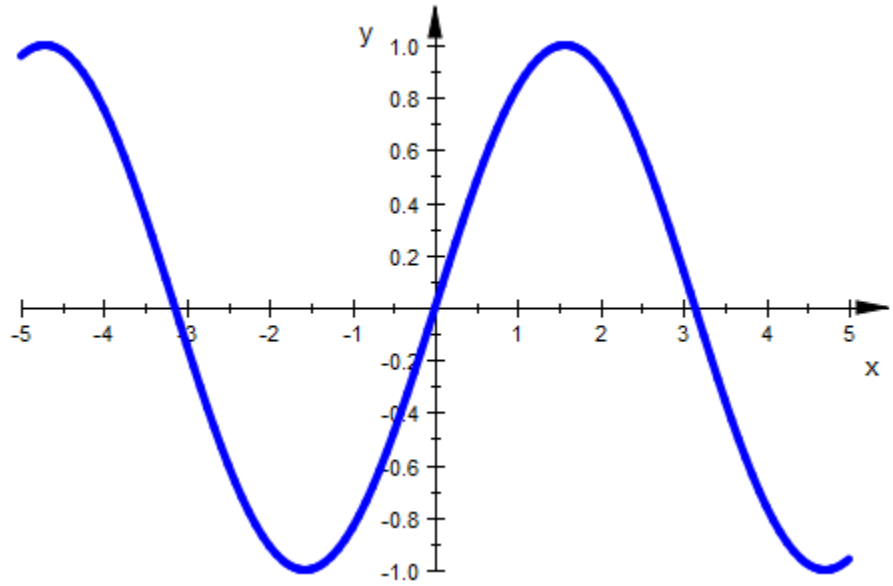
For some applications, this is undesirable, for example when projecting graphics for a larger audience. It is always possible to set thicker lines in the call:

```
plotfunc2d(sin(x), LineWidth = 1*unit::mm)
```



However, this is distracting and cumbersome. Using `plot::setDefault`, we change the default setting once and for the whole session:

```
plot::setDefault(plot::Function2d::LineWidth = 1*unit::mm):
plotfunc2d(sin(x))
```



One thing you should know in this context: `plotfunc2d` and `plotfunc3d` use `plot::Function2d` and `plot::Function3d` for the actual plotting. Changing color and legend settings of the latter two does not influence the former, however, since `plotfunc2d` and `plotfunc3d` set color and legend settings explicitly.

Parameters

`type`

A domain of the plot library, i.e., an object type such as `plot::Function2d`

`attr`

Attributes admissible for the object type `type`

Return Values

`plot::setDefault` returns the previous default value(s). `plot::getDefault` returns the current default value.

Algorithms

"Admissible attributes" includes all the attributes the object itself reacts to. Hints cannot be set or changed with `plot::setDefault`.

For attributes marked as "mandatory," default values are read and used the moment an object is created. Default values of attributes marked as "optional" or "inherited" are read when the object is plotted and can therefore be changed after creating an object.

See Also

MuPAD Functions

`plot::setDefault`

plot::setDefault

Set default setting of attributes

Syntax

```
plot::setDefault(type::attr = value, …)
```

Description

`plot::setDefault(plot::Object::Attribute = Value)` sets the default of the attribute `Attribute` for objects of type `plot::Object` to `Value`.

While not all attributes have defaults, it is in general possible to set defaults for them, although some examples like setting a default function to plot for `plot::Function2d` are probably more exotic than others, to say the least.

Defaults are set and retrieved per object; with the exception of `OutputFile` and `OutputOptions`, the attribute must be prefixed with the name of the object type the setting shall be valid for. There is, e.g., no function to turn of all lines on all 3D objects. `OutputFile` and `OutputOptions` are not associated with an object and must be set directly.

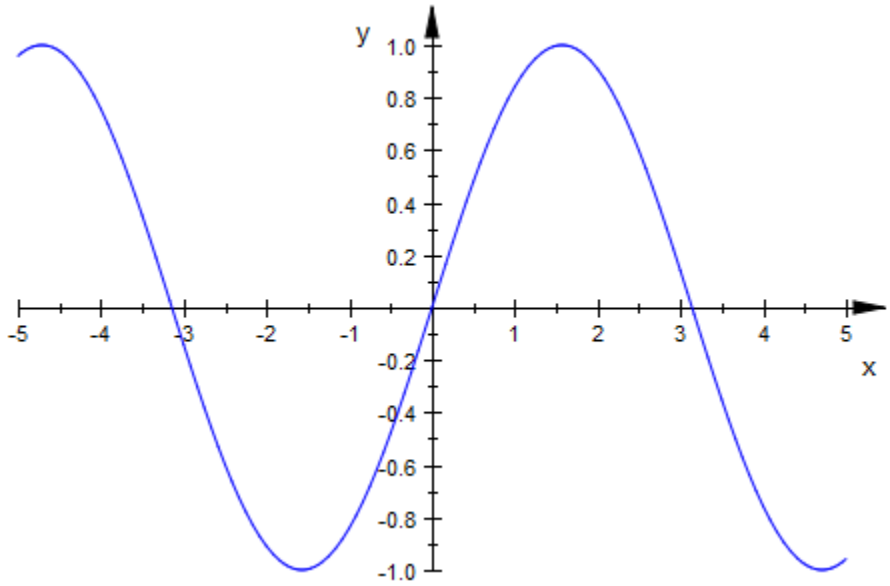
To delete a default (which is not recommended for attributes having a default in the standard installation), set `value` to `FAIL`.

Examples

Example 1

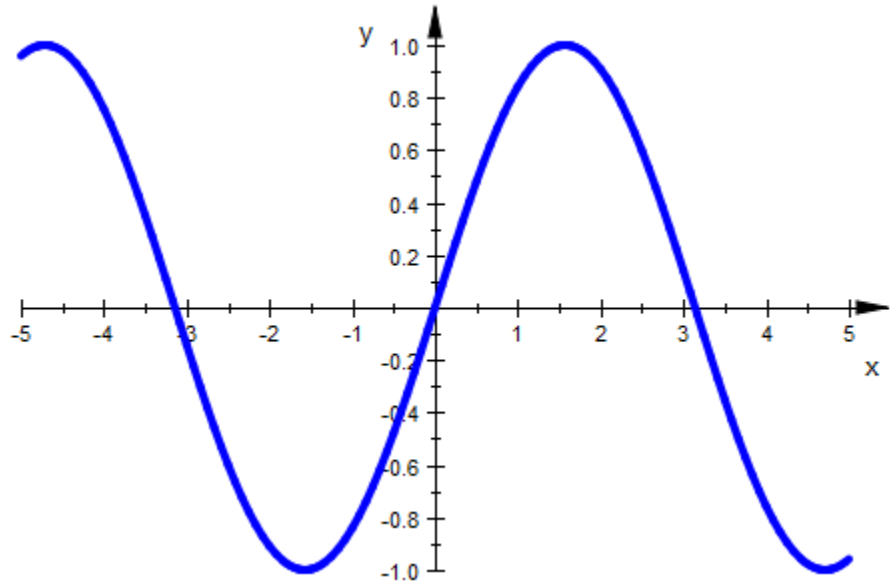
By default, function plots use relatively thin lines:

```
plotfunc2d(sin(x))
```



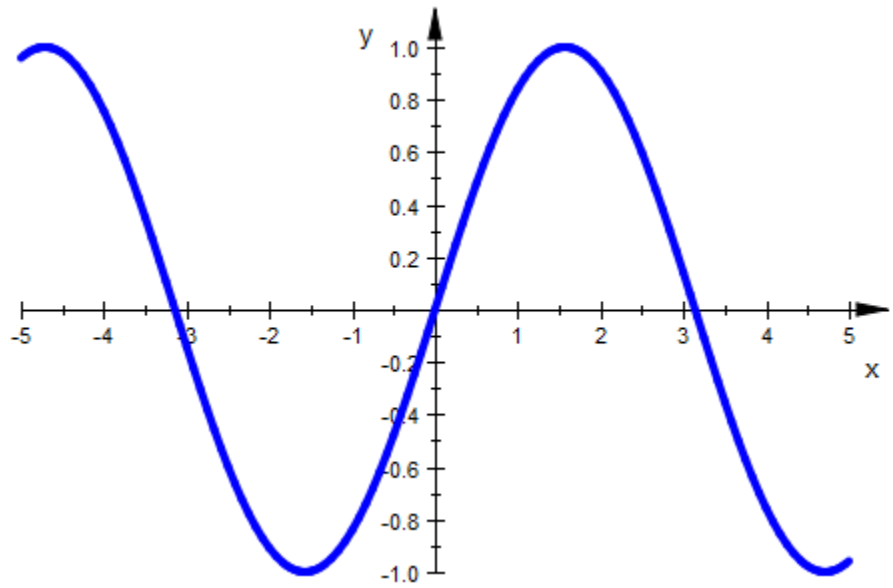
For some applications, this is undesirable, for example when projecting graphics for a larger audience. It is always possible to set thicker lines in the call:

```
plotfunc2d(sin(x), LineWidth = 1*unit::mm)
```



However, this is distracting and cumbersome. Using `plot::setDefault`, we change the default setting once and for the whole session:

```
plot::setDefault(plot::Function2d::LineWidth = 1*unit::mm):
plotfunc2d(sin(x))
```



One thing you should know in this context: `plotfunc2d` and `plotfunc3d` use `plot::Function2d` and `plot::Function3d` for the actual plotting. Changing color and legend settings of the latter two does not influence the former, however, since `plotfunc2d` and `plotfunc3d` set color and legend settings explicitly.

Parameters

`type`

A domain of the plot library, i.e., an object type such as `plot::Function2d`

`attr`

Attributes admissible for the object type `type`

`value`

The new default value: a value admissible for `attr` in objects of type `type`

Return Values

`plot::setDefault` returns the previous default value(s). `plot::getDefault` returns the current default value.

Algorithms

"Admissible attributes" includes all the attributes the object itself reacts to. Hints cannot be set or changed with `plot::setDefault`.

For attributes marked as "mandatory," default values are read and used the moment an object is created. Default values of attributes marked as "optional" or "inherited" are read when the object is plotted and can therefore be changed after creating an object.

See Also

MuPAD Functions
`plot::getDefault`

plot::copy

Make a physical copy of a plot object

Syntax

```
plot::copy(obj)
```

Description

Plot objects usually have a reference effect. `plot::copy` creates copies which are independent of the original.

Objects created from inside the `plot` library have a *reference effect:* If you make another reference to some object, say by calling `o2 := o1;` and then change an attribute of `o2`, e.g., setting `o2::Visible := FALSE`, this change will also effect the object referred to by `o1`, since they actually refer to the same object. To create an actual copy of an object instead, use `o2 := plot::copy(o1);`.

The function `plot::modify` is a variant of `plot::copy`. It allows setting new values of attributes in the same call, as in `o2 := plot::modify(o1, Visible = FALSE);`.

Examples

Example 1

The following call does *not* create two points, but rather one which we can access by two names:

```
A := plot::Point2d(0, 0):
B := A:
```

This surfaces as soon as we try to modify `one of the points':

```
B::Position := [1, 1]:
A
```

```
plot::Point2d(1, 1)
```

Instead, we can use `plot::modify` to achieve the desired effect:

```
B := plot::modify(A, Position = [2, 2]):
A, B
```

```
plot::Point2d(1, 1), plot::Point2d(2, 2)
```

Note: Note that `plot::modify` does not modify its argument, but returns a modified copy instead, whatever the name may suggest.

Parameters

`obj`

Plot objects

Return Values

Object of the same type as `obj`

See Also

MuPAD Functions

`plot::modify`

plot::modify

Make a physical copy of a plot object setting new values of attributes in the same call

Syntax

```
plot::modify(obj, <attr, …>)
```

Description

Plot objects usually have a reference effect. `plot::modify` creates copies which are independent of the original.

Objects created from inside the `plot` library have a *reference effect:* If you make another reference to some object, say by calling `o2 := o1;` and then change an attribute of `o2`, e.g., setting `o2::Visible := FALSE`, this change will also effect the object referred to by `o1`, since they actually refer to the same object. To create an actual copy of an object instead, use `o2 := plot::copy(o1);`.

The function `plot::modify` is a variant of `plot::copy`. It allows setting new values of attributes in the same call, as in `o2 := plot::modify(o1, Visible = FALSE);`.

Examples

Example 1

The following call does *not* create two points, but rather one which we can access by two names:

```
A := plot::Point2d(0, 0):
B := A:
```

This surfaces as soon as we try to modify `one of the points':

```
B::Position := [1, 1]:
A
```

```
plot::Point2d(1, 1)
```

Instead, we can use `plot::modify` to achieve the desired effect:

```
B := plot::modify(A, Position = [2, 2]):
A, B
```

```
plot::Point2d(1, 1), plot::Point2d(2, 2)
```

Note: Note that `plot::modify` does not modify its argument, but returns a modified copy instead, whatever the name may suggest.

Parameters

`obj`

Plot objects

`attr`

Attributes admissible for the object `obj`, in the form `Attribute = Value`

Return Values

Object of the same type as `obj`

See Also

MuPAD Functions

`plot::copy`

plot::delaunay

Compute the Delaunay triangulation of a set of points

Syntax

```
plot::delaunay(L)
```

Description

`plot::delaunay` computes the Delaunay triangulation of a list of points in arbitrary dimension.

The Delaunay triangulation of a list of points is a triangulation of their convex hull such that for each edge of the triangulation, there is a circle containing the two endpoints of this edge but no other point of the list.

Environment Interactions

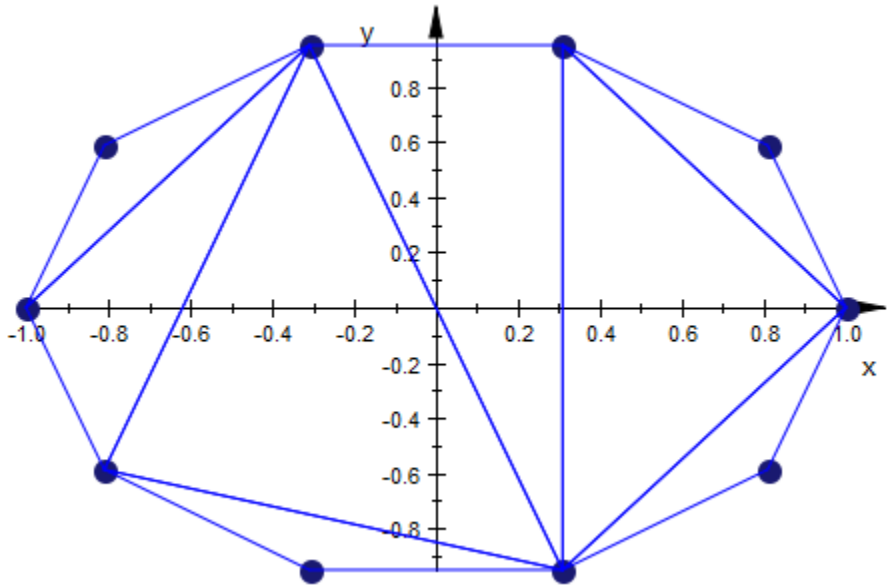
Although `plot::delaunay` accepts and returns floating-point values, the actual computations take place in hardware floating-points and are therefore *not* affected by the value of `DIGITS`.

Examples

Example 1

Delaunay triangulation does not introduce new points:

```
n0 := 10:
l := [[Re, Im](exp(float(2*I*PI*n)/n0)) $ n = 1.. n0]:
d := plot::delaunay(l):
plot(plot::PointList2d(l, PointSize=3),
     plot::Polygon2d(t, Closed) $ t in d)
```



Parameters

L

A list of points, which are given as lists of real values

Return Values

List of simplices in the dimension of the points in L, given as lists of lists of floating-point values.

Algorithms

`plot::delaunay` uses qhull from the Geometry Center of the University of Minnesota, see www.qhull.org.

plot::hull

Compute the convex hull of a set of points

Syntax

```
plot::hull(L)
```

Description

`plot::hull` computes the convex hull of a list of points in any dimension, i.e., the smallest convex region containing all the points. Such a region is bounded by simplices (straight lines in the plane, triangles in 3D) and it is these simplices which `plot::hull` returns.

Environment Interactions

Although `plot::hull` accepts and returns floating point values, the actual computations take place in hardware floating points and are therefore *not* affected by the value of `DIGITS`.

Examples

Example 1

We generate a list of random points and compute their convex hull:

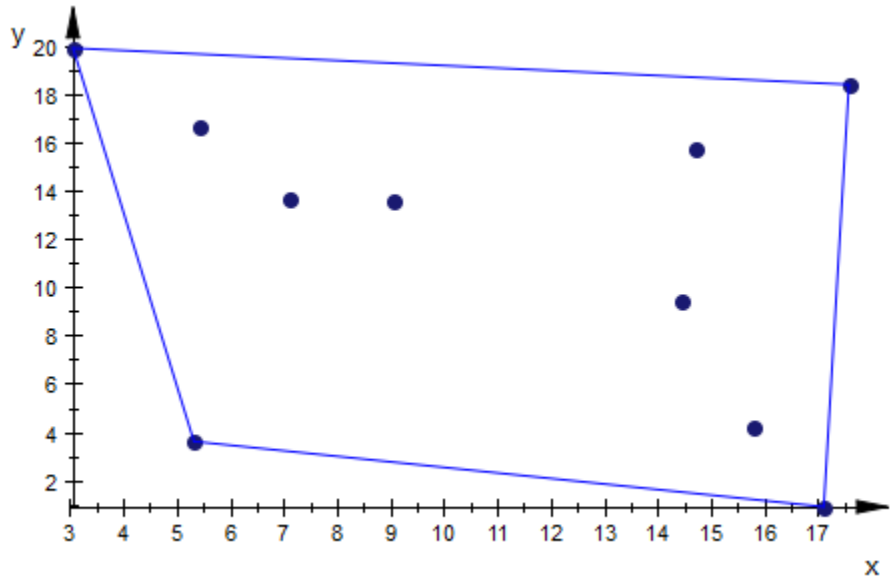
```
X := stats::uniformRandom(0, 20):
l := [[X(), X()] $ i = 1..10]:
h := plot::hull(l):
```

The convex hull is returned as lists of lists, as accepted by `plot::Polygon2d`:

```
h[1]
```

```
[[17.58320254, 18.38769696], [3.063130321, 19.89625562]]
```

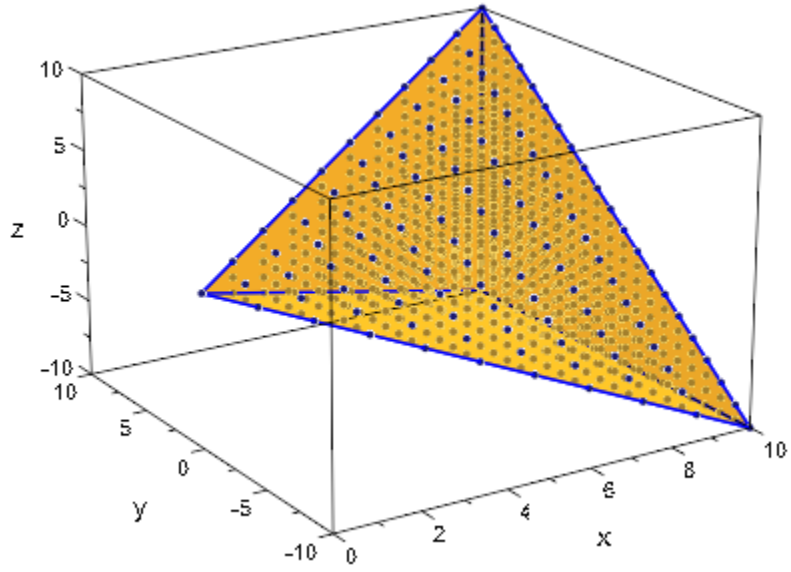
```
plot(plot::PointList2d(l),
     plot::Polygon2d(t) $ t in h,
     Closed, PointSize=2)
```



Example 2

The convex hull of a list of points in 3D is also easy to visualize:

```
l := [[x, y, z] $ y = z..x $ z = -x..x $ x = 0..10]:
h := plot::hull(l):
plot(plot::PointList3d(l, PointSize=1),
     plot::Polygon3d(t) $ t in h,
     Closed, Filled, FillColor=RGB::LightOrange.[0.6])
```



Parameters

L

A list of points, which are given as lists of real values

Return Values

List of simplices of dimension one less than that of the points in L, given as lists of lists of floating-point values.

Algorithms

`plot::hull` uses qhull from the Geometry Center of the University of Minnesota, see www.qhull.org.

plot::Arc2d

Circular and elliptical arcs in 2D

Syntax

```
plot::Arc2d(r, <[c_x, c_y]>, <α .. β>, <a = a_min .. a_max>, options)
```

```
plot::Arc2d([r_1, r_2], <[c_x, c_y]>, <α .. β>, <a = a_min .. a_max>, options)
```

Description

`plot::Arc2d(r, [x, y], α .. β )` creates a circular arc with radius *r* and center (*x*, *y*) with a polar angle between *α* and *β*.

`plot::Arc2d([ r_1, r_2], [x, y], α .. β )` creates a corresponding elliptical arc with semi-axes r_1, r_2.

The angle of a point on the arc is the usual polar angle to the positive *x*-axis known from polar coordinates. It is measured in radians.

If no range for the polar angle is specified, a full circle/ellipse is created.

If no center point is specified, an arc with center [0, 0] is created.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Angle`	rotation angle	`0`
`AngleEnd`	end of angle range	`PI/2`
`AngleBegin`	begin of angle range	`0`
`AngleRange`	angle range	`0 .. PI/2`
`AntiAliased`	antialiased lines and points?	`TRUE`

Attribute	Purpose	Default Value
`Center`	center of objects, rotation center	[0, 0]
`CenterX`	center of objects, rotation center, x-component	`0`
`CenterY`	center of objects, rotation center, y-component	`0`
`Closed`	open or closed polygons	`FALSE`
`Filled`	filled or transparent areas and surfaces	`FALSE`
`FillColor`	color of areas and surfaces	`RGB::Red`
`FillPattern`	type of area filling	`DiagonalLines`
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`LineColor`	color of lines	`RGB::Blue`
`LineWidth`	width of lines	`0.35`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LinesVisible`	visibility of lines	`TRUE`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	

Attribute	Purpose	Default Value
ParameterRange	range of the animation parameter	
SemiAxes	semi axes of ellipses and ellipsods	[1, 1]
SemiAxisX	first semi axis of ellipses and ellipsods	1
SemiAxisY	second semi axis of ellipses and ellipsods	1
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE

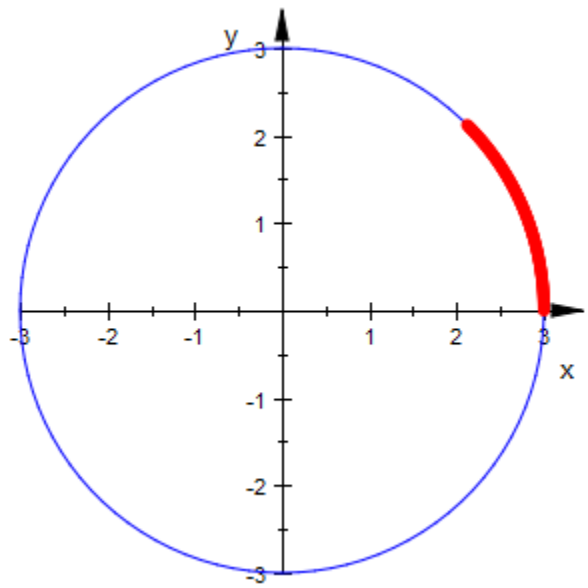
Attribute	Purpose	Default Value
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

An arc is a segment of a circle:

```
circle := plot::Circle2d(3, [0, 0]):
arc := plot::Arc2d(3, [0, 0], 0 .. PI/4, LineColor = RGB::Red,
                   LineWidth = 1.5*unit::mm):
plot(circle, arc)
```

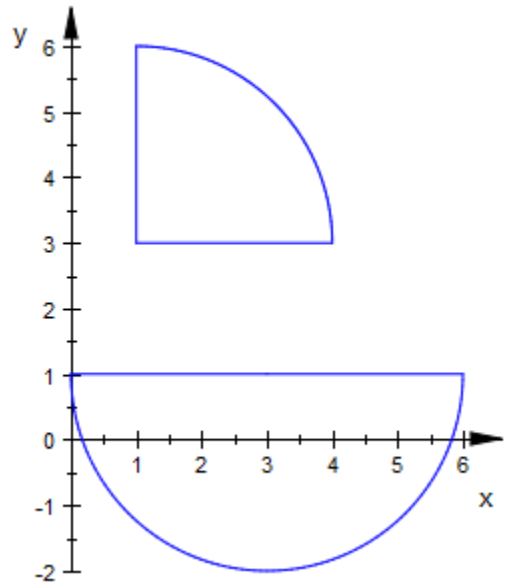



```
delete circle, arc:
```

Example 2

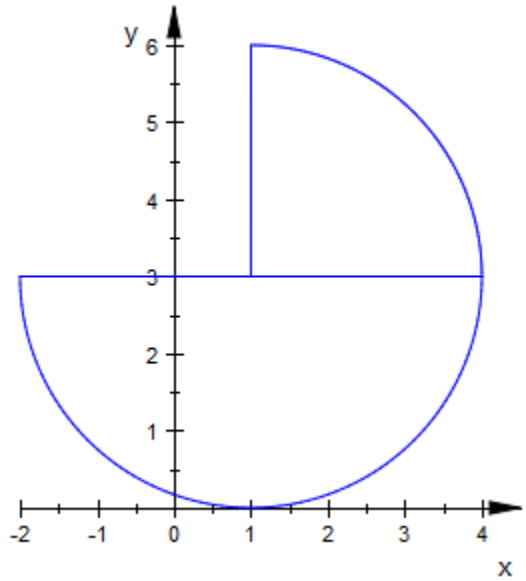
The center of an arc may be given as the second argument to `plot::Arc2d`:

```
arc1 := plot::Arc2d(3, [1, 3], 0..PI/2, Closed = TRUE):
arc2 := plot::Arc2d(3, [3, 1], -PI ..0, Closed = TRUE):
plot(arc1, arc2)
```



The center is accessible as the attribute `Center` of the arc object. We change the center of the second arc:

```
arc2::Center := [1, 3]:
plot(arc1, arc2)
```

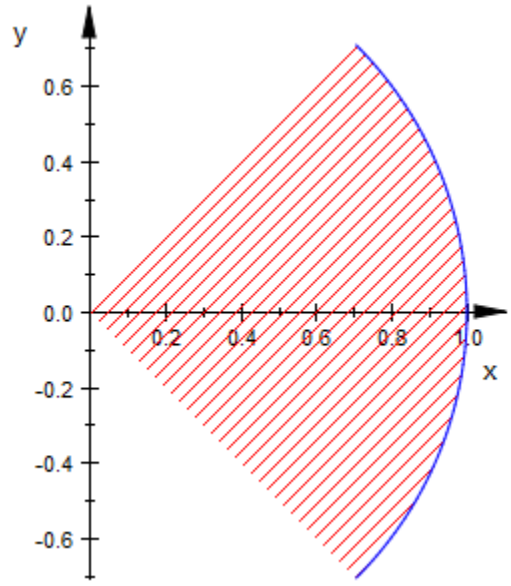



```
delete arc1, arc2:
```

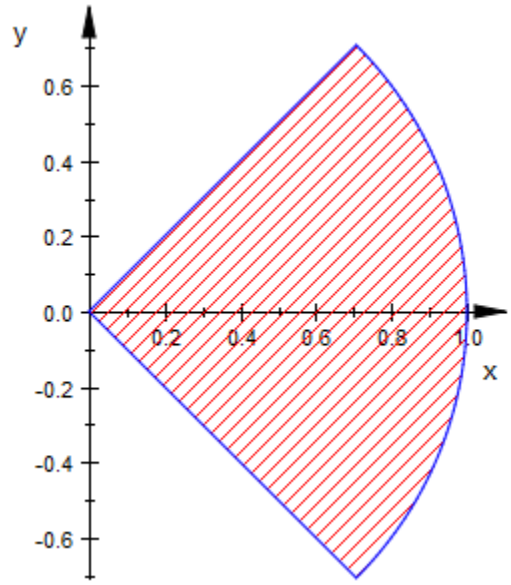
Example 3

A filled arc is a segment of a circle, like a piece of pie:

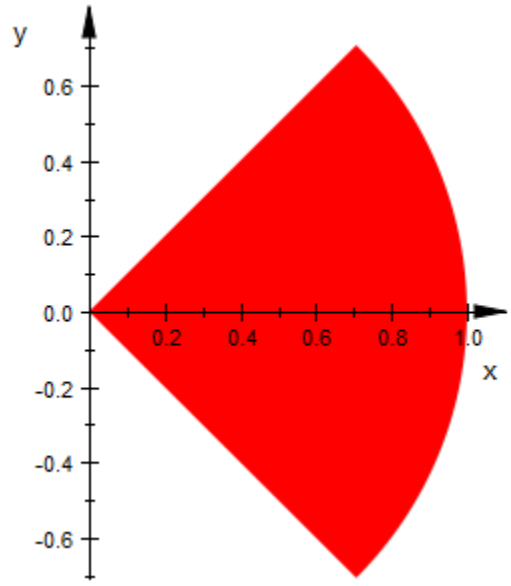
```
plot(plot::Arc2d(1, -PI/4..PI/4, Filled = TRUE))
```




```
plot(plot::Arc2d(1, -PI/4..PI/4, Filled = TRUE, Closed = TRUE))
```



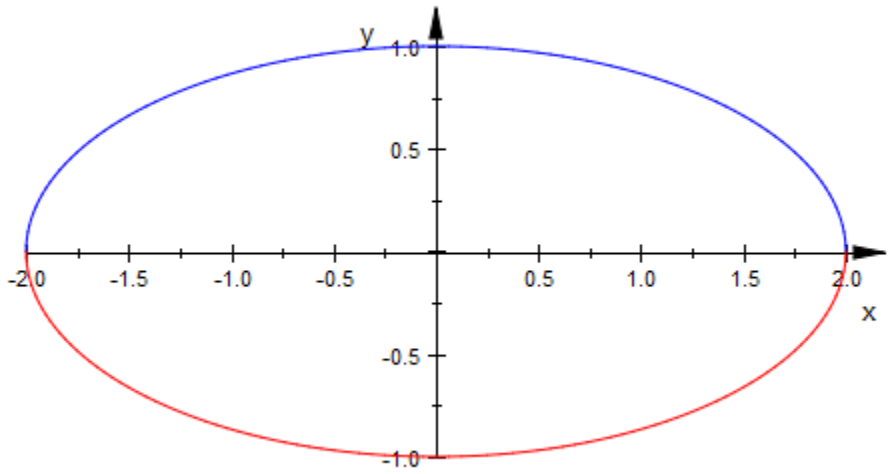

```
plot(plot::Arc2d(1, -PI/4..PI/4, Filled = TRUE,
                 FillPattern = Solid, LinesVisible = FALSE),
                 AxesInFront = TRUE)
```



Example 4

When giving a list of two radii, `plot::Arc2d` draws a segment of anellipse with the corresponding semi-axes:

```
arc1 := plot::Arc2d([2, 1], 0 .. PI, Color = RGB::Blue):
arc2 := plot::Arc2d([2, 1], -PI .. 0, Color = RGB::Red):
plot(arc1, arc2)
```

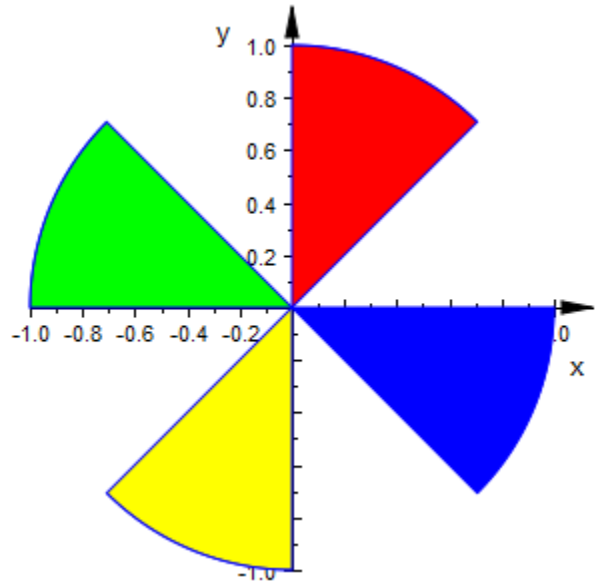



```
delete arc1, arc2:
```

Example 5

To plot or animate segments of a tilted ellipse, use the attribute `Angle`:

```
arc:= [1, 1], [0, 0], PI/4..PI/2, Filled, Closed, FillPattern=Solid:
plot(plot::Arc2d(arc, Angle=a+0,      a=0..2*PI, FillColor=RGB::Red),
     plot::Arc2d(arc, Angle=a+1/2*PI, a=0..2*PI, FillColor=RGB::Green),
     plot::Arc2d(arc, Angle=a+PI,     a=0..2*PI, FillColor=RGB::Yellow),
     plot::Arc2d(arc, Angle=a+3/2*PI, a=0..2*PI, FillColor=RGB::Blue))
```

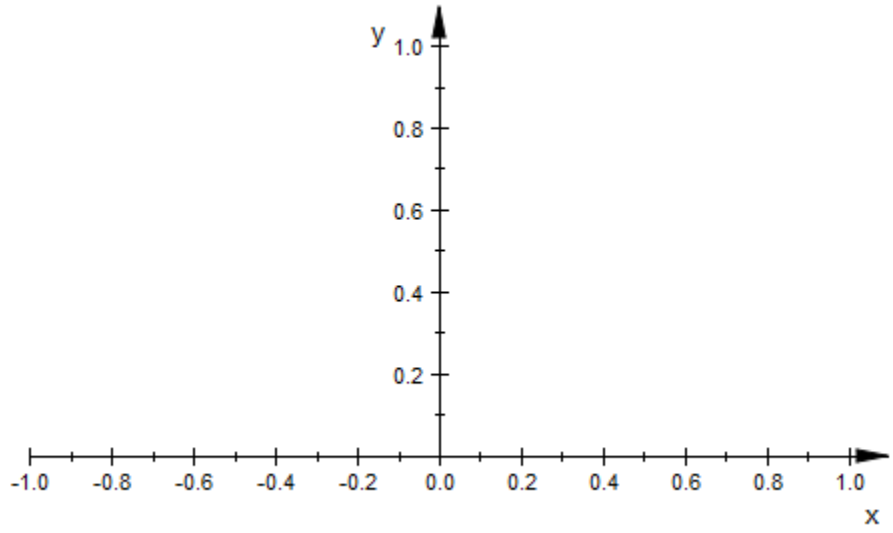



```
delete arc:
```

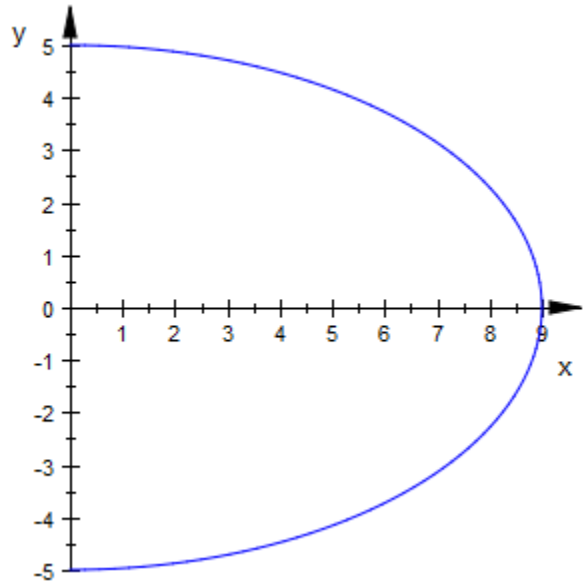
Example 6

Further examples of animated 2D arcs:

```
plot(plot::Arc2d(1, a .. PI, a = 0..PI))
```



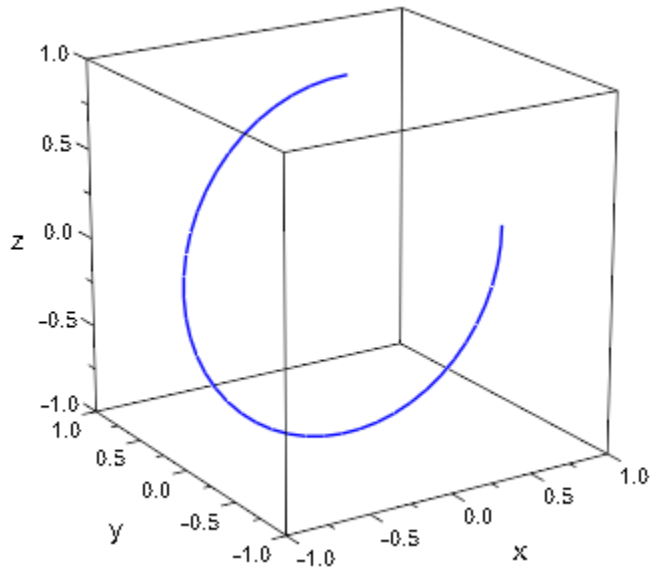

```
plot(plot::Arc2d([1 + a^2/2, 1 + a], -PI/2 .. PI/2, a = 0..4))
```



Example 7

We plot an animated 3D arc:

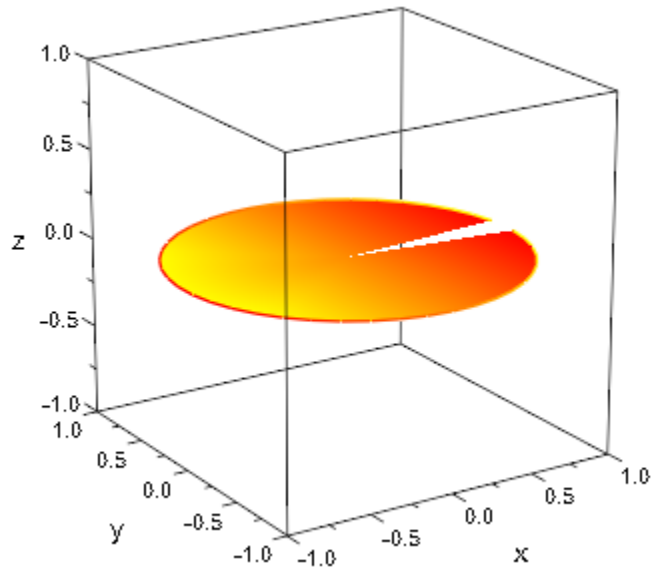
```
plot(plot::Arc3d(1, [0,0,0], [0,a,1-a], 0..3/2*PI, a = 0..1))
```



Example 8

We plot a colored 3D arcs:

```
plot(plot::Arc3d(1, [0,0,0], 0.1..2*PI-0.1, Filled,
                 LineColor=RGB::Yellow, LineColor2=RGB::Red,
                 LineColorType = Dichromatic, LineColorDirection=[+1,0,0],
                 FillColor=RGB::Yellow, FillColor2=RGB::Red,
                 FillColorType = Dichromatic, FillColorDirection=[-1,0,0]
))
```



Parameters

r

The radius of the circle. This must be a real numerical value or an arithmetical expression of the animation parameter `a`.

r is equivalent to the attributes `SemiAxisX`, `SemiAxisY`.

r_1, r_2

The semi-axes of an elliptical arc. They must be real numerical values or arithmetical expressions of the animation parameter `a`.

r_1, r_2 are equivalent to the attributes `SemiAxisX`, `SemiAxisY`.

c_x, c_y

The center point. The coordinates c_x, c_y must be real numerical values or arithmetical expressions of the animation parameter `a`. If no center is specified, an arc centered at the origin is created.

c_x, c_y are equivalent to the attribute `Center`.

α .. β

The angle range in radians: α and β must be real numerical values or arithmetical expressions of the animation parameter `a`. The default range is `0 .. 2*PI`.

α .. ß is equivalent to the attribute `AngleRange`.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Arc3d` | `plot::Circle2d` | `plot::Ellipse2d` | `plot::Ellipse3d`

plot::Arc3d

Circular and elliptical arcs in 3D

Syntax

```
plot::Arc3d(r, <[c_x, c_y, c_z], <[n_x, n_y, n_z]>>, <α .. β>, <a = a_min .. a_max>, options)
```

```
plot::Arc3d([r_1, r_2], <[c_x, c_y, c_z], <[n_x, n_y, n_z]>>, <α .. β>, <a = a_min .. a_max>, options
```

Description

`plot::Arc3d(r, [x, y, z], [nx, ny, nz], α .. β )` creates a circular arc with radius *r* and center (*x*, *y*, *z*) with a polar angle between *α* and *β* in the plane with the normal vector (*nx*, *ny*, *nz*).

`plot::Arc3d([ r_1, r_2], [x, y, z], [nx, ny, nz], α .. β )` creates a corresponding elliptical arc with semi-axes r_1, r_2.

The angle of a point on the arc is the usual polar angle to the positive *x*-axis known from polar coordinates. It is measured in radians.

If no range for the polar angle is specified, a full circle/ellipse is created.

If no center point is specified, an arc with center [0, 0, 0], is created.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Angle`	rotation angle	`0`
`AngleEnd`	end of angle range	`PI/2`
`AngleBegin`	begin of angle range	`0`
`AngleRange`	angle range	`0 .. PI/2`

Attribute	Purpose	Default Value
`Center`	center of objects, rotation center	[0, 0, 0]
`CenterX`	center of objects, rotation center, x-component	0
`CenterY`	center of objects, rotation center, y-component	0
`CenterZ`	center of objects, rotation center, z-component	0
`Closed`	open or closed polygons	`FALSE`
`Filled`	filled or transparent areas and surfaces	`FALSE`
`FillColor`	color of areas and surfaces	`RGB::LightBlue`
`FillColor2`	second color of areas and surfaces for color blends	`RGB::CornflowerBlue`
`FillColorType`	surface filling types	`Flat`
`FillColorDirection`	the direction of color transitions on surfaces	[0, 0, 1]
`FillColorDirectionX`	x-component of the direction of color transitions on surfaces	0
`FillColorDirectionY`	y-component of the direction of color transitions on surfaces	0
`FillColorDirectionZ`	z-component of the direction of color transitions on surfaces	1
`Frames`	the number of frames in an animation	50
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	

Attribute	Purpose	Default Value
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Blue
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
LineColorType	line coloring types	Flat
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	0
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Name	the name of a plot object (for browser and legend)	
Normal	normal vector of circles and discs, etc. in 3D	[0, 0, 1]
NormalX	normal vector of circles and discs, etc. in 3D, x-component	0
NormalY	normal vector of circles and discs, etc. in 3D, y-component	0
NormalZ	normal vector of circles and discs, etc. in 3D, z-component	1
ParameterEnd	end value of the animation parameter	

Attribute	Purpose	Default Value
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
SemiAxes	semi axes of ellipses and ellipsods	
SemiAxisX	first semi axis of ellipses and ellipsods	1
SemiAxisY	second semi axis of ellipses and ellipsods	1
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	

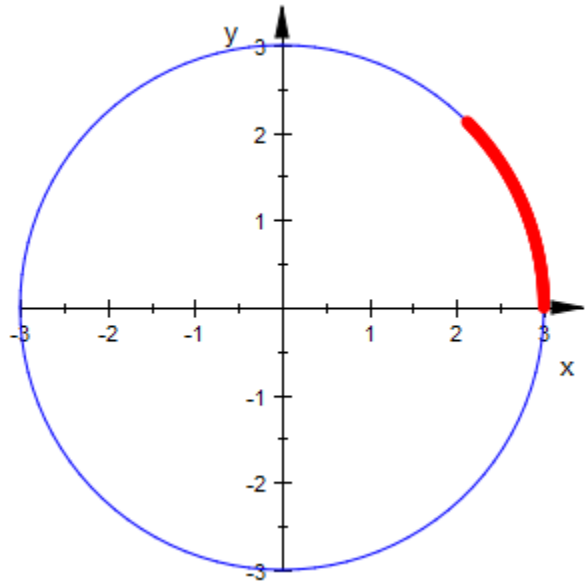
Attribute	Purpose	Default Value
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

An arc is a segment of a circle:

```
circle := plot::Circle2d(3, [0, 0]):
arc := plot::Arc2d(3, [0, 0], 0 .. PI/4, LineColor = RGB::Red,
                   LineWidth = 1.5*unit::mm):
plot(circle, arc)
```

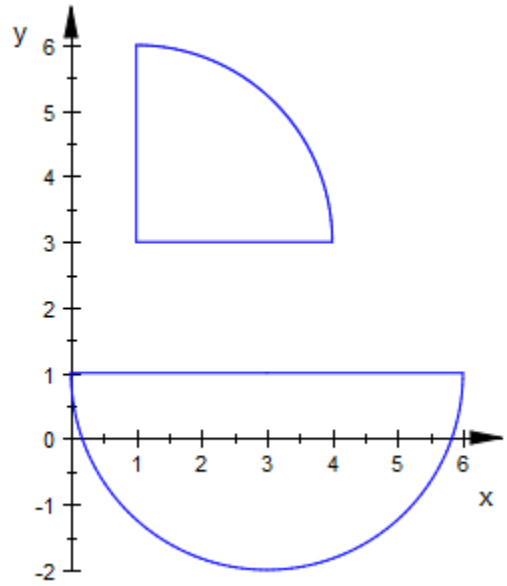



```
delete circle, arc:
```

Example 2

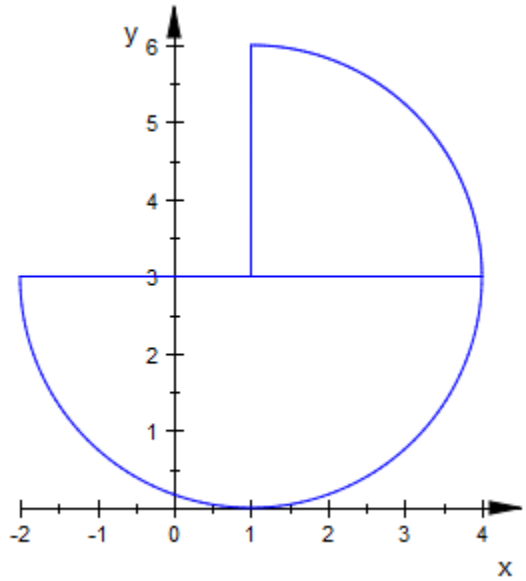
The center of an arc may be given as the second argument to `plot::Arc2d`:

```
arc1 := plot::Arc2d(3, [1, 3], 0..PI/2, Closed = TRUE):
arc2 := plot::Arc2d(3, [3, 1], -PI ..0, Closed = TRUE):
plot(arc1, arc2)
```



The center is accessible as the attribute `Center` of the arc object. We change the center of the second arc:

```
arc2::Center := [1, 3]:
plot(arc1, arc2)
```

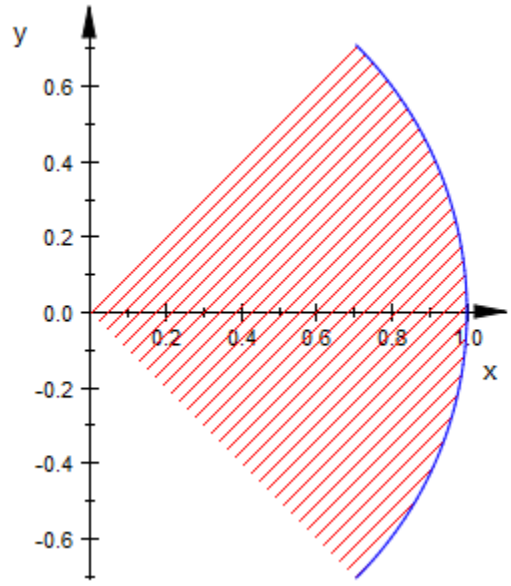



```
delete arc1, arc2:
```

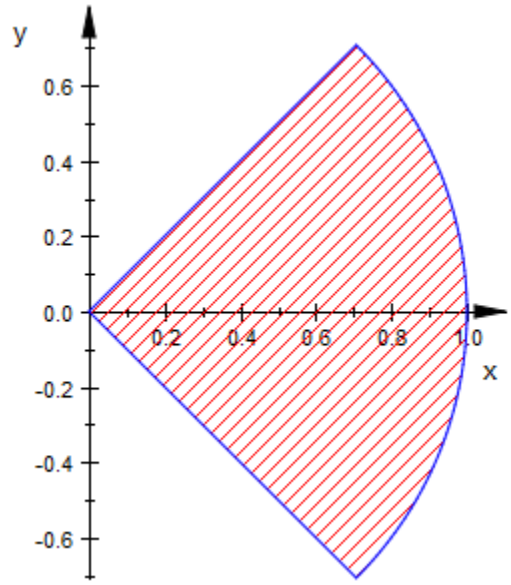
Example 3

A filled arc is a segment of a circle, like a piece of pie:

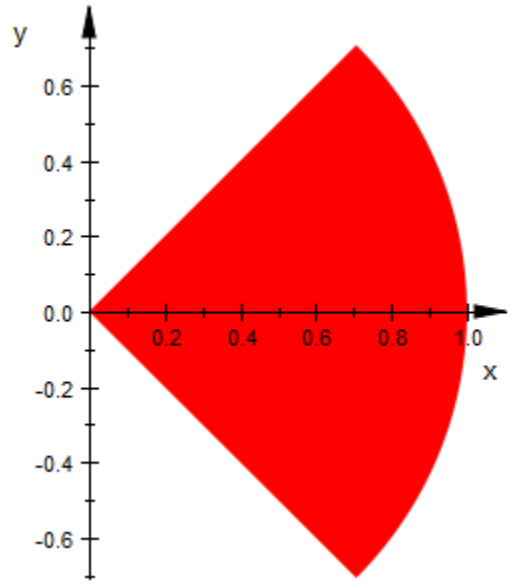
```
plot(plot::Arc2d(1, -PI/4..PI/4, Filled = TRUE))
```




```
plot(plot::Arc2d(1, -PI/4..PI/4, Filled = TRUE, Closed = TRUE))
```



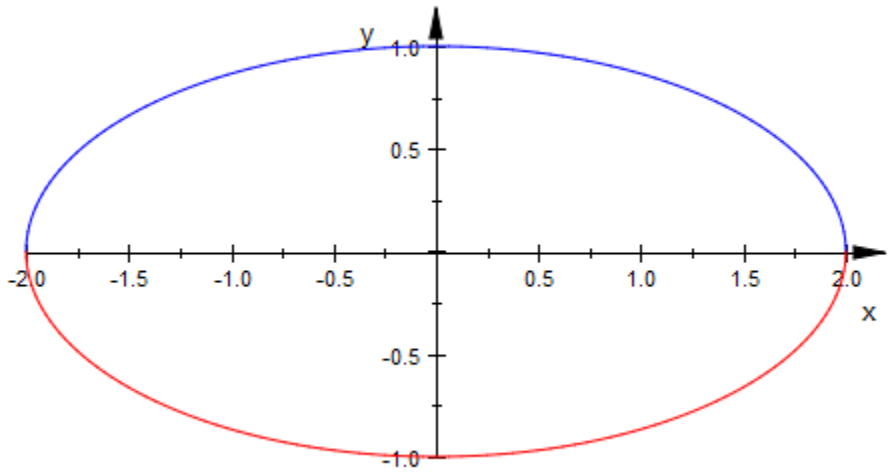

```
plot(plot::Arc2d(1, -PI/4..PI/4, Filled = TRUE,
                 FillPattern = Solid, LinesVisible = FALSE),
                 AxesInFront = TRUE)
```



Example 4

When giving a list of two radii, `plot::Arc2d` draws a segment of anellipse with the corresponding semi-axes:

```
arc1 := plot::Arc2d([2, 1], 0 .. PI, Color = RGB::Blue):
arc2 := plot::Arc2d([2, 1], -PI .. 0, Color = RGB::Red):
plot(arc1, arc2)
```

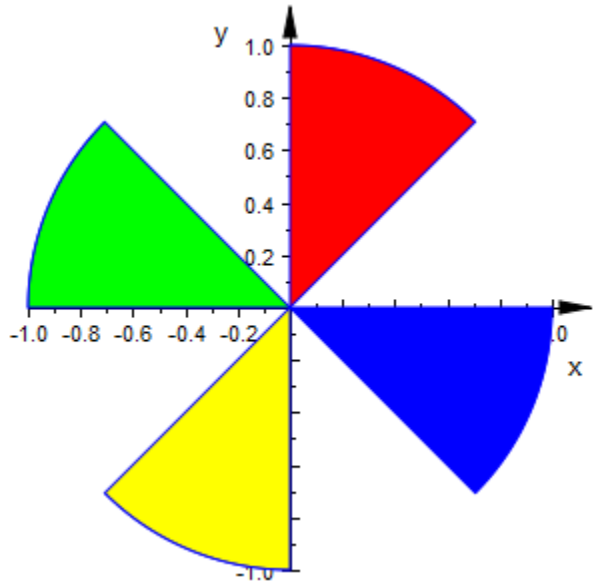



```
delete arc1, arc2:
```

Example 5

To plot or animate segments of a tilted ellipse, use the attribute `Angle`:

```
arc:= [1, 1], [0, 0], PI/4..PI/2, Filled, Closed, FillPattern=Solid:
plot(plot::Arc2d(arc, Angle=a+0,      a=0..2*PI, FillColor=RGB::Red),
     plot::Arc2d(arc, Angle=a+1/2*PI, a=0..2*PI, FillColor=RGB::Green),
     plot::Arc2d(arc, Angle=a+PI,     a=0..2*PI, FillColor=RGB::Yellow),
     plot::Arc2d(arc, Angle=a+3/2*PI, a=0..2*PI, FillColor=RGB::Blue))
```

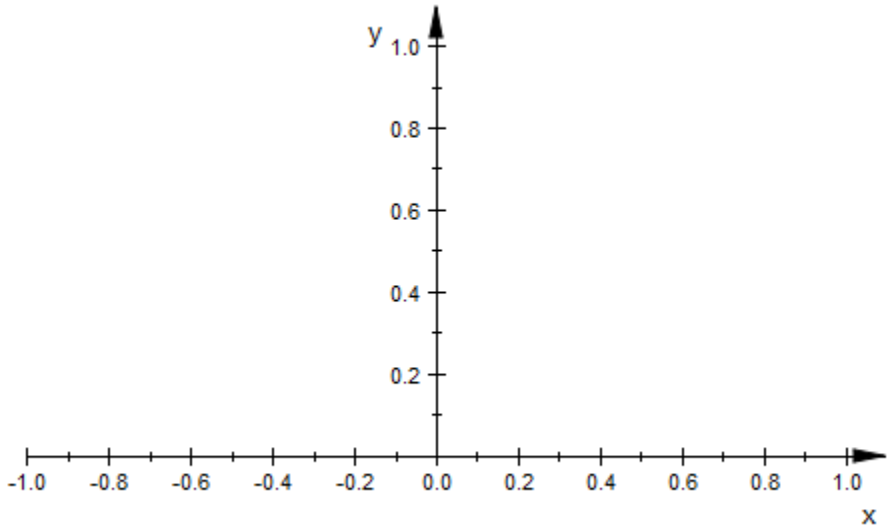



```
delete arc:
```

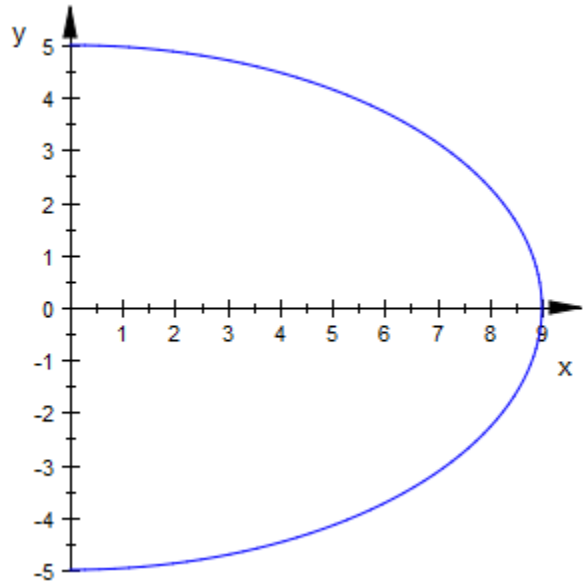
Example 6

Further examples of animated 2D arcs:

```
plot(plot::Arc2d(1, a .. PI, a = 0..PI))
```



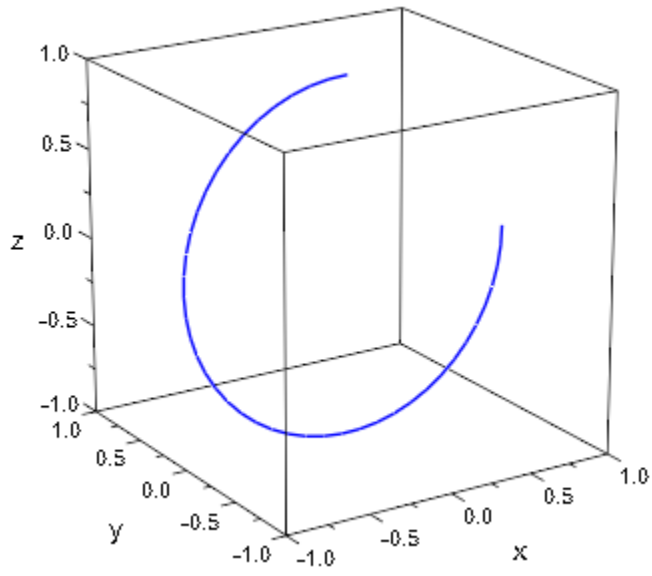

```
plot(plot::Arc2d([1 + a^2/2, 1 + a], -PI/2 .. PI/2, a = 0..4))
```



Example 7

We plot an animated 3D arc:

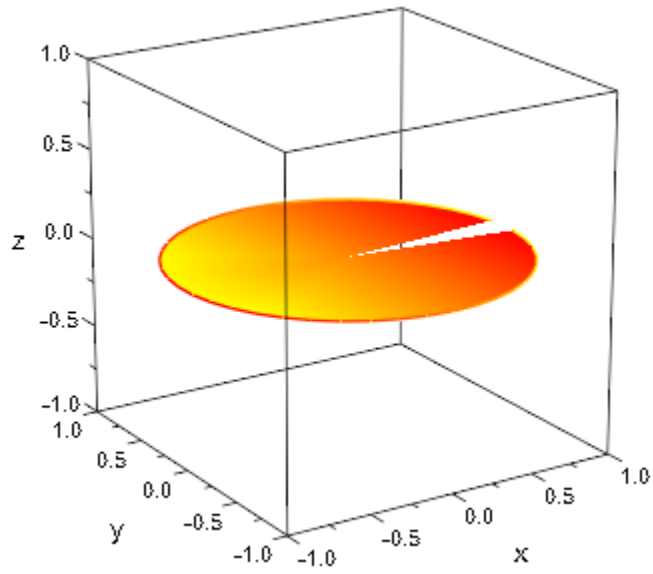
```
plot(plot::Arc3d(1, [0,0,0], [0,a,1-a], 0..3/2*PI, a = 0..1))
```



Example 8

We plot a colored 3D arcs:

```
plot(plot::Arc3d(1, [0,0,0], 0.1..2*PI-0.1, Filled,
                 LineColor=RGB::Yellow, LineColor2=RGB::Red,
                 LineColorType = Dichromatic, LineColorDirection=[+1,0,0],
                 FillColor=RGB::Yellow, FillColor2=RGB::Red,
                 FillColorType = Dichromatic, FillColorDirection=[-1,0,0]
))
```



Parameters

r

The radius of the circle. This must be a real numerical value or an arithmetical expression of the animation parameter `a`.

r is equivalent to the attributes `SemiAxisX`, `SemiAxisY`.

r_1, r_2

The semi-axes of an elliptical arc. They must be real numerical values or arithmetical expressions of the animation parameter `a`.

r_1, r_2 are equivalent to the attributes `SemiAxisX`, `SemiAxisY`.

c_x, c_y, c_z

The center point. The coordinates c_x, c_y, c_z must be real numerical values or arithmetical expressions of the animation parameter `a`. If no center is specified, an arc centered at the origin is created.

c_x, c_y, c_z are equivalent to the attribute `Center`.

n_x, n_y, n_z

The normal vector. The coordinates n_x, n_y, n_z must be real numerical values or arithmetical expressions of the animation parameter `a`. If no normal vector is specified, the arc is created in the xy-plane.

n_x, n_y, n_z are equivalent to the attribute `Normal`.

α .. β

The angle range in radians: α and β must be real numerical values or arithmetical expressions of the animation parameter `a`. The default range is `0 .. 2*PI`.

α .. ß is equivalent to the attribute `AngleRange`.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Arc2d` | `plot::Circle2d` | `plot::Ellipse2d` | `plot::Ellipse3d`

plot::Arrow2d

2D arrows

Syntax

```
plot::Arrow2d(<[x1, y1]>, [x2, y2], <a = amin .. amax>, options)
```

Description

`plot::Arrow2d([ x1, y1], [ x2, y2])` creates a 2D arrow from the point (x_1, y_1) to the point (x_2, y_2).

`plot::Arrow2d([ x2, y2])` creates a 2D arrow from the point (0, 0) to the point (x_2, y_2).

The points defining an arrow can also be passed as vectors.

The appearance of arrows can be controlled by various attributes:

- `Color` sets the color.
- `LineWidth` and `LineStyle` set the width and the style (solid, dashed, dotted).
- `TipLength`, `TipAngle`, and `TipStyle` control the arrow tip.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`
`Color`	the main color	`RGB::Blue`
`Frames`	the number of frames in an animation	`50`
`From`	starting point of arrows and lines	`[0, 0]`

Attribute	Purpose	Default Value
FromX	starting point of arrows and lines, x-coordinate	0
FromY	starting point of arrows and lines, y-coordinate	0
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Blue
LineWidth	width of lines	0.35
LineStyle	solid, dashed or dotted lines?	Solid
LineColorDirection	the direction of color transitions on lines	[0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	1
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0

Attribute	Purpose	Default Value
TimeRange	the real time span of an animation	0.0 .. 10.0
TipAngle	opening angle of arrow heads	(2*PI)/15
TipStyle	presentation style of arrow heads	Filled
TipLength	length of arrow heads	4
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
To	end point of arrows and lines	[1, 0]
ToX	end point of arrows and lines, x-coordinate	1
ToY	end point of arrows and lines, y-coordinate	0
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE

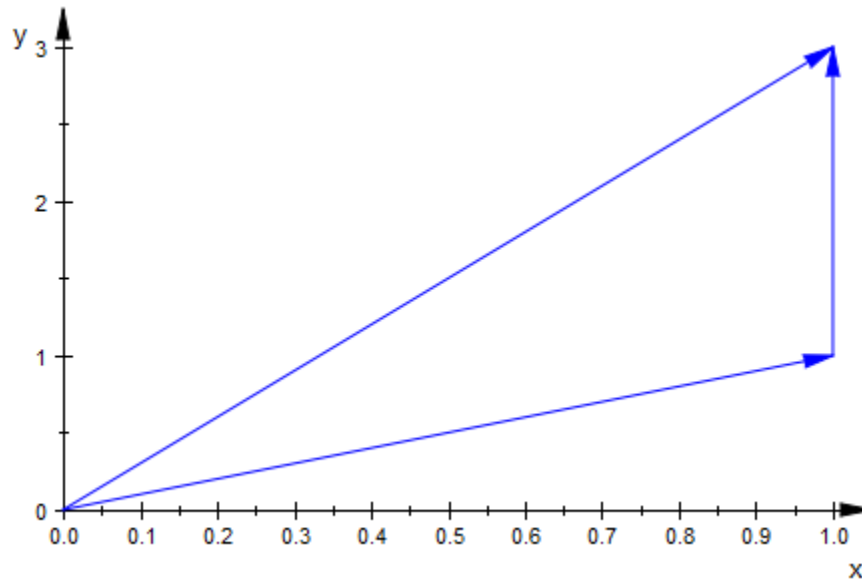
Attribute	Purpose	Default Value
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

We create and plot some arrows:

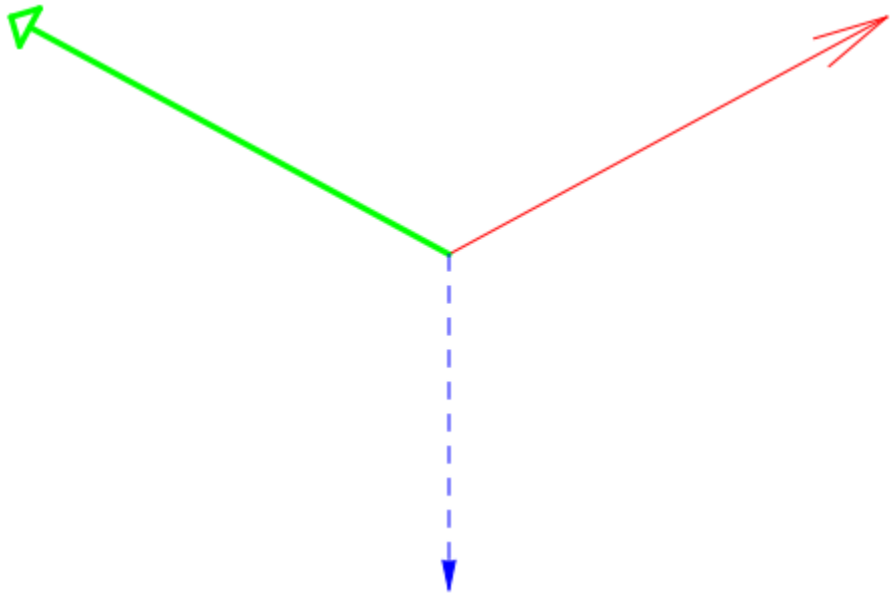
```
plot(plot::Arrow2d([1, 1]), plot::Arrow2d([1, 3]),
     plot::Arrow2d([1, 1], [1, 3]))
```



Various attributes are available to control the presentation style of an arrow:

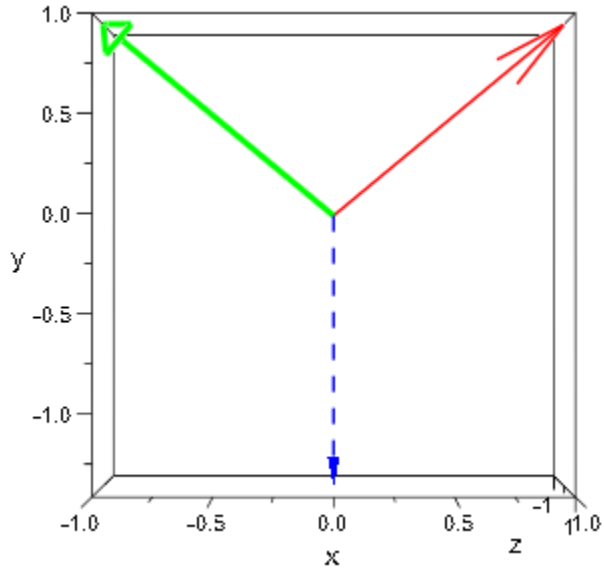
```
plot(plot::Arrow2d([1, 1], Color = RGB::Red,
                   TipStyle = Open, TipLength = 10*unit::mm),
```

```
plot::Arrow2d([-1, 1], Color = RGB::Green,
              LineWidth = 0.8*unit::mm,
              TipStyle = Closed, TipAngle = PI/2),
plot::Arrow2d([0, -sqrt(2)], Color = RGB::Blue,
              LineStyle = Dashed),
Axes = None)
```



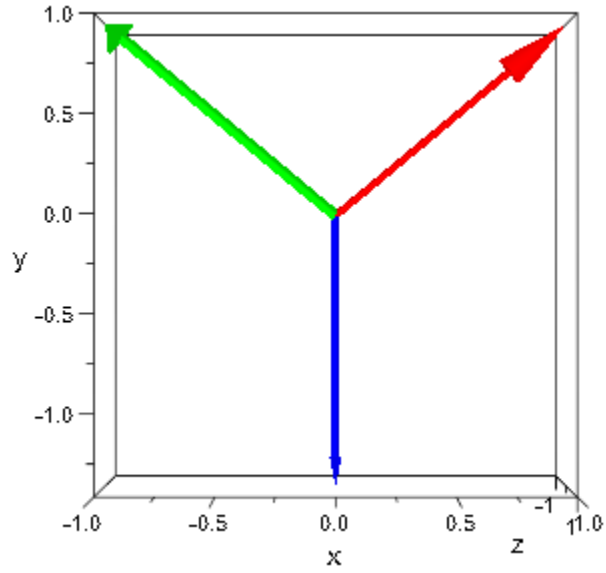
Here are corresponding arrows in 3D:

```
plot(plot::Arrow3d([1, 1, 0], Color = RGB::Red,
                   TipStyle = Open, TipLength = 10*unit::mm),
     plot::Arrow3d([-1, 1, 0], Color = RGB::Green,
                   LineWidth = 0.8*unit::mm,
                   TipStyle = Closed, TipAngle = PI/2),
     plot::Arrow3d([0, -sqrt(2), 0], Color = RGB::Blue,
                   LineStyle = Dashed),
     CameraDirection = [0, -1, 1000])
```



We use `Tubular` = `TRUE`:

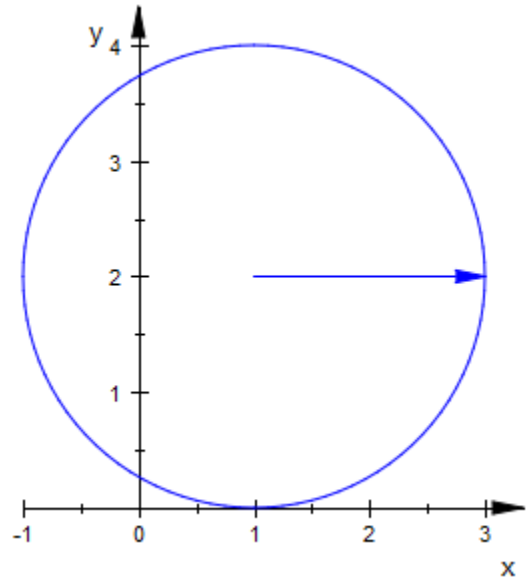
```
plot(plot::Arrow3d([1, 1, 0], Color = RGB::Red,
                   TipLength = 10*unit::mm),
     plot::Arrow3d([-1, 1, 0], Color = RGB::Green,
                   TubeDiameter = 1.5*unit::mm,
                   TipAngle = PI/2),
     plot::Arrow3d([0, -sqrt(2), 0], Color = RGB::Blue),
     Tubular = TRUE, CameraDirection = [0, -1, 1000])
```



Example 2

We plot an arrow with fixed starting point and animated end point:

```
plot(plot::Circle2d(2, [1, 2]),
     plot::Arrow2d([1, 2], [1 + 2*cos(a), 2 + 2*sin(a)],
                   a = 0..2*PI))
```



Parameters

x_1, y_1

The coordinates of the starting point: real numerical values or arithmetical expressions of the animation parameter `a`. If no starting point is specified, an arrow starting at the origin is created.

x_1, y_1 are equivalent to the attributes `FromX`, `FromY`.

x_2, y_2

The coordinates of the end point: real numerical values or arithmetical expressions of the animation parameter `a`.

x_2, y_2 are equivalent to the attributes `ToX`, `ToY`.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Arrow3d` | `plot::Line2d` | `plot::Line3d` | `plot::VectorField2d`

plot::Arrow3d

3D arrows

Syntax

```
plot::Arrow3d(<[x1, y1, z1]>, [x2, y2, z2], <a = amin .. amax>, options)
```

Description

`plot::Arrow3d([ x1, y1, z1], [ x2, y2, z2])` creates a 3D arrow from the point (x_1, y_1, z_1) to the point (x_2, y_2, z_2).

`plot::Arrow3d([ x2, y2, z2])` creates a 3D arrow from the point (0, 0, 0) to the point (x_2, y_2, z_2).

The points defining an arrow can also be passed as vectors.

The appearance of arrows can be controlled by various attributes:

- `Color` sets the color.
- `LineWidth` and `LineStyle` set the width and the style (solid, dashed, dotted).
- `TipLength`, `TipAngle`, and `TipStyle` control the arrow tip.
- With `Tubular` = `TRUE`, 3D arrows are rendered as 3D tubes with a diameter set by `TubeDiameter`. The arrow head is rendered as a cone.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Color`	the main color	`RGB::Blue`
`Frames`	the number of frames in an animation	`50`

Attribute	Purpose	Default Value
`From`	starting point of arrows and lines	`[0, 0, 0]`
`FromX`	starting point of arrows and lines, x-coordinate	`0`
`FromY`	starting point of arrows and lines, y-coordinate	`0`
`FromZ`	starting point of arrows and lines, z-coordinate	`0`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`LineColor`	color of lines	`RGB::Blue`
`LineWidth`	width of lines	`0.35`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LineColorDirection`	the direction of color transitions on lines	`[0, 0, 1]`
`LineColorDirectionX`	x-component of the direction of color transitions on lines	`0`
`LineColorDirectionY`	y-component of the direction of color transitions on lines	`0`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	

Attribute	Purpose	Default Value
ParameterRange	range of the animation parameter	
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
TipAngle	opening angle of arrow heads	(2*PI)/15
TipStyle	presentation style of arrow heads	Filled
TipLength	length of arrow heads	4
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
To	end point of arrows and lines	[1, 0, 0]
ToX	end point of arrows and lines, x-coordinate	1
ToY	end point of arrows and lines, y-coordinate	0
ToZ	end point of arrows and lines, z-coordinate	0

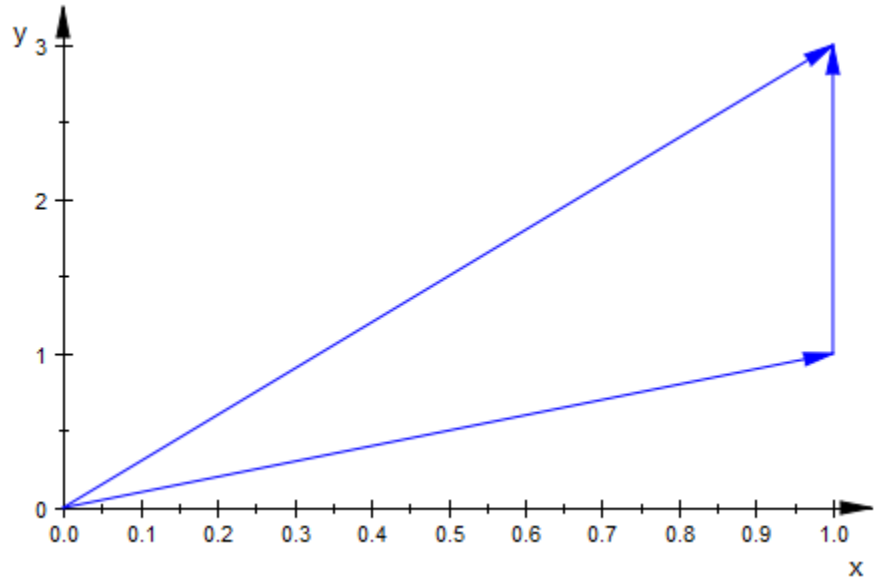
Attribute	Purpose	Default Value
`Tubular`	display 3D arrows and lines as tubes?	`FALSE`
`TubeDiameter`	diameter of tubular arrows and lines.	`1.0`
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

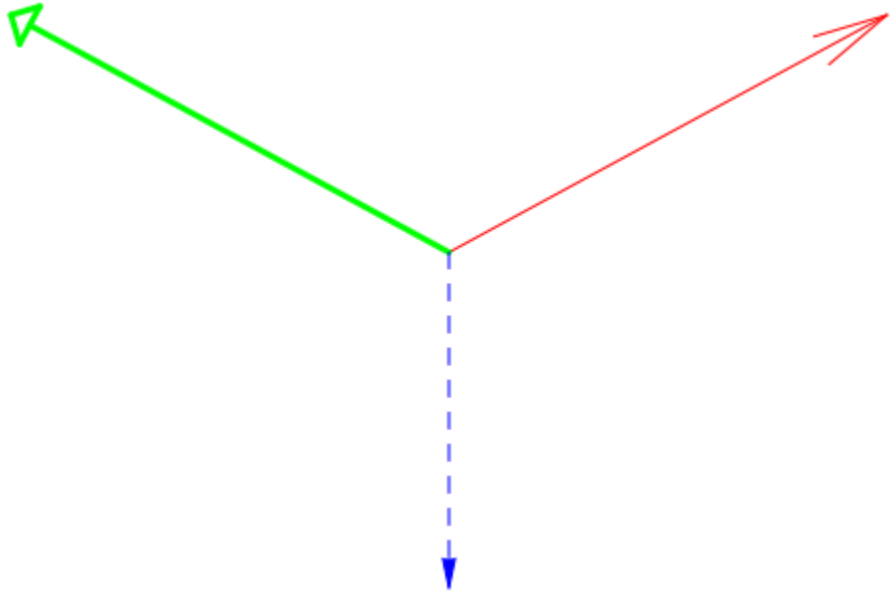
We create and plot some arrows:

```
plot(plot::Arrow2d([1, 1]), plot::Arrow2d([1, 3]),
     plot::Arrow2d([1, 1], [1, 3]))
```



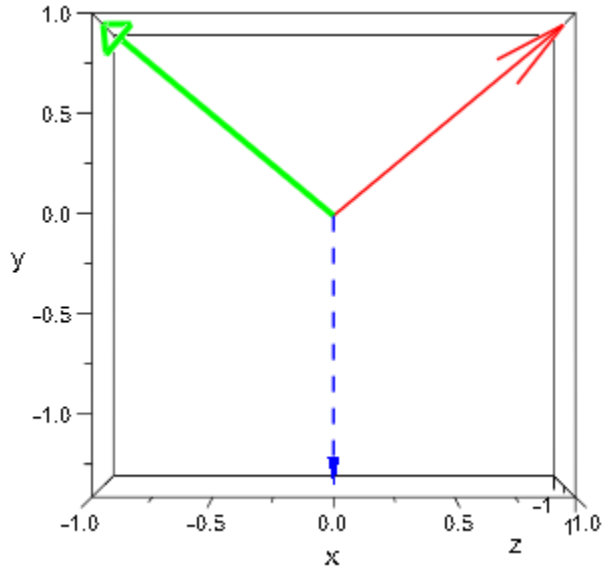
Various attributes are available to control the presentation style of an arrow:

```
plot(plot::Arrow2d([1, 1], Color = RGB::Red,
                   TipStyle = Open, TipLength = 10*unit::mm),
     plot::Arrow2d([-1, 1], Color = RGB::Green,
                   LineWidth = 0.8*unit::mm,
                   TipStyle = Closed, TipAngle = PI/2),
     plot::Arrow2d([0, -sqrt(2)], Color = RGB::Blue,
                   LineStyle = Dashed),
     Axes = None)
```



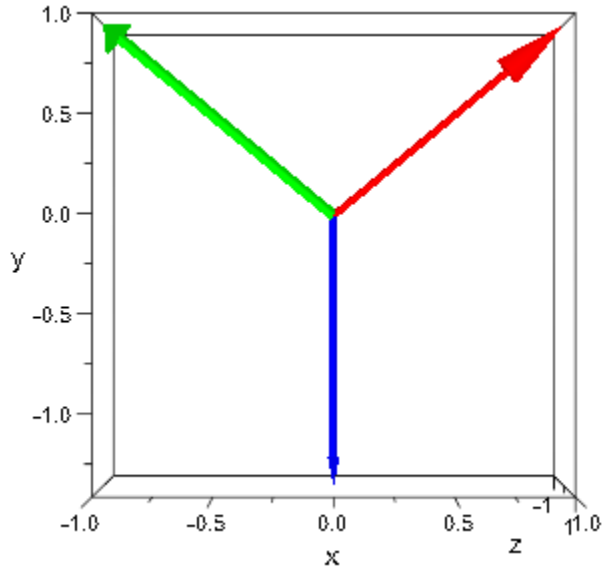
Here are corresponding arrows in 3D:

```
plot(plot::Arrow3d([1, 1, 0], Color = RGB::Red,
                   TipStyle = Open, TipLength = 10*unit::mm),
     plot::Arrow3d([-1, 1, 0], Color = RGB::Green,
                   LineWidth = 0.8*unit::mm,
                   TipStyle = Closed, TipAngle = PI/2),
     plot::Arrow3d([0, -sqrt(2), 0], Color = RGB::Blue,
                   LineStyle = Dashed),
     CameraDirection = [0, -1, 1000])
```



We use `Tubular` = `TRUE`:

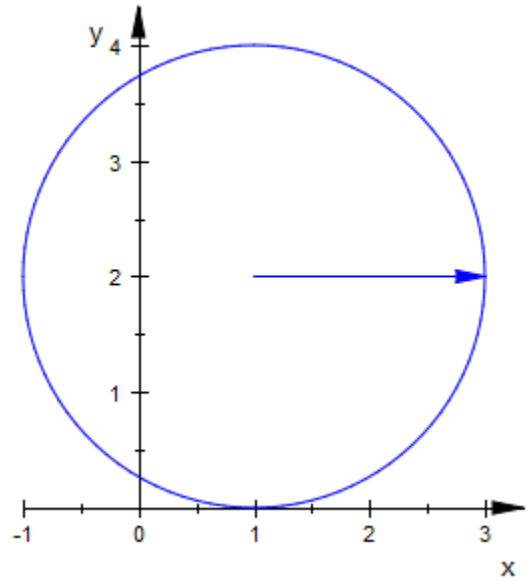
```
plot(plot::Arrow3d([1, 1, 0], Color = RGB::Red,
                   TipLength = 10*unit::mm),
     plot::Arrow3d([-1, 1, 0], Color = RGB::Green,
                   TubeDiameter = 1.5*unit::mm,
                   TipAngle = PI/2),
     plot::Arrow3d([0, -sqrt(2), 0], Color = RGB::Blue),
     Tubular = TRUE, CameraDirection = [0, -1, 1000])
```



Example 2

We plot an arrow with fixed starting point and animated end point:

```
plot(plot::Circle2d(2, [1, 2]),
     plot::Arrow2d([1, 2], [1 + 2*cos(a), 2 + 2*sin(a)],
                   a = 0..2*PI))
```



Parameters

x_1, y_1, z_1

The coordinates of the starting point: real numerical values or arithmetical expressions of the animation parameter `a`. If no starting point is specified, an arrow starting at the origin is created.

x_1, y_1, z_1 are equivalent to the attributes `FromX`, `FromY`, `FromZ`.

x_2, y_2, z_2

The coordinates of the end point: real numerical values or arithmetical expressions of the animation parameter `a`.

x_2, y_2, z_2 are equivalent to the attributes `ToX`, `ToY`, `ToZ`.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Arrow2d` | `plot::Line2d` | `plot::Line3d` | `plot::VectorField2d`

plot::Bars2d

2D bar chart

Syntax

`plot::Bars2d([[`a_1`, `a_2`, …], [`b_1`, `b_2`, …], …], <a = `a_{min}` .. `a_{max}`>, options)`

`plot::Bars2d([`a_1`, `a_2`, …], <a = `a_{min}` .. `a_{max}`>, options)`

Description

`plot::Bars2d([[ `a_1`, `a_2`, ...], [ `b_1`, `b_2`, ...], ...])` generates a bar chart with bar heights `$a_1$`, `$b_1$`, ..., `$a_2$`, `$b_2$`, ...

`plot::Bars2d([ `a_1`, `a_2`, ...])` creates a bar chart with bars of height `$a_1$`, `$a_2$`, …

With `plot::Bars2d([[ `a_1`, `a_2`,...], [ `b_1`, `b_2`,...], ...])`, bars are plotted in the order `$a_1$`, `$b_1$`, …, a gap, `$a_2$`, `$b_2$`, … Cf. “Example 2” on page 24-160.

The horizontal positions and the widths of the bars may be controlled by the attributes `BarCenters` and `BarWidths`, respectively.

The attribute `GroupStyle` provides grouping options.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`FALSE`
`BarStyle`	display style of bar plots	`Boxes`
`BarWidths`	widths of bars	[[1.0]]
`BarCenters`	position of bars	
`Color`	the main color	

Attribute	Purpose	Default Value
Colors	list of colors to use	[RGB::Blue, RGB::Red, RGB::Green, RGB::MuPADGold, RGB::Orange, RGB::Cyan, RGB::Magenta, RGB::LimeGreen, RGB::CadmiumYellowLight, RGB::AlizarinCrimson, RGB::Aqua, RGB::Lavender, RGB::SeaGreen, RGB::AureolineYellow, RGB::Banana, RGB::Beige, RGB::YellowGreen, RGB::Wheat, RGB::IndianRed, RGB::Black]
Data	the (statistical) data to plot	
DrawMode	orientation of boxes and bars	Vertical
Filled	filled or transparent areas and surfaces	TRUE
FillPatterns	list of area fill types	[Solid]
Frames	the number of frames in an animation	50
GroupStyle	grouping options in 2D bar plots	MultipleBars
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Black
LineWidth	width of lines	0.35

Attribute	Purpose	Default Value
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
Shadows	display “shadows” for bar plots?	FALSE
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	

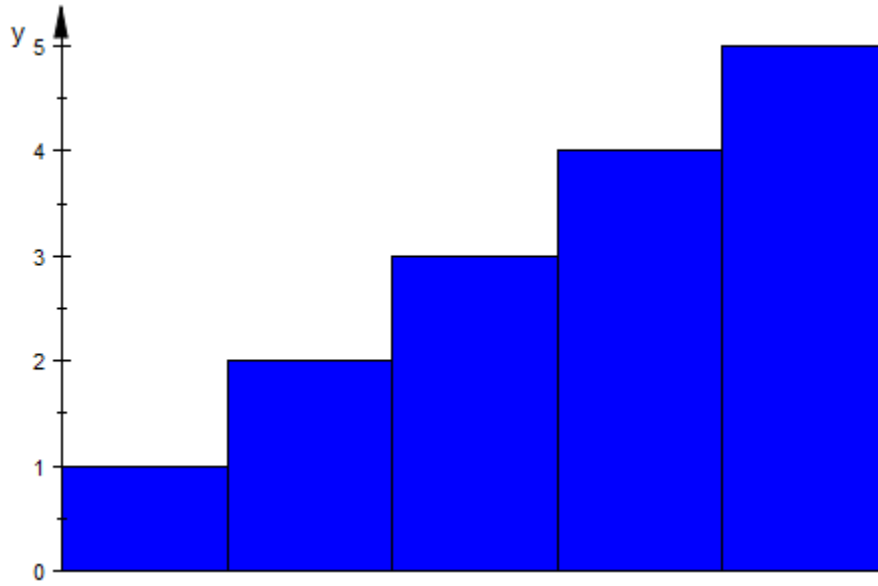
Attribute	Purpose	Default Value
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

Given a single list of values, `plot::Bars2d` plots bars of the corresponding height, filled solidly in one color:

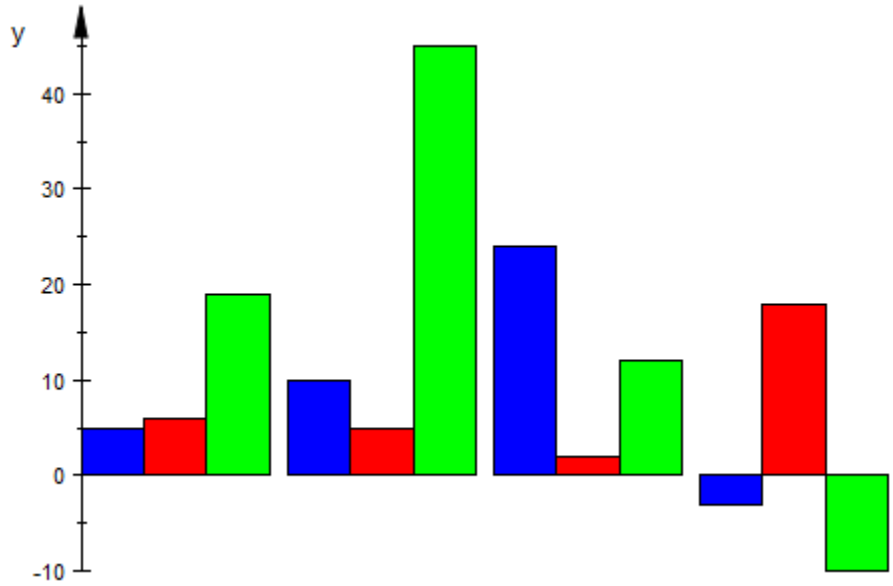
```
plot(plot::Bars2d([1, 2, 3, 4, 5]))
```



Example 2

When asked to plot a list of lists of values, `plot::Bars2d` will group the first entries of all lists, the second entries and so on, with a small gap between the groups:

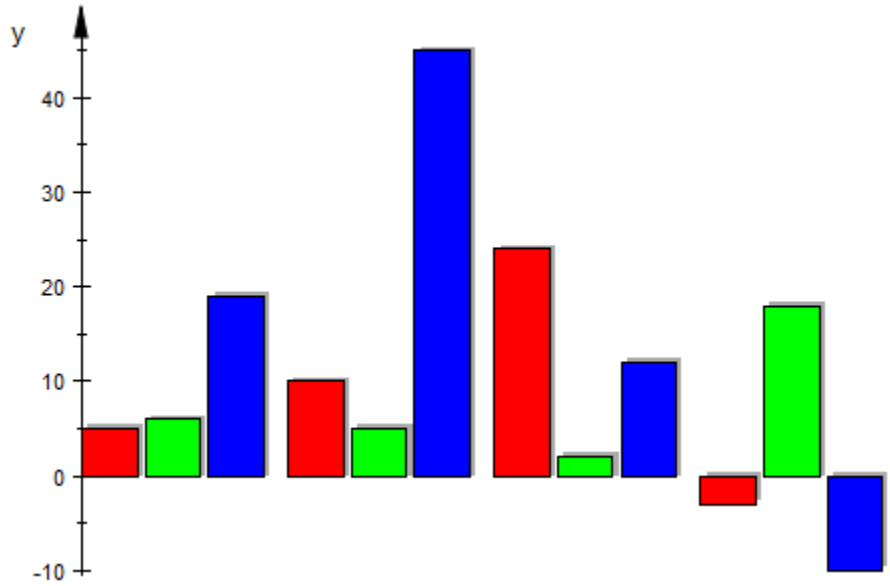
```
plot(plot::Bars2d([[ 5, 10, 24, -3],
                   [ 6,  5,  2, 18],
                   [19, 45, 12,-10]]))
```



Example 3

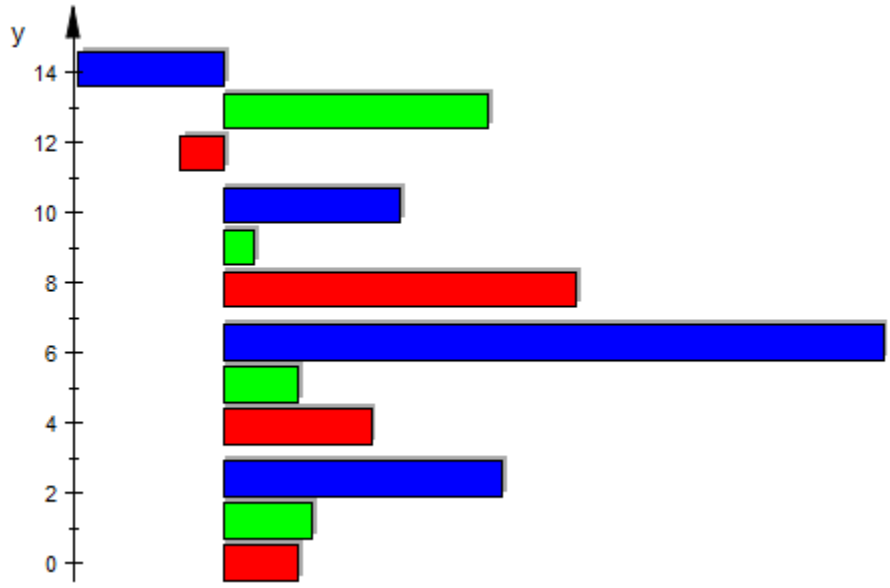
The appearance of the plots can be controlled with a number of attributes. For example, `Colors` accepts a list of colors for the bars and `Shadows` switches on "shadows," giving a slight impression of depth:

```
plot(plot::Bars2d([[ 5, 10, 24,  -3],
                   [ 6,  5,  2,  18],
                   [19, 45, 12, -10]],
                  Colors = [RGB::Red, RGB::Green, RGB::Blue],
                  Shadows = TRUE))
```



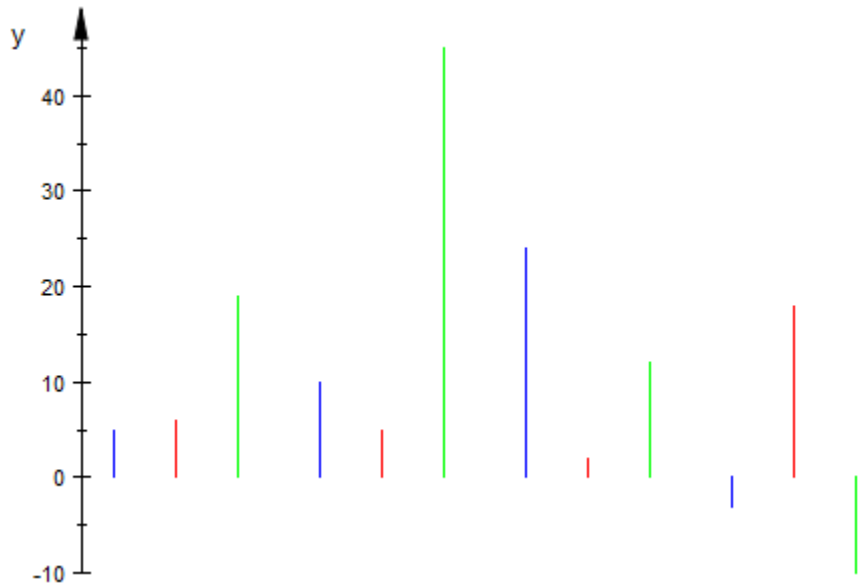
Using `DrawMode`, `plot::Bars2d` can be made to draw horizontal bars instead of vertical ones:

```
plot(plot::Bars2d([[ .5, 1.0, 2.4,  -.3],
                   [ .6,  .5,  .2,  1.8],
                   [1.9, 4.5, 1.2, -1.0]],
                  Colors = [RGB::Red,RGB::Green,RGB::Blue],
                  Shadows = TRUE, DrawMode = Horizontal))
```



`BarStyle` is used to plot points or lines instead of rectangles:

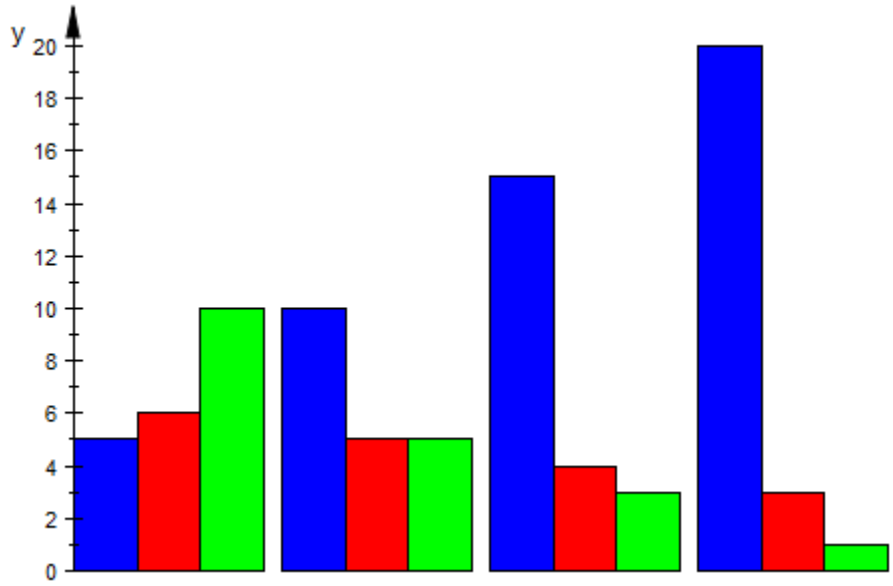
```
plot(plot::Bars2d([[ 5, 10, 24,  -3],
                   [ 6,  5,  2,  18],
                   [19, 45, 12, -10]], BarStyle = Lines))
```



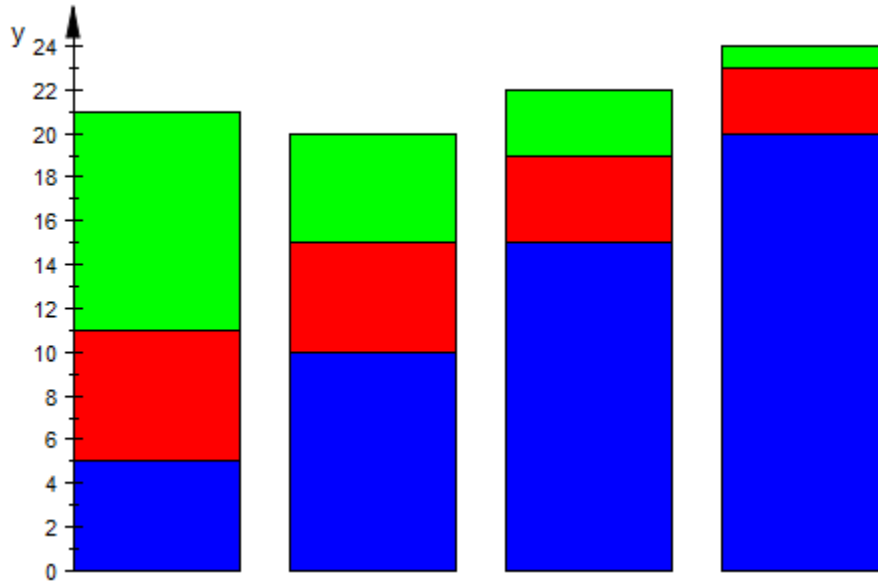
Example 4

We demonstrate alternative `grouping styles`:

```
plot(plot::Bars2d([[ 5, 10, 15, 20],
                   [ 6,  5,  4,  3],
                   [10,  5,  3,  1]], GroupStyle = MultipleBars))
```



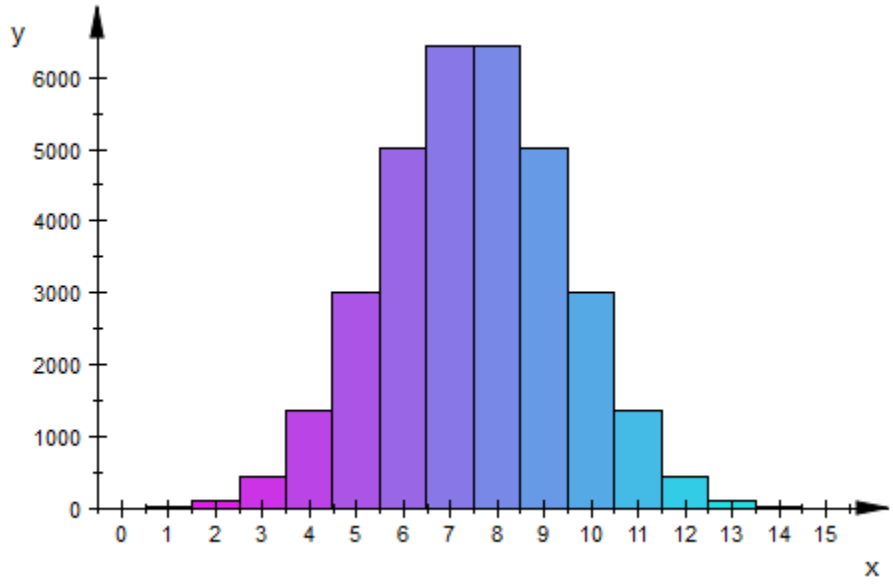

```
plot(plot::Bars2d([[ 5, 10, 15, 20],
                   [ 6,  5,  4,  3],
                   [10,  5,  3,  1]], GroupStyle = SingleBars))
```



Example 5

To plot a single group of data with different colors, they must be placed in individual lists:

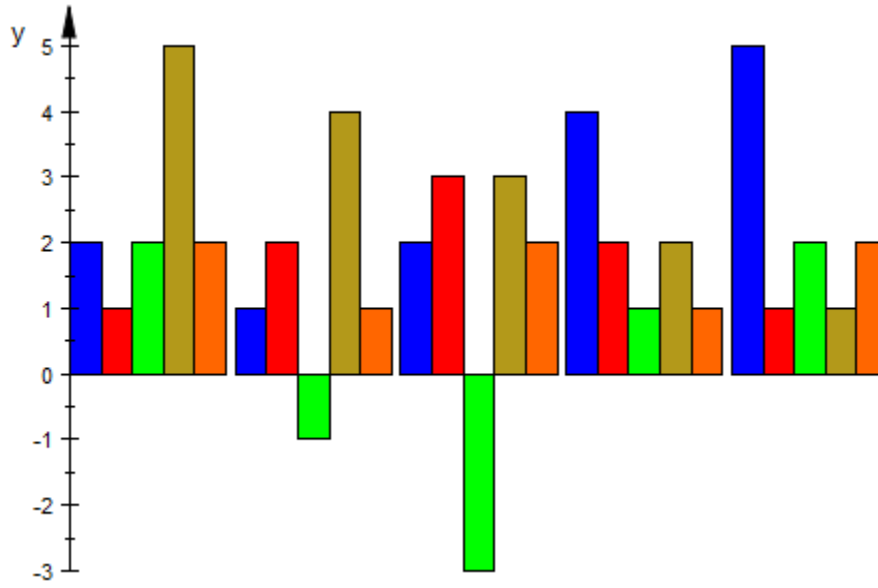
```
plot(plot::Bars2d([[binomial(15,i)] $ i = 0..15],
                  Colors = [[1-j/15, j/15, 0.9] $ j = 0..15]),
             XAxisVisible)
```



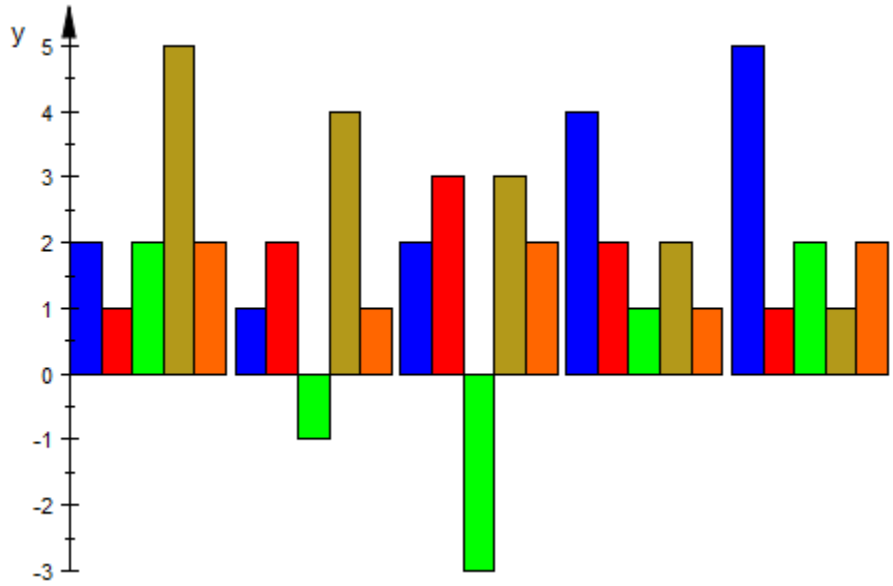
Example 6

`plot::Bars2d` accepts input in form of lists (as above), as a matrix, or as a one- or two-dimensional array:

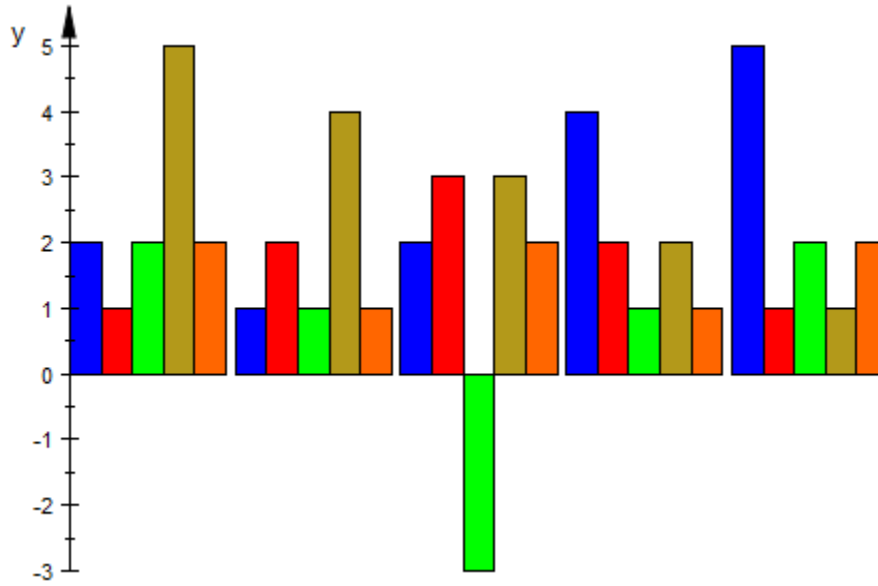
```
L := [ [2,  1,  2, 4, 5],
       [1,  2,  3, 2, 1],
       [2, -1, -3, 1, 2],
       [5,  4,  3, 2, 1],
       [2,  1,  2, 1, 2]]:
M := matrix(L):
A :=array(1..5, 1..5,
          (1,1) = 2, (1,2) = 1, (1,3) =  2, (1,4) = 4, (1,5) = 5,
          (2,1) = 1, (2,2) = 2, (2,3) =  3, (2,4) = 2, (2,5) = 1,
          (3,1) = 2, (3,2) = 1, (3,3) = -3, (3,4) = 1, (3,5) = 2,
          (4,1) = 5, (4,2) = 4, (4,3) =  3, (4,4) = 2, (4,5) = 1,
          (5,1) = 2, (5,2) = 1, (5,3) =  2, (5,4) = 1, (5,5) = 2):
plot(plot::Bars2d(L))
```




```
plot(plot::Bars2d(M))
```




```
plot(plot::Bars2d(A))
```



Example 7

Here is a real life example of a bar plot taken from a German magazine. It visualizes data related to waste management. We reproduce the plot via MuPAD. The main ingredient is a bar plot generated via `plot::Bars2d` with the option `GroupStyle` = `SingleBars`. Generating the annotations is somewhat tricky:

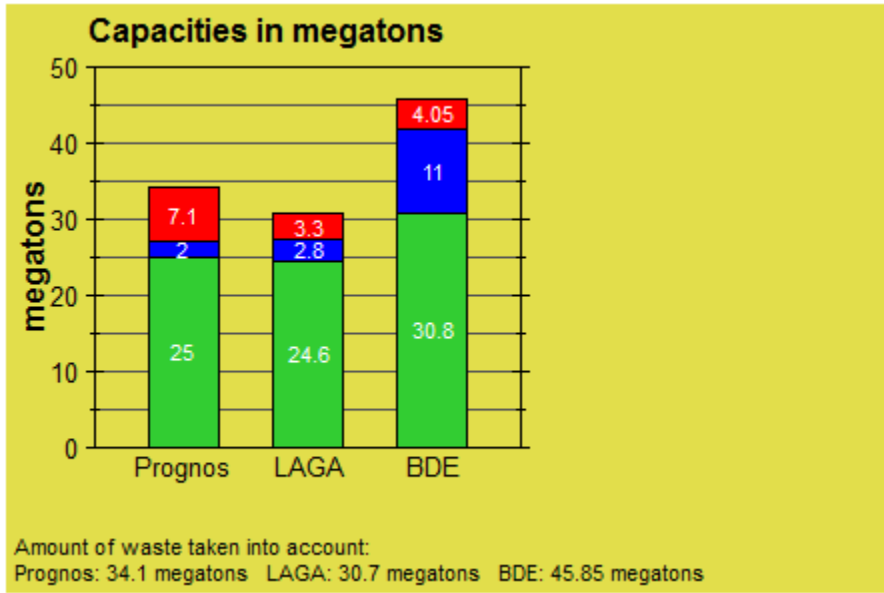
```
data := [[25  , 24.6, 30.8 ],
         [ 2  ,  2.8, 11   ],
         [ 7.1,  3.3,  4.05]]:

sw := 1.5:
bw := 2.0:
n := nops(data):
w := sw + bw:
myticks := [(i-1)* w + sw + bw/2 $ i = 1..n]:

m := nops(data[1]):

datalabels := ["Prognos", "LAGA", "BDE"]:
```

```
// cumulative data for the groups
datasums := _concat(datalabels[i].": ".
                    expr2text(_plus(data[j][i]$j=1..m)).
                    " megatons   " $i=1..n):

// generate a list of text objects containing the data values
// and place them in the centers of the bars:
datatext := []:
for i from 1 to n do
  h := 0;
  for j from 1 to m do
    d := data[j][i];
    datatext := datatext, plot::Text2d(expr2text(d),
                                       [myticks[i], h + d/2],
                                       TextFont = [8, RGB::White],
                                       VerticalAlignment = Center,
                                       HorizontalAlignment = Center);
    h := h + d
  end
end:
```

Here is the bar plot with the annotations. Many scene options are used to fine tune the graphics:

```
S1:=plot::Scene2d(
    plot::Bars2d(data,
                 Colors=[RGB::LimeGreen, RGB::Blue, RGB::Red],
                 GroupStyle = SingleBars,
                 BarCenters = [myticks[i] $ i=1..n],
                 BarWidths = [[bw]],
                 DrawMode = Vertical),

     // scene options:

     ViewingBox = [0 .. w*n + sw, 0 .. 50],

     // options for the grid
     XGridVisible = FALSE,
     YGridVisible = TRUE,
     XSubgridVisible = FALSE,
     YSubgridVisible = TRUE,
     GridLineColor = RGB::DarkGrey,
     SubgridLineColor = RGB::DarkGrey,

     // options for the axes
```

```
        Axes = Boxed,
        AxesTips = FALSE,
        AxesInFront = TRUE,
        AxesTitleFont = ["Arial", 12, Bold],
        XAxisVisible = TRUE,
        YAxisTitleOrientation = Vertical,
        YAxisTitleAlignment = Center,
        YAxisTitle = "megatons",
        XAxisTitle = "",

        // options for the ticks along the axes
        TicksLabelFont = ["Arial", 10],
        XTicksVisible = FALSE,
        XTicksNumber = None,
        XTicksAt = [myticks[i] = datalabels[i] $ i=1..n],

        // layout
        RightMargin = 50,

        // annotation
        datatext,

        // header and footer
        Header = "Capacities in megatons",
        HeaderFont = ["Arial", 12, Bold],
        Footer = "\n\nAmount of waste taken into account:\n".datasums,
        FooterFont = ["Arial", 8],
        FooterAlignment = Left,

        // use a yellowish background
        BackgroundColor = [0.886275, 0.870588, 0.294118]
    ):

    plot(S1)
```



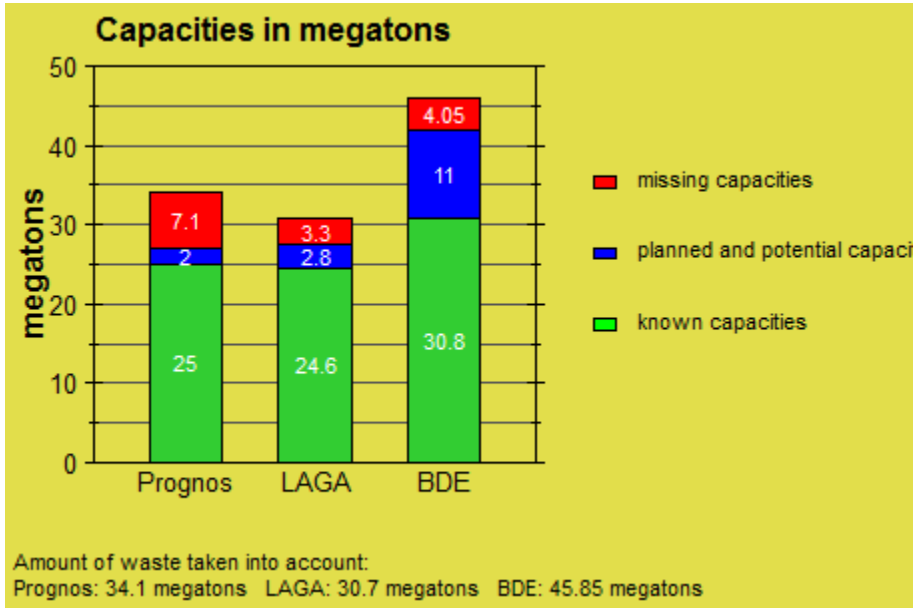
Next, we build a legend made of colored rectangles and text objects:

```
S2 := plot::Scene2d(
  ViewingBox = [0..20, 0..50],
  Axes = None,
  plot::Rectangle(13..13.5, 35..36,
                  Filled = TRUE,
                  FillPattern = Solid,
                  FillColor = RGB::Red,
                  LineColor = RGB::Black),
  plot::Text2d("missing capacities", [14, 35],
               HorizontalAlignment = Left,
               TextFont = ["Arial", 8]),
  plot::Rectangle(13..13.5, 29..30,
                  Filled = TRUE,
                  FillPattern = Solid,
                  FillColor = RGB::Blue,
                  LineColor = RGB::Black),
  plot::Text2d("planned and potential capacities", [14, 29],
               HorizontalAlignment = Left, TextFont = ["Arial", 8]),
  plot::Rectangle(13..13.5, 23..24,
                  Filled = TRUE,
```

```
                  FillPattern = Solid,
                  FillColor = RGB::Green,
                  LineColor = RGB::Black),
  plot::Text2d("known capacities", [14, 23],
               HorizontalAlignment = Left,
               TextFont = ["Arial", 8])
):

plot(S2, BorderWidth = 0.2)
```

The final picture consists of the bar plot S1 and the legend S2. We just put S2 on top of S1, making the background of S2 transparent:

```
S1::Width := 1: S1::Height := 1:
S2::Width := 1: S2::Height := 1:
S1::Bottom := 0: S1::Left := 0:
S2::Bottom := 0: S2::Left := 0:
S1::BackgroundTransparent := FALSE:
S2::BackgroundTransparent := TRUE:

plot(S1, S2, Layout = Relative)
```




```
delete data, datalabels, datasums, datatext, myticks,
       sw, bw, n, m, w, i, h, j, d, S1, S2:
```

Parameters

a_1, a_2, …, b_1, b_2, …, …

Real-valued expressions, possibly in the animation parameter.

a_1, a_2, …, b_1, b_2, …, … is equivalent to the attribute `Data`.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions

`plot | plot::copy`

MuPAD Graphical Primitives
`plot::Bars3d` | `plot::Histogram2d` | `plot::Scatterplot`

plot::Bars3d

3D bar chart of matrix data

Syntax

```
plot::Bars3d(A, <x = x_min .. x_max, y = y_min .. y_max>, <a = a_min .. a_max>, options)
plot::Bars3d(L, <x = x_min .. x_max, y = y_min .. y_max>, <a = a_min .. a_max>, options)
```

Description

`plot::Bars3d(A)` generates a 3D bar chart with bar heights given by the entries of the matrix `A`.

The rows of the matrix are plotted along the *x* coordinate, the columns along the *y* coordinate.

Different rows may be regarded as different classes of data. Each row has a different color determined by the the attribute `Colors = [ c1, c2, ...]` with RGB or RGBa colors c_1, c_2 etc.

The simplest way to obtain a uniform coloring of all rows with the color `c` is to specify the attribute `Color = c`.

Arrays/matrices do not need to be indexed from 1. E.g.,

```
A = array( `i_{min}` .. `i_{max}` , `j_{min}` .. `j_{max}` ,
[..data..])
```

yields a bar chart with $i_{max} - i_{min} + 1$ rows and $j_{max} - j_{min} + 1$ columns, stretching from x_{min} to x_{max} in *x* direction and from y_{min} to y_{max} in *y* direction.

If no plot range `` `x_{min}` .. `x_{max}` ``, `` `y_{min}` .. `y_{max}` `` is specified, $x_{min} = j_{min} - 1$, $x_{max} = j_{max}$, $y_{min} = i_{min} - 1$, $y_{max} = i_{max}$ is used.

When the values are specified by a list of lists *L* and no plot range `` `x_{min}` .. `x_{max}` ``, `` `y_{min}` .. `y_{max}` `` is specified, $x_{min} = 0$, $x_{max} = m$, $y_{min} = 0$, $y_{max} = n$ is used, where *n* is the length of *L* and *m* is the (common) length of the sublists in *L*. All sublists (“rows”) must have the same length.

The attribute `BarStyle` allows to switch the style of the bars between `Boxes` (bars), `Lines` (vertical lines), `LinesPoints` (vertical lines and points), and `Points`(points only). See “Example 1” on page 24-181.

The attribute `Gap` = `[`g_x`, `g_y`]` or, equivalently, `XGap` = g_x, `YGap` = g_y allows to introduce gaps between adjacent bars. The values g_x, g_y may be real numerical values between 0 and 1 or expressions of the animation parameter `a`. These values set the fraction of the space reserved for a bar that is not filled by the bar.

With g_x `= 0`, g_y `= 0`, there are no gaps. With g_x `= 0.5`, g_y `= 0.5`, the gaps between adjacent bars are of the same size as the bars. With g_x `= 1`, g_y `= 1`, there bars become lines.

Values of g_x, g_y larger than 1 are treated like 1, negative values like 0.

The `Gap` attribute has an effect only for `BarStyle` = `Boxes`.

The attribute `Ground` = z_0 determines the *z* value of the lower or upper face of the bars. Matrix values `m >` z_0 are displayed as bars stretching in *z* direction from the lower face z_0 to the upper face `m`. Matrix values `m <` z_0 are displayed as bars stretching in *z* direction from the upper face z_0 down to the lower face `m`.

The parameter z_0 has to be a numerical real value or an expression of the animation parameter `a`.

If the attribute `Ground` = z_0 is not specified, the default value z_0 `= 0` is used.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`BarStyle`	display style of bar plots	`Boxes`
`Color`	the main color	
`Colors`	list of colors to use	`[RGB::Blue, RGB::Red, RGB::Green, RGB::MuPADGold, RGB::Orange, RGB::Cyan,`

Attribute	Purpose	Default Value
		RGB::Magenta, RGB::LimeGreen, RGB::CadmiumYellowLight, RGB::AlizarinCrimson, RGB::Aqua, RGB::Lavender, RGB::SeaGreen, RGB::AureolineYellow, RGB::Banana, RGB::Beige, RGB::YellowGreen, RGB::Wheat, RGB::IndianRed, RGB::Black]
Data	the (statistical) data to plot	
Filled	filled or transparent areas and surfaces	TRUE
Frames	the number of frames in an animation	50
Gap	gaps between the bars of a bar chart	[0, 0]
Ground	base value	0
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Black.[0.25]
LineWidth	width of lines	0.35
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
Name	the name of a plot object (for browser and legend)	

Attribute	Purpose	Default Value
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	

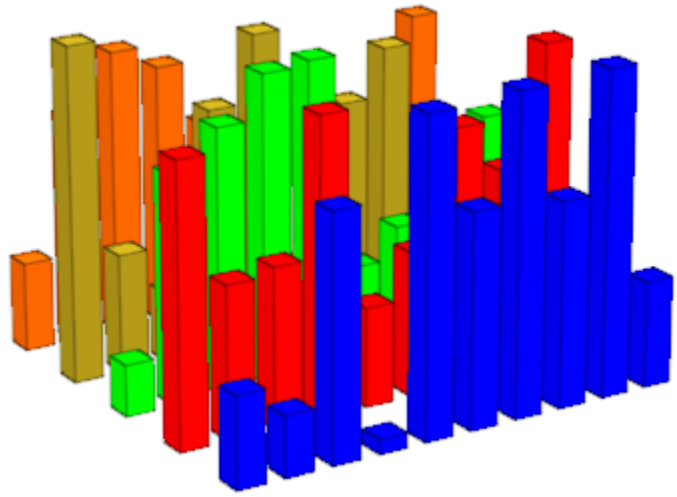
Attribute	Purpose	Default Value
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`
`XGap`	gaps in x direction between the bars of a bar chart	`0`
`XMax`	final value of parameter “x”	
`XMin`	initial value of parameter “x”	
`XName`	name of parameter “x”	
`XRange`	range of parameter “x”	
`YGap`	gaps in y direction between the bars of a bar chart	`0`
`YMax`	final value of parameter “y”	
`YMin`	initial value of parameter “y”	
`YName`	name of parameter “y”	
`YRange`	range of parameter “y”	

Examples

Example 1

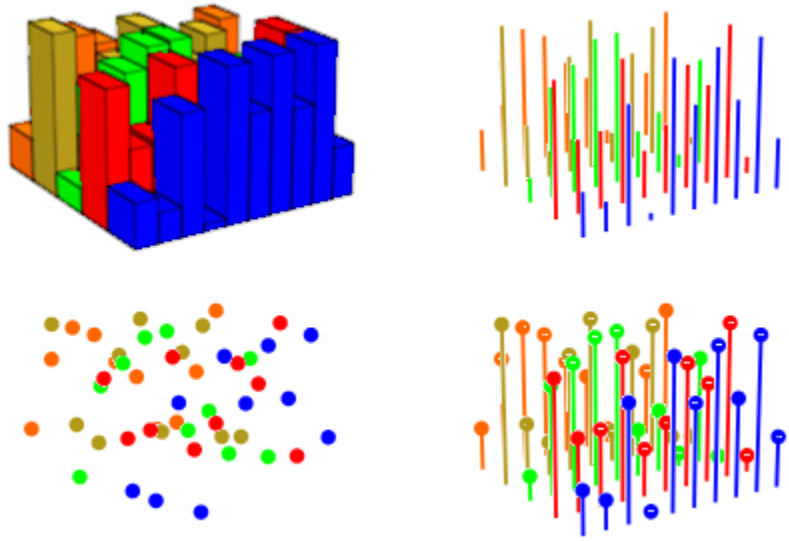
We create some random matrix data and plot them as a bar chart:

```
A := matrix::random(5, 10, frandom) :
plot(plot::Bars3d(A, Gap = [0.4, 0.7]))
```



We create bar charts of the data with different `BarStyle` options:

```
plot(plot::Scene3d(plot::Bars3d(A, BarStyle = Boxes)),
     plot::Scene3d(plot::Bars3d(A, BarStyle = Lines)),
     plot::Scene3d(plot::Bars3d(A, BarStyle = Points)),
     plot::Scene3d(plot::Bars3d(A, BarStyle = LinesPoints)),
     PointSize = 2.0*unit::mm, LineWidth = 0.5*unit::mm
):
```

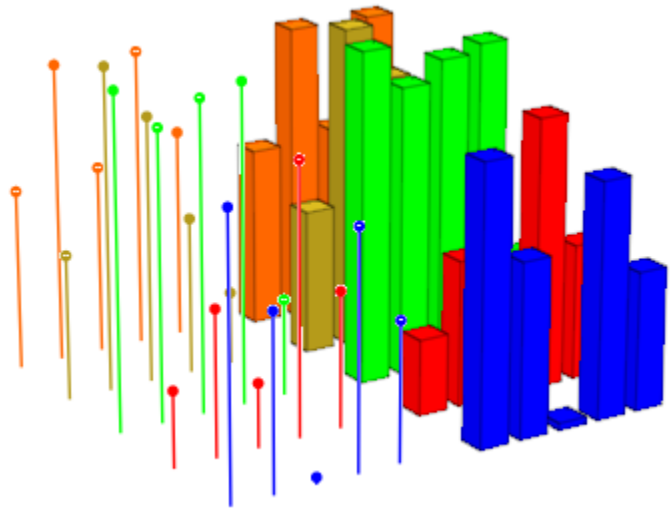


```
delete A:
```

Example 2

We demonstrate the positioning of bar charts by specifying ranges for the *x* and the *y* coordinate. The following two bar charts are plotted in one scene. They are placed side by side via suitable *x* ranges:

```
A := matrix::random(5, 5, frandom):
plot(plot::Bars3d(A, x = 0 .. 0.9, y = 0 .. 1,
                  BarStyle = LinesPoints),
     plot::Bars3d(A, x = 1.1 .. 2, y = 0 .. 1,
                  Gap = [0.3, 0.7])):
```

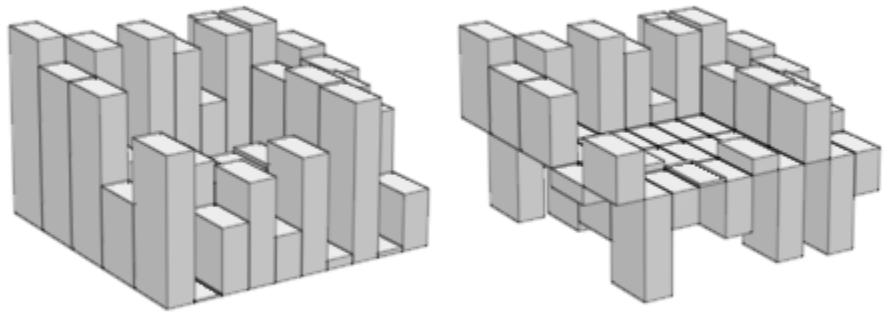


```
delete A:
```

Example 3

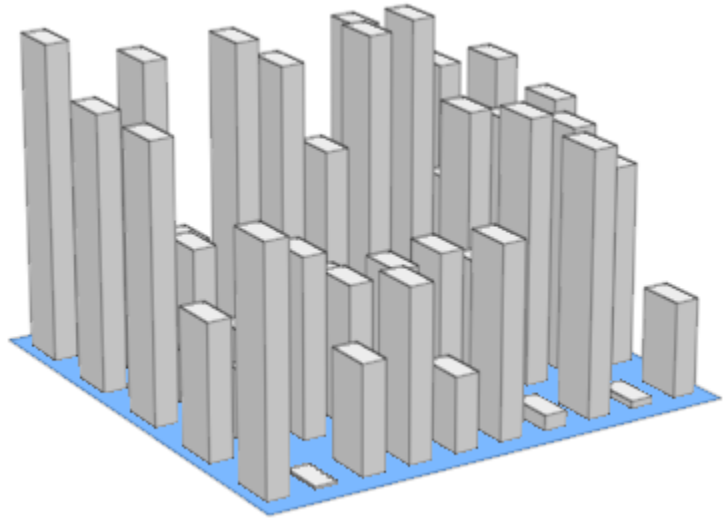
We demonstrate the attributes `Ground` and `Color`:

```
A := matrix::random(5, 10, frandom):
plot(plot::Scene3d(plot::Bars3d(A, Ground = 0,
                                Color = RGB::Grey)),
     plot::Scene3d(plot::Bars3d(A, Ground = 0.5,
                                Color = RGB::Grey)),
     Layout = Horizontal):
```



In the next call, the ground level is animated. Note that in animations one must specify ranges for the *x* and *y* coordinates.We include a transparent plane visualizing the ground level:

```
plot(plot::Bars3d(A, x = 0 .. 1, y = 0 .. 1, a = 0 .. PI,
                  Color = RGB::Grey, Gap = [0.5, 0.5],
                  Ground = sin(a)),
     plot::Surface([x, y, sin(a) + 0.001],
                   x = 0 .. 1, y = 0 .. 1, a = 0 .. PI,
                   Mesh = [2, 2], Color = RGB::Blue.[0.5])):
```



```
delete A:
```

Parameters

A

An array of domain type `DOM_ARRAY` or a matrix of category `Cat::Matrix` (e.g., of type `matrix` or `densematrix`) containing real numerical values or expressions of the animation parameter *a*. Rows/columns of the array, respectively matrix, correspond to rows/columns of the bar chart.

A is equivalent to the attribute `Data`.

L

A list of lists of real numerical values or expressions of the animation parameter *a*. Each sublist of *L* represents a row of the bar chart.

L is equivalent to the attribute `Data`.

x

Name of the horizontal variable: an identifier or an indexed identifier. It is used as the title of the coordinate axis in *x* direction.

x is equivalent to the attribute `XName`.

x_{min} .. x_{max}

The range of the horizontal variable: x_{min}, x_{max} must be numerical real value or expressions of the animation parameter *a*.

x_{min} .. x_{max} is equivalent to the attribute `XRange`.

y

Name of the vertical variable: an identifier or an indexed identifier. It is used as the title of the coordinate axis in *y* direction.

y is equivalent to the attribute `YName`.

y_{min} .. y_{max}

The range of the vertical variable: y_{min}, y_{max} must be numerical real value or expressions of the animation parameter *a*.

y_{min} .. y_{max} is equivalent to the attribute `YRange`.

a

Animation parameter, specified as $a = a_{min}..a_{max}$, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Bars2d` | `plot::Histogram2d` | `plot::Matrixplot`

plot::Box

Boxes in 3D

Syntax

```
plot::Box(x_min .. x_max, y_min .. y_max, z_min .. z_max, <a = a_min .. a_max>, options)
plot::Box([x_min, y_min, z_min], [x_max, y_max, z_max], <a = a_min .. a_max>, options)
```

Description

`plot::Box(`x_{min}`..`x_{max}`, `y_{min}`..`y_{max}`, `z_{min}`..`z_{max}`)` creates the 3D box

$$\{(x\ y\ z) \mid x_{min} \le x \le x_{max}, y_{min} \le y \le y_{max}, z_{min} \le z \le z_{max}\}$$

`plot::Box([x_min, y_min, z_min], [x_max, y_max, z_max])` produces the same box.

`plot::Box` creates 3D boxes with edges parallel to the coordinate axes. Using `plot::Rotate3d` or `plot::Transform3d` one can create boxes and parallelepipeds with arbitrary orientation. Cf. examples “Example 3” on page 24-192 and “Example 4” on page 24-195.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Color`	the main color	`RGB::LightBlue`
`Filled`	filled or transparent areas and surfaces	`TRUE`
`FillColor`	color of areas and surfaces	`RGB::LightBlue`
`Frames`	the number of frames in an animation	`50`

Attribute	Purpose	Default Value
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Black.[0.25]
LineWidth	width of lines	0.35
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	0
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	

Attribute	Purpose	Default Value
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XMax	final value of parameter “x”	1
XMin	initial value of parameter “x”	-1
XRange	range of parameter “x”	-1 .. 1
YMax	final value of parameter “y”	1
YMin	initial value of parameter “y”	-1
YRange	range of parameter “y”	-1 .. 1
ZMax	final value of parameter “z”	1

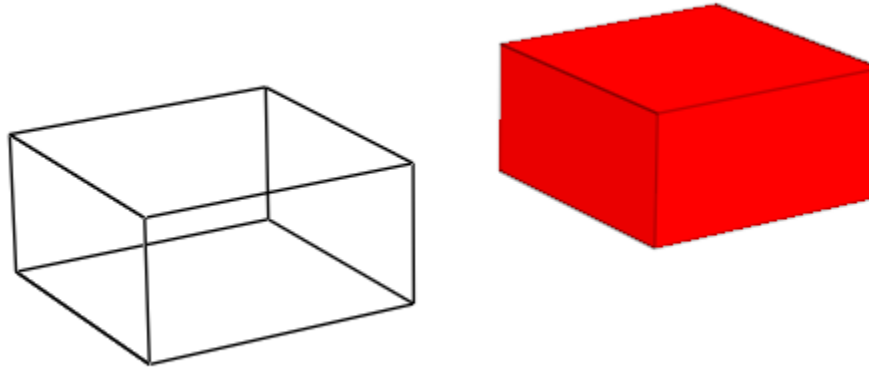
Attribute	Purpose	Default Value
ZMin	initial value of parameter "z"	-1
ZRange	range of parameter "z"	-1 .. 1

Examples

Example 1

We draw a box consisting of its edges and a filled box:

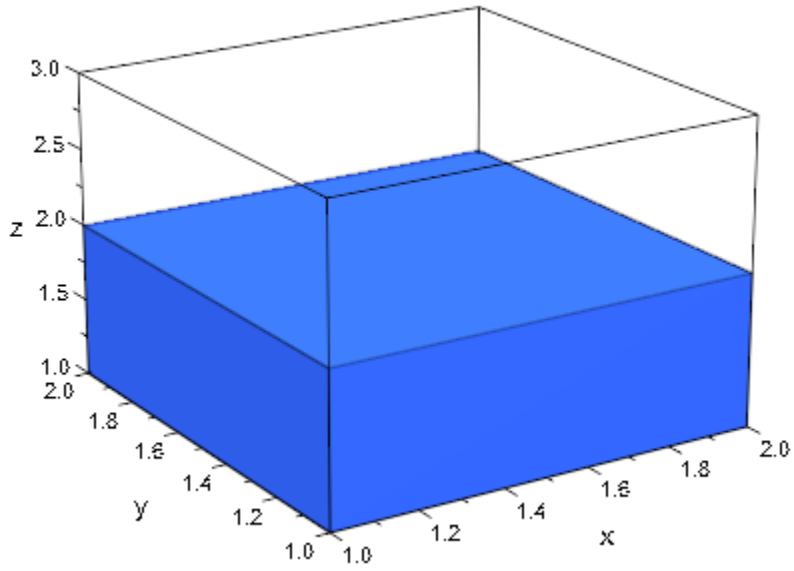
```
plot(plot::Box(-3..-1, 0..2, 0..1, Filled = FALSE,
               LineColor = RGB::Black),
     plot::Box(1..3, 0..2, 0..1, Filled = TRUE,
               FillColor = RGB::Red),
     Axes = None, Scaling = Constrained)
```



Example 2

The borders of a box can be animated:

```
plot(plot::Box([1, 1, 1], [2, 2, 2 + sin(r)], r = 0..2*PI)):
```



Example 3

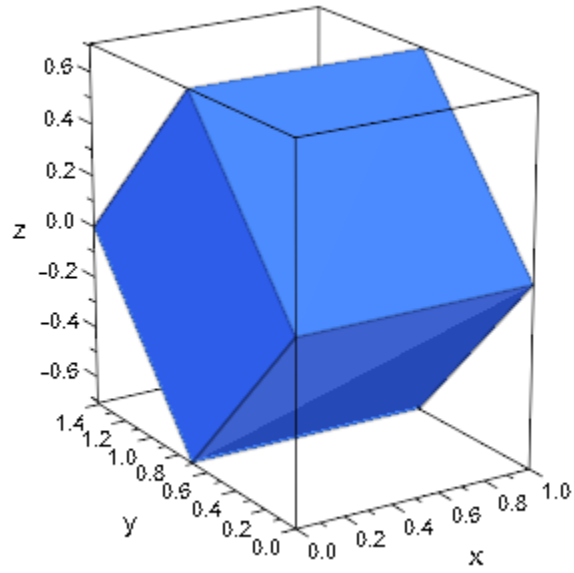
We want to display a cube "standing" on one of its corners. First, we define the cube:

```
b0 := plot::Box(0..1, 0..1, 0..1)
```

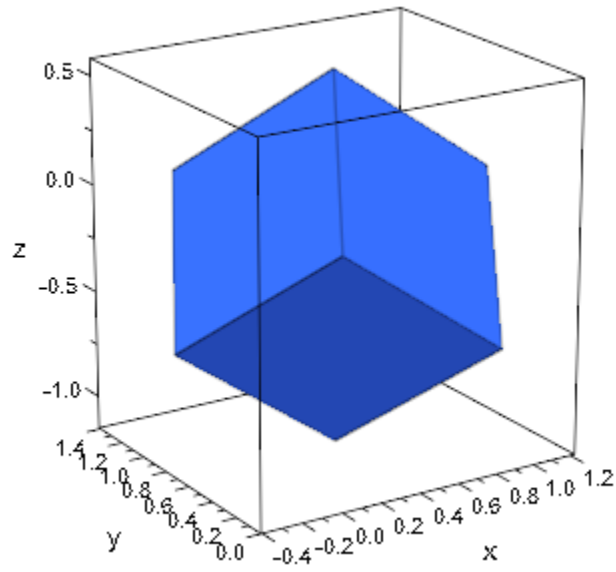
$$\text{plot::Box}(0..1, 0..1, 0..1)$$

Now, rotating the cube to stand on a corner is equivalent to first rotating around the x-axis by 45 degrees, then rotating around the y-axis:

```
b1 := plot::Rotate3d(b0, Axis = [1, 0, 0], Angle = -PI/4):
plot(b1, Scaling = Constrained)
```

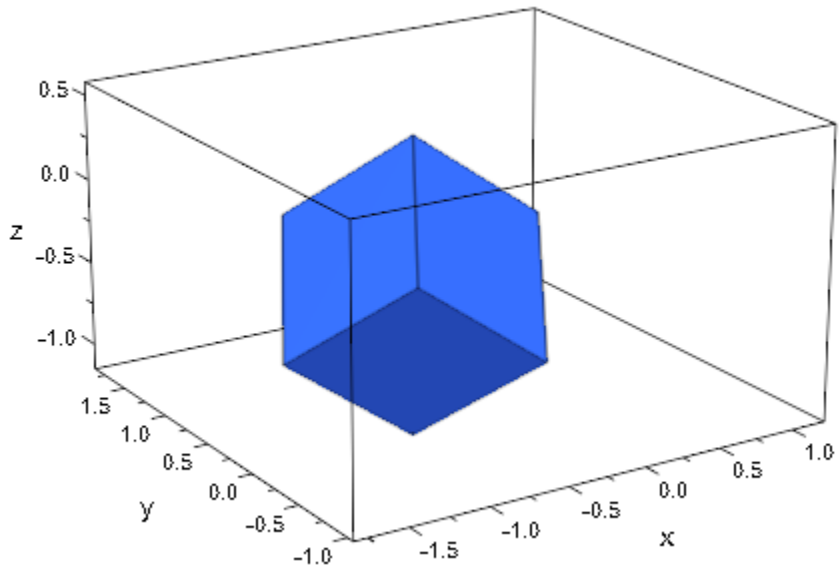



```
b2 := plot::Rotate3d(b1, Axis = [0, 1, 0], Angle = 7*PI/36):
plot(b2, Scaling = Constrained)
```



Finally, we let it rotate around the *z*-axis:

```
plot(plot::Rotate3d(b2, Axis = [0, 0, 1], Angle = a,
                    a = 0..2*PI/3),
     Scaling = Constrained)
```

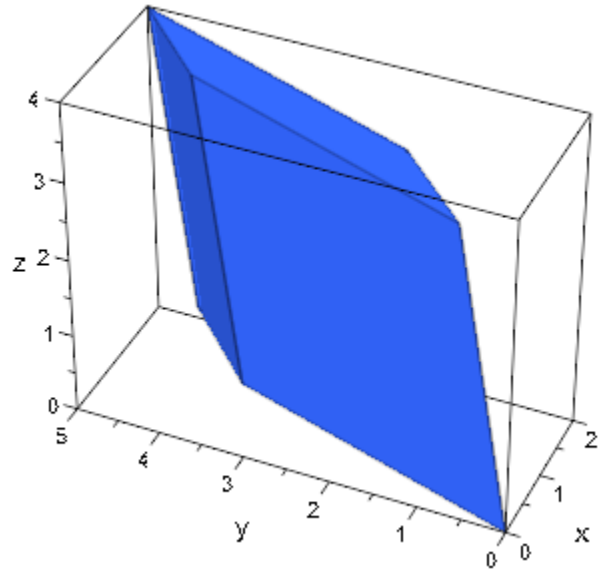



```
delete b0, b1, b2:
```

Example 4

A parallelepiped can be obtained from a `plot::Box` by a linear transformation:

```
plot(plot::Transform3d([0, 0, 0], [1, 1, 0,
                                   1, 1, 3,
                                   0, 3, 1],
                       plot::Box(0..1, 0..1, 0..1)),
     Scaling = Constrained,
     CameraDirection = [-27, -12, 22])
```



Parameters

x_{min}, y_{min}, z_{min}

The lower borders: numerical real values or arithmetical expressions of the animation parameter `a`.

x_{min}, y_{min}, z_{min} are equivalent to the attributes `XMin`, `YMin`, `ZMin`.

x_{max}, y_{max}, z_{max}

The upper borders: numerical real values or arithmetical expressions of the animation parameter `a`.

x_{max}, y_{max}, z_{max} are equivalent to the attributes `XMax`, `YMax`, `ZMax`.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Polygon3d` | `plot::Rotate3d` | `plot::Scale3d` | `plot::Surface` | `plot::Transform3d`

plot::Boxplot

Statistical box plots

Syntax

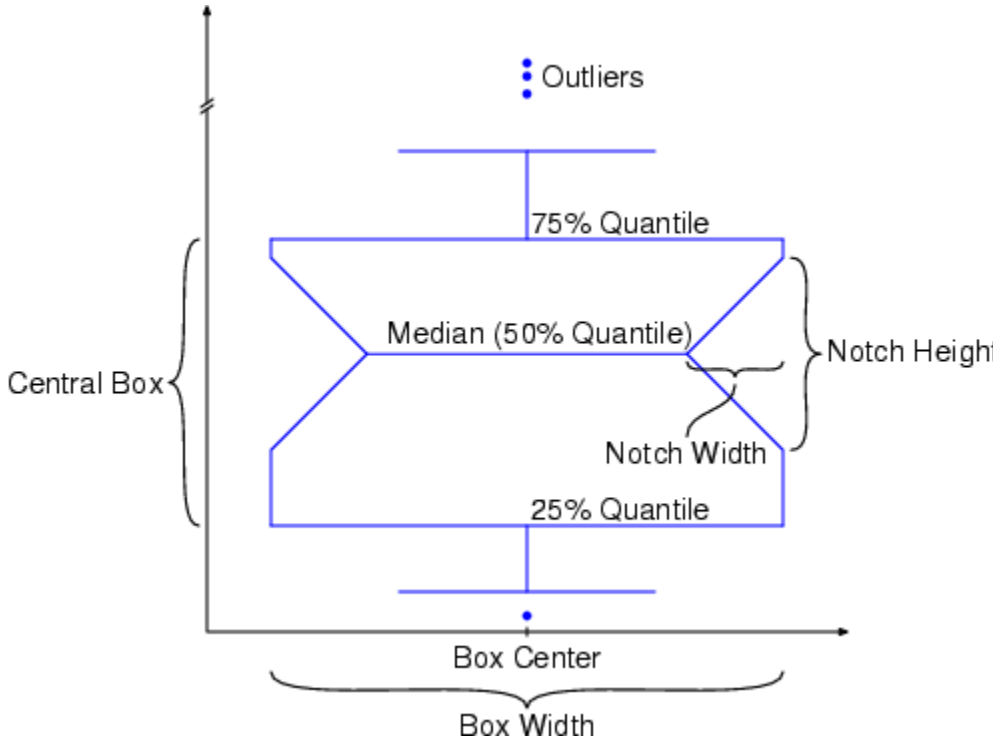
```
plot::Boxplot(L1, …, <a = amin .. amax>, options)

plot::Boxplot([L1, …], <a = amin .. amax>, options)

plot::Boxplot(A, <a = amin .. amax>, options)

plot::Boxplot(s, <c1, …>, <a = amin .. amax>, options)

plot::Boxplot(s, <[c1, …]>, <a = amin .. amax>, options)
```

Description

`plot::Boxplot(data)` creates a box plot of the given data.

`plot::Boxplot` creates a box plot of discrete data samples. Box plots reduce data samples to a number of descriptive parameters and are a useful means of comparing statistical data.

In particular, each data sample is represented as one box. A typical box consists of the following subparts:



- A “central box” representing the central 50% of the data. Its lower and upper boundary lines are at the 25%/75% quantile of the data. A central line indicates the median of the data.
- Two vertical lines extending from the central box indicating the remaining data outside the central box that are not regarded as outliers. These lines extend maximally to $\frac{3}{2}$ times the height of the central box but not past the range of the data.
- Outliers: these are points indicating the remaining data.

With the special attribute `Notched` = `TRUE`, the sides of the boxes can be notched, thus providing additional information on the data sample. The horizontal width of the notches may be set by the attribute `NotchWidth`.

The special attributes `BoxCenters` and `BoxWidths` allow to center the boxes at arbitrary positions along the horizontal axis and to set the horizontal width of the boxes, respectively.

The special attribute `Averaged` determines whether the quantile values are computed with or without the option `Averaged` (cf. `stats::empiricalQuantile`).

Specifying `DrawMode` = `Horizontal`, the boxes are rotated by 90 degrees.

The attribute `Colors` allows to specify the color of each box in a box plot. A common color for all boxes may be specified via `Color`.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`FALSE`
`Averaged`	mode for computing quantile lines in box plots	`TRUE`
`BoxWidths`	widths of boxes in a box plot	`[0.8]`
`BoxCenters`	position of boxes in a box plot	`[1]`
`Color`	the main color	
`Colors`	list of colors to use	`[RGB::Blue, RGB::Red, RGB::Green, RGB::MuPADGold, RGB::Orange, RGB::Cyan, RGB::Magenta, RGB::LimeGreen, RGB::CadmiumYellowLight, RGB::AlizarinCrimson]`
`Data`	the (statistical) data to plot	
`DrawMode`	orientation of boxes and bars	`Vertical`
`Filled`	filled or transparent areas and surfaces	`TRUE`
`FillPattern`	type of area filling	`DiagonalLines`

Attribute	Purpose	Default Value
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineWidth	width of lines	0.35
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
Name	the name of a plot object (for browser and legend)	
Notched	notched boxes in box plots	FALSE
NotchWidth	width of notches in box plots	0.2
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	

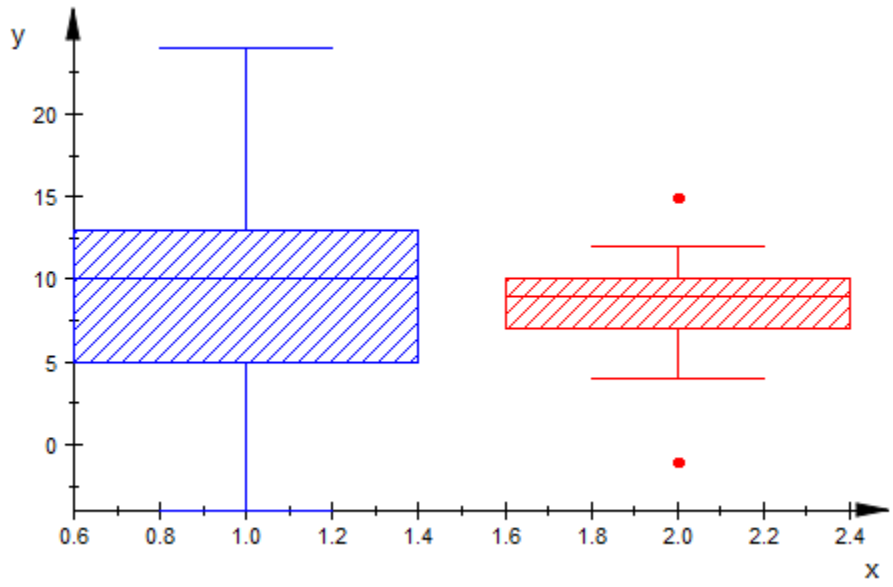
Attribute	Purpose	Default Value
`TitleAlignment`	horizontal alignment of titles w.r.t. their coordinates	`Center`
`TitlePositionX`	position of object titles, x component	
`TitlePositionY`	position of object titles, y component	
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

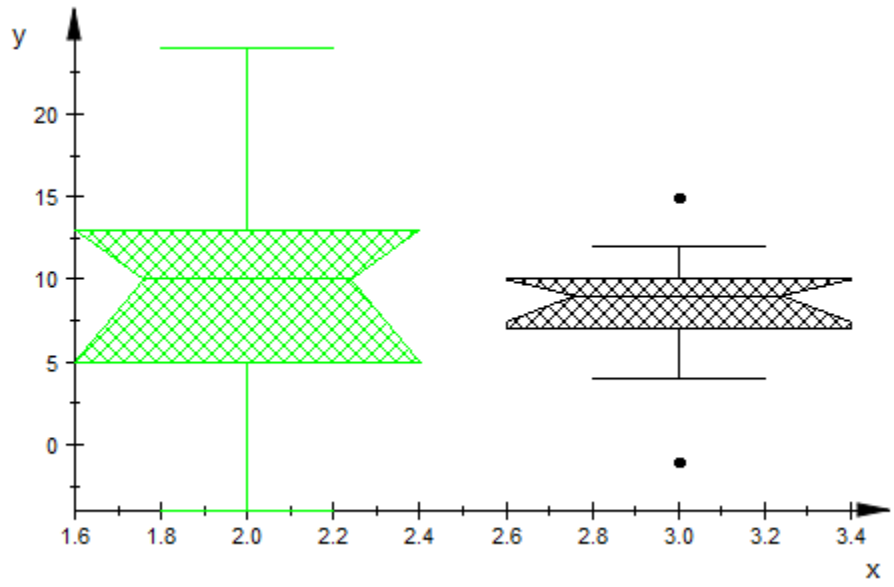
Just to show basic usage of `plot::Boxplot`, we plot some data samples chosen arbitrarily:

```
data1 := [5, 10, 24, -4, 13]:
data2 := [7, 9, -1, 4, 10, 8, 12, 10, 15]:
b := plot::Boxplot(data1, data2):
plot(b)
```



We can modify the appearance of the box plot in various ways:

```
b::Notched := TRUE:
b::Colors  := [RGB::Green, RGB::Black]:
b::BoxCenters := [2, 3]:
b::FillPattern := XCrossedLines:

plot(b)
```

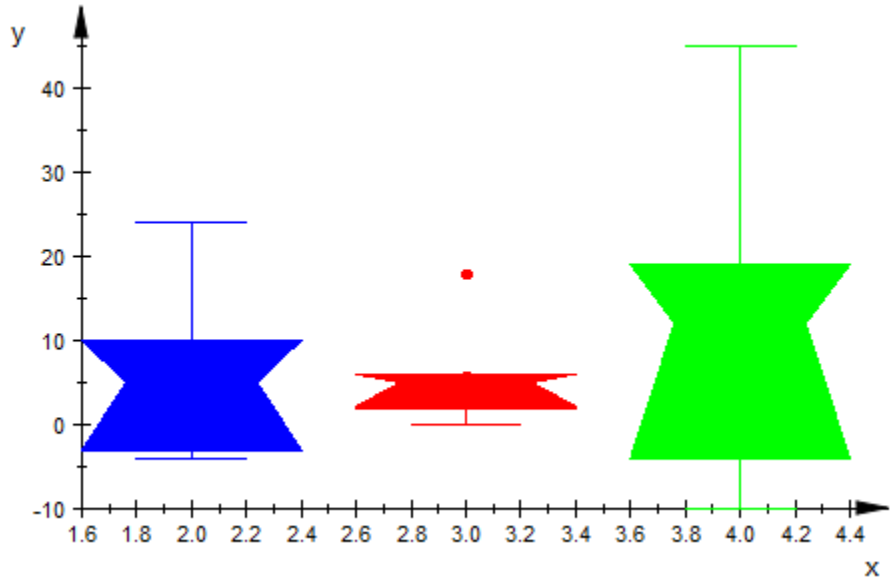



```
delete b:
```

Example 2

It is possible to shift the whole plot in x-direction by providing a center for the first box via `BoxCenters`:

```
A := matrix([[ 5,  6,  19],
             [10,  5,  45],
             [24,  2,  12],
             [-3, 18, -10],
             [-4,  0, - 4]]):

plot(plot::Boxplot(A, BoxCenters = [2], Notched = TRUE,
                   FillPattern = Solid))
```




```
delete A:
```

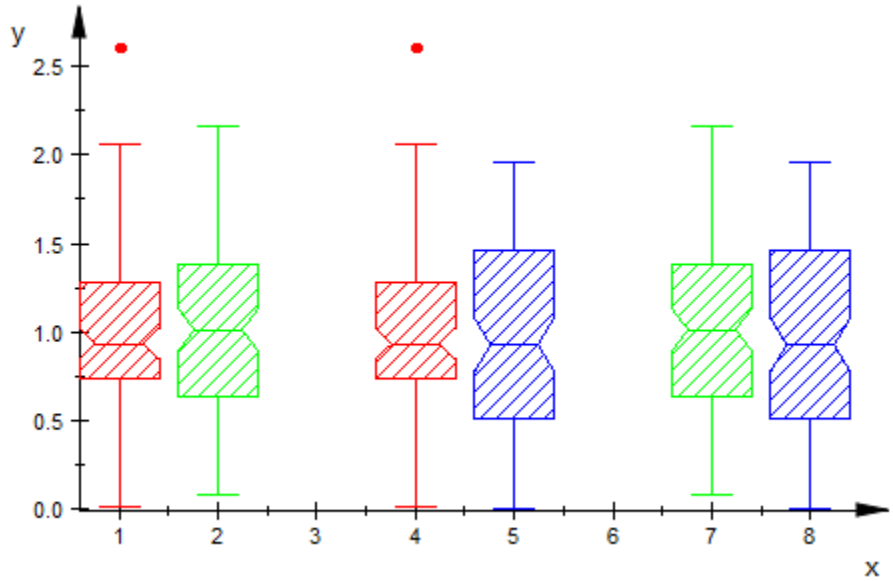
Example 3

The primary use of `plot::Boxplot` is comparing data sets. We shall do this for data produced by the following random number generators:

```
f := stats::normalRandom(1, 0.2):
g := stats::uniformRandom(0, 2):
```

Now, we create small samples and compare their boxes:

```
data1 := [f() $ k = 1..100]: // Red
data2 := [f() $ k = 1..100]: // Green
data3 := [g() $ k = 1..100]: // Blue
plot(plot::Boxplot(data1, data2, data1, data3, data2, data3,
     Colors = [RGB::Red, RGB::Green, RGB::Red, RGB::Blue,
               RGB::Green, RGB::Blue],
     BoxCenters = [1, 2, 4, 5, 7, 8], Notched = TRUE))
```



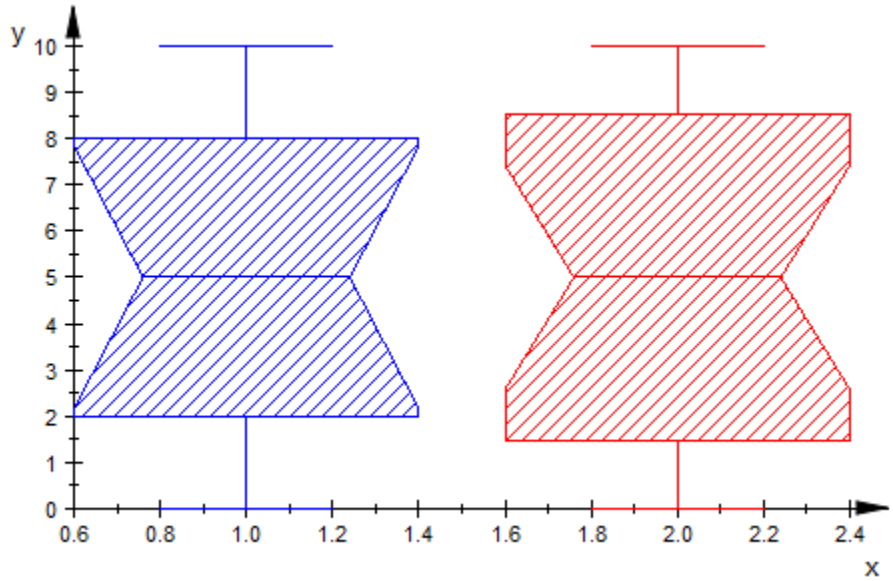
Comparing the central boxes, the blue data differ significantly from the red and the green data. The red and green boxes, however, are quite similar – as they should, given that the red and green data were produced by the same random generator `f`.

```
delete f, g, data1, data2, data3:
```

Example 4

For symmetric input data, the images generated by `plot::Boxplot` are symmetric, too:

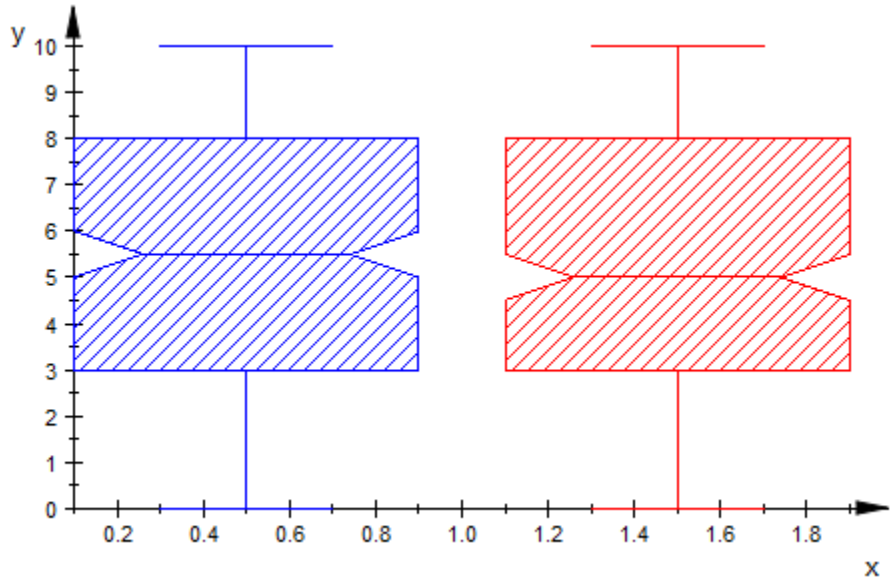
```
plot(plot::Boxplot([$0..10], [5+5*sin(PI*n/20) $ n=-10..10], Notched))
```



Example 5

By default, the quantile lines of the boxes are computed with the option `Averaged` (see `stats::empiricalQuantile` for details). When using `Averaged` = `FALSE`, the quantiles are computed without this option:

```
r := random(0..10):
SEED := 123:
data := [r() $ k = 1..250]:
plot(plot::Boxplot(data, Averaged = TRUE, BoxCenters = 0.5,
                         Color = RGB::Blue, Notched),
     plot::Boxplot(data, Averaged = FALSE, BoxCenters = 1.5,
                         Color = RGB::Red, Notched)
    ):
```




```
delete r, SEED, data:
```

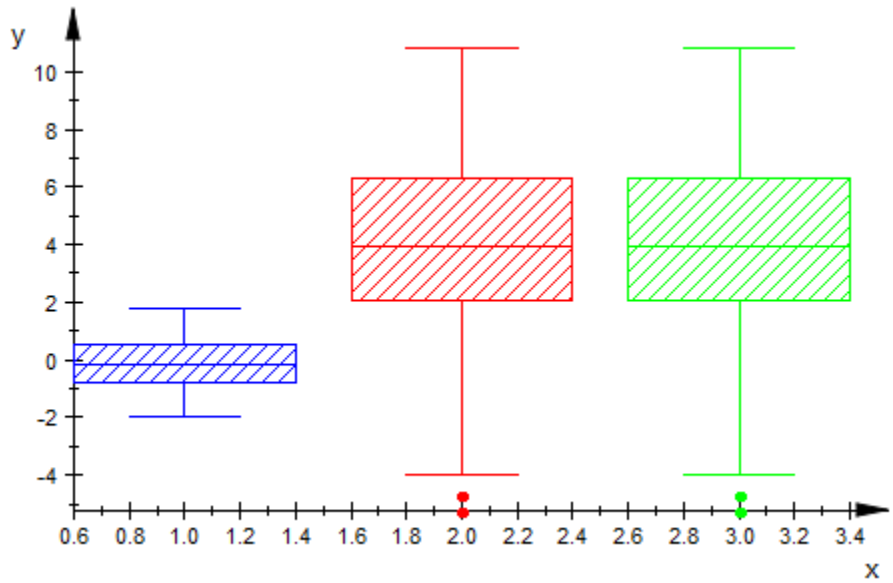
Example 6

Box plots can be animated. We create two data samples and fuse them to a symbolic superposition:

```
f1 := stats::normalRandom(0, 1):
f2 := stats::normalRandom(4, 8):
data0 := sort([f1() $ k = 1..100]):
data1 := sort([f2() $ k = 1..100]):
data01 := [(1 - a)*data0[i] + a*data1[i] $ i = 1..100]:
```

The box associated with the data sample `data01` changes from the box associated with `data0` to the box associated with `data1` as the animation parameter increases from $a = 0$ to $a = 1$:

```
plot(plot::Boxplot(data0, data01, data1, a = 0..1))
```




```
delete f1, f2, data0, data1, data01:
```

Parameters

L_1, …

Data samples: lists of numerical real values or arithmetical expressions of the animation parameter `a`.

L_1, … is equivalent to the attribute `Data`.

A

An array of domain type `DOM_ARRAY` or a matrix of category `Cat::Matrix` (e.g., of type `matrix` or `densematrix`) providing numerical real values or arithmetical expressions of the animation parameter `a`. The columns are regarded as separate data samples. Also a 1-dimensional array, regarded as a single data sample, is accepted.

A is equivalent to the attribute `Data`.

s

A data collection of domain type `stats::sample`. The columns in `s` are regarded as separate data samples.

s is equivalent to the attribute `Data`.

c_1, …

Column indices into `s`: positive integers. These indices, if given, indicate that only the specified columns in `s` should be used as data samples. If no column indices are specified, *all* columns in `s` are used as data samples.

a

Animation parameter, specified as a = a_{min}..a_{max}, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions

`plot` | `plot::copy`

MuPAD Graphical Primitives

`plot::Bars2d` | `plot::Bars3d` | `plot::Histogram2d` | `plot::Scatterplot`

plot::Circle2d

2D circles

Syntax

```
plot::Circle2d(r, <[x, y]>, <a = a_min .. a_max>, options)
```

Description

`plot::Circle2d(r, [x, y])` creates a 2D circle with radius *r* and center (*x*, *y*).

Per default circles are drawn as lines both in 2D and 3D. The attribute `LineColor` or, equivalently, `Color` serves for setting the line color.

Use the attribute `Filled` = `TRUE` to create filled circles in 2D.

In 2D, one can choose between hatched and solidly filled circles via the attribute `FillPattern`. The fill color is determined by `FillColor`. The circumferential line can be "switched off" via `LinesVisible = FALSE`.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`
`Center`	center of objects, rotation center	`[0, 0]`
`CenterX`	center of objects, rotation center, x-component	`0`
`CenterY`	center of objects, rotation center, y-component	`0`
`Color`	the main color	`RGB::Blue`

Attribute	Purpose	Default Value
`Filled`	filled or transparent areas and surfaces	`FALSE`
`FillColor`	color of areas and surfaces	`RGB::Red`
`FillPattern`	type of area filling	`DiagonalLines`
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`LineColor`	color of lines	`RGB::Blue`
`LineWidth`	width of lines	`0.35`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LinesVisible`	visibility of lines	`TRUE`
`LineColorDirection`	the direction of color transitions on lines	[0, 1]
`LineColorDirectionX`	x-component of the direction of color transitions on lines	`0`
`LineColorDirectionY`	y-component of the direction of color transitions on lines	`1`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	

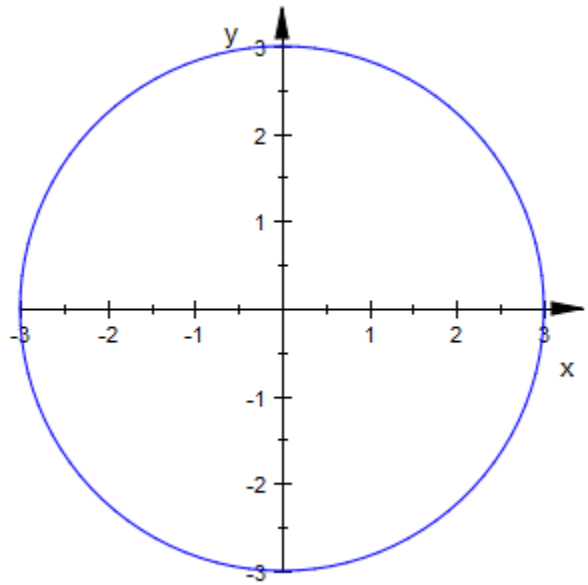
Attribute	Purpose	Default Value
ParameterRange	range of the animation parameter	
Radius	radius of circles, spheres etc.	1
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE

Examples

Example 1

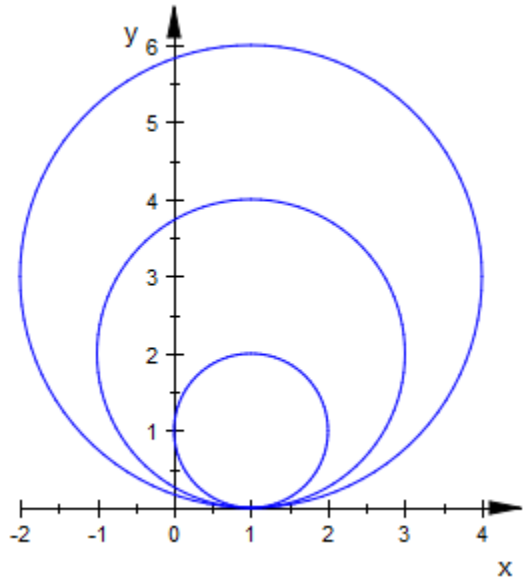
Circles centered at the origin are created if only a radius is specified:

```
plot(plot::Circle2d(3)):
```



A center may be given as a list of coordinates:

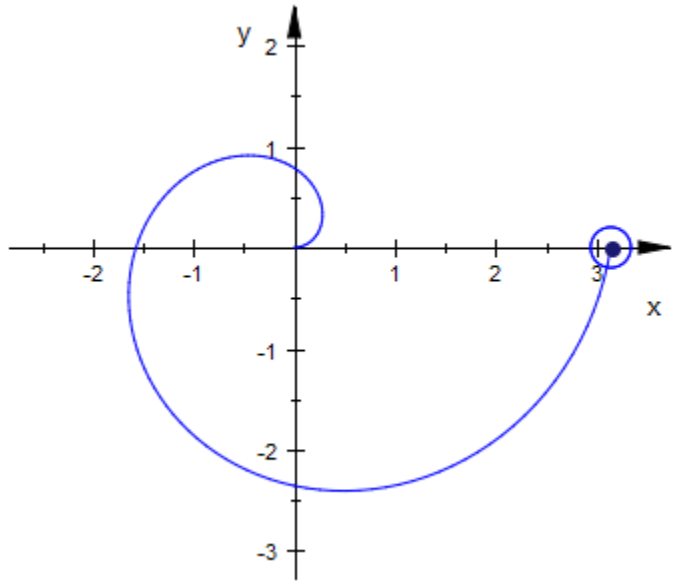
```
plot(plot::Circle2d(1, [1, 1]),
     plot::Circle2d(2, [1, 2]),
     plot::Circle2d(3, [1, 3])):
```



Example 2

Radius and center of a circle can be animated. We plot an animated circle with a changing radius and a center moving on a spiral:

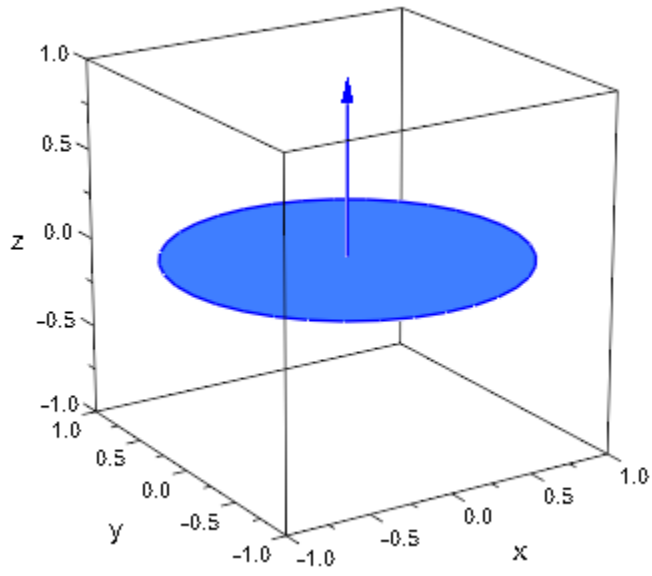
```
plot(plot::Curve2d([a*cos(2*a), a*sin(2*a)], a = 0..PI),
     plot::Point2d([a*cos(2*a), a*sin(2*a)], a = 0..PI,
                   PointSize = 2*unit::mm),
     plot::Circle2d(0.2 + sin(a), [a*cos(2*a), a*sin(2*a)],
                    a = 0..PI))
```



Example 3

In three dimensions, a circle requires a normal vector. We animate this vector:

```
normal_ := plot::Arrow3d(
      [0, 0, 0],
      [sin(2*a), sin(a)*cos(2*a), cos(a)*cos(2*a)],
      a = 0..2*PI):
circle := plot::Circle3d(1, [0, 0, 0], normal_::To,
                         a = 0..2*PI, Filled):
plot(normal_, circle)
```



Parameters

r

The radius: a real numerical value or an arithmetical expression in the animation parameter `a`.

r is equivalent to the attribute `Radius`.

x, y

The center. The coordinates `x`, `y` must be real numerical values or arithmetical expressions in the animation parameter `a`. If no center is specified, a circle centered at the origin is created.

x, y are equivalent to the attributes `CenterX`, `CenterY`.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Circle3d` | `plot::Cone` | `plot::Sphere`

plot::Circle3d

3D circles

Syntax

```
plot::Circle3d(r, <[x, y, z]>, <a = a_min .. a_max>, options)
```

```
plot::Circle3d(r, [x, y, z], [n_x, n_y, n_z], <a = a_min .. a_max>, options)
```

Description

`plot::Circle3d(r, [x, y, z], [ n_x, n_y, n_z])` creates a 3D circle with radius r, center (x, y, z), and normal vector (n_x, n_y, n_z).

Per default circles are drawn as lines both in 2D and 3D. The attribute `LineColor` or, equivalently, `Color` serves for setting the line color.

Use the attribute `Filled = TRUE` to create circular discs in 3D.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Center`	center of objects, rotation center	`[0, 0, 0]`
`CenterX`	center of objects, rotation center, x-component	`0`
`CenterY`	center of objects, rotation center, y-component	`0`
`CenterZ`	center of objects, rotation center, z-component	`0`
`Color`	the main color	`RGB::Blue`

Attribute	Purpose	Default Value
`Filled`	filled or transparent areas and surfaces	`FALSE`
`FillColor`	color of areas and surfaces	`RGB::LightBlue`
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`LineColor`	color of lines	`RGB::Blue`
`LineWidth`	width of lines	`0.35`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LinesVisible`	visibility of lines	`TRUE`
`LineColorDirection`	the direction of color transitions on lines	[0, 0, 1]
`LineColorDirectionX`	x-component of the direction of color transitions on lines	`0`
`LineColorDirectionY`	y-component of the direction of color transitions on lines	`0`
`LineColorDirectionZ`	z-component of the direction of color transitions on lines	`1`
`Name`	the name of a plot object (for browser and legend)	
`Normal`	normal vector of circles and discs, etc. in 3D	[0, 0, 1]
`NormalX`	normal vector of circles and discs, etc. in 3D, x-component	`0`

Attribute	Purpose	Default Value
NormalY	normal vector of circles and discs, etc. in 3D, y-component	0
NormalZ	normal vector of circles and discs, etc. in 3D, z-component	1
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
Radius	radius of circles, spheres etc.	1
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Visible	visibility	TRUE

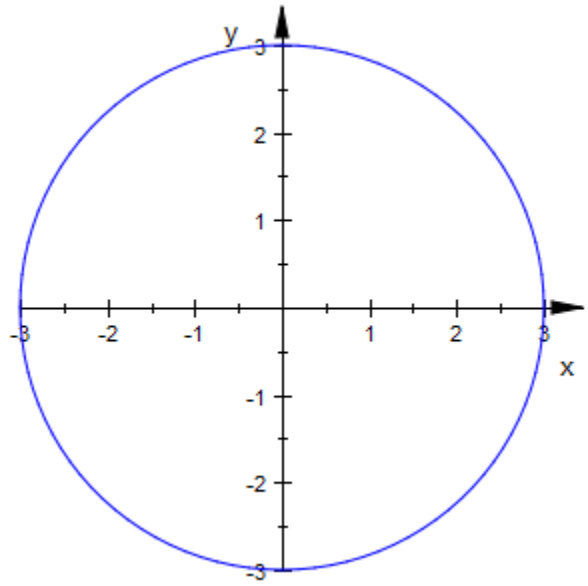
Attribute	Purpose	Default Value
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

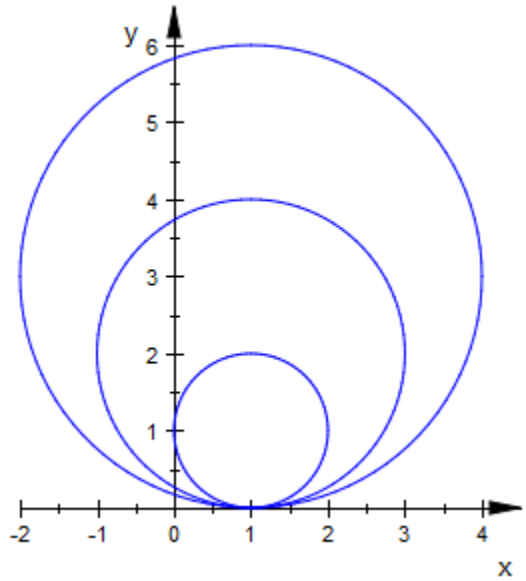
Circles centered at the origin are created if only a radius is specified:

```
plot(plot::Circle2d(3)):
```



A center may be given as a list of coordinates:

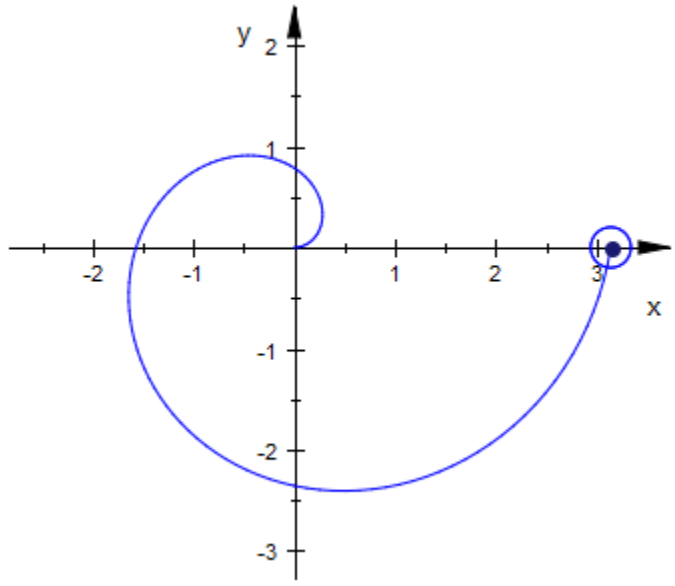
```
plot(plot::Circle2d(1, [1, 1]),
     plot::Circle2d(2, [1, 2]),
     plot::Circle2d(3, [1, 3])):
```



Example 2

Radius and center of a circle can be animated. We plot an animated circle with a changing radius and a center moving on a spiral:

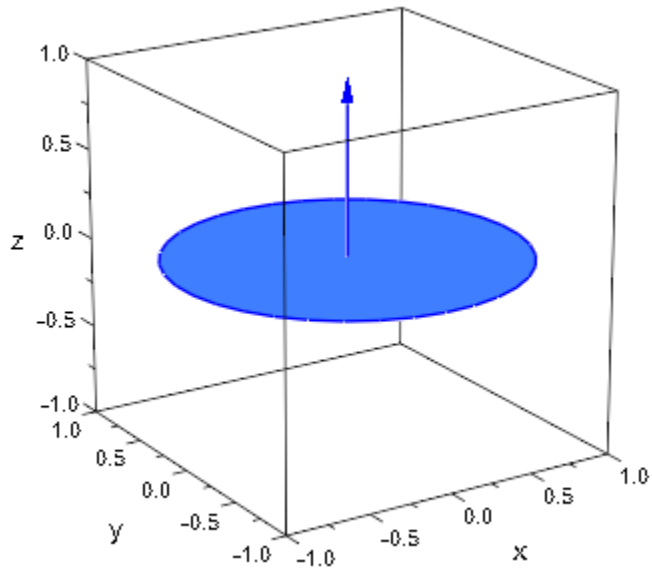
```
plot(plot::Curve2d([a*cos(2*a), a*sin(2*a)], a = 0..PI),
     plot::Point2d([a*cos(2*a), a*sin(2*a)], a = 0..PI,
                   PointSize = 2*unit::mm),
     plot::Circle2d(0.2 + sin(a), [a*cos(2*a), a*sin(2*a)],
                    a = 0..PI))
```



Example 3

In three dimensions, a circle requires a normal vector. We animate this vector:

```
normal_ := plot::Arrow3d(
      [0, 0, 0],
      [sin(2*a), sin(a)*cos(2*a), cos(a)*cos(2*a)],
      a = 0..2*PI):
circle := plot::Circle3d(1, [0, 0, 0], normal_::To,
                         a = 0..2*PI, Filled):
plot(normal_, circle)
```



Parameters

r

The radius: a real numerical value or an arithmetical expression in the animation parameter `a`.

r is equivalent to the attribute `Radius`.

x, y, z

The center. The coordinates `x`, `y`, `z` must be real numerical values or arithmetical expressions in the animation parameter `a`. If no center is specified, a circle centered at the origin is created.

x, y, z are equivalent to the attributes `CenterX`, `CenterY`, `CenterZ`.

n_x, n_y, n_z

The normal vector. The components n_x, n_y, n_z must be real numerical values or arithmetical expressions in the animation parameter `a`. If no normal is specified, the normal (0, 0, 1) is used.

n_x, n_y, n_z are equivalent to the attributes `NormalX`, `NormalY`, `NormalZ`.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Circle2d` | `plot::Cone` | `plot::Sphere`

plot::Cone

Cones and frustums

Syntax

```
plot::Cone(br, [bx, by, bz], <tr>, [tx, ty, tz], <a = amin .. amax>, options)
```

Description

`plot::Cone(br, [ bx, by, bz], [ tx, ty, tz])` creates a cone stretching from the base with radius `br` and center `[bx, by, bz]` to the top `[tx, ty, tz]`.

`plot::Cone(br, [ bx, by, bz], tr, [ tx, ty, tz])` creates a conical frustum from the base center `[bx, by, bz]` to the top center `[tx, ty, tz]`. The base radius is `br`, the top radius is `tr`.

The lower center and upper center of the cone can also be passed as vectors.

The optional "top radius" `tr` for creating a frustum may also be specified as the attribute `TopRadius = tr`.

The upper and lower faces of a cone/frustum are not filled. They can be added as filled `plot::Circle3d`s.

Note that only circular cones can be created with `plot::Cone`. For elliptical bases, use a `plot::Surface` primitive or apply a `plot::Scale3d` transformation.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Base`	base center of cones, cylinders, pyramids and prisms	`[0, 0, 0]`

Attribute	Purpose	Default Value
`BaseX`	x-coordinate of top center of cones, cylinders, pyramids and prisms	`0`
`BaseY`	y-coordinate of top center of cones, cylinders, pyramids and prisms	`0`
`BaseZ`	z-coordinate of top center of cones, cylinders, pyramids and prisms	`0`
`BaseRadius`	base radius of cones/conical frustums and pyramids/ frustums of pyramids	`1`
`Color`	the main color	`RGB::LightBlue`
`Filled`	filled or transparent areas and surfaces	`TRUE`
`FillColor`	color of areas and surfaces	`RGB::LightBlue`
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`LineColor`	color of lines	`RGB::Black.[0.25]`
`LineWidth`	width of lines	`0.35`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LinesVisible`	visibility of lines	`TRUE`
`LineColorDirection`	the direction of color transitions on lines	[0, 0, 1]
`LineColorDirectionX`	x-component of the direction of color transitions on lines	`0`

Attribute	Purpose	Default Value
LineColorDirectionY	y-component of the direction of color transitions on lines	0
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
Shading	smooth color blend of surfaces	Smooth
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Top	top center of cones, cylinders, pyramids and prisms	[0, 0, 1]

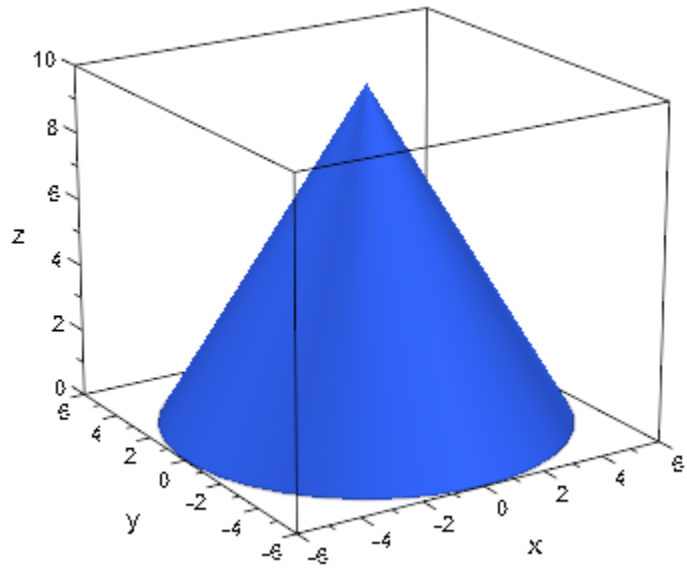
Attribute	Purpose	Default Value
`TopX`	base and top center of cones, cylinders, pyramids and prisms	0
`TopY`	base and top center of cones, cylinders, pyramids and prisms	0
`TopZ`	base and top center of cones, cylinders, pyramids and prisms	1
`TopRadius`	top radius of cones/conical frustums and pyramids/ frustums of pyramids	0
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

We draw a cone with base radius 6:

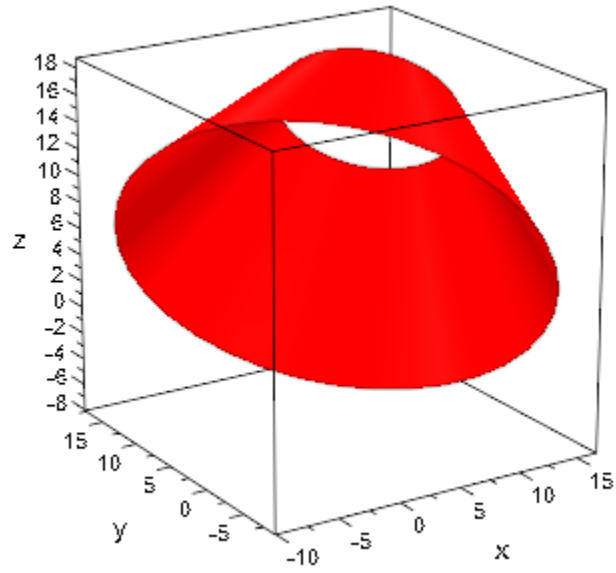
```
plot(plot::Cone(6, [0, 0, 0], [0, 0, 10])):
```



Example 2

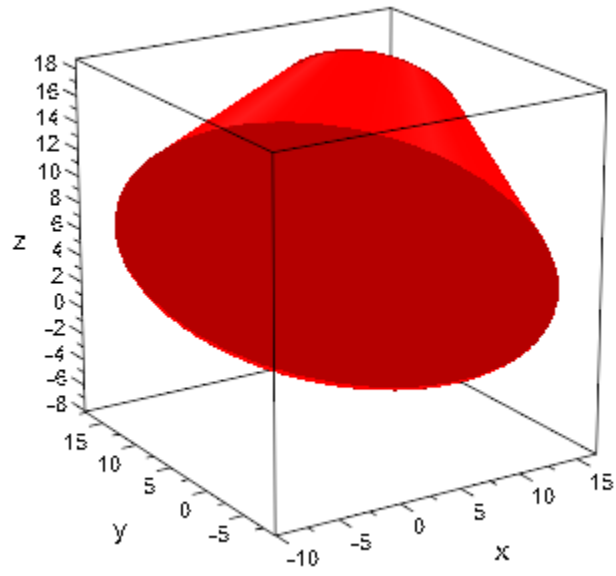
We create a conical frustum by specifying a non-zero top radius. Note that no discs are attached to the base and the top. You can look through the frustum:

```
br := 16: base := [3, 4, 5]:
tr:= 7: top := [11, 12, 13]:
plot(plot::Cone(br, base, tr, top, FillColor = RGB::Red)):
```



We add the discs at the base and the top. Their normals `n` are given as the vector from the base to the top:

```
n := zip(top, base, _subtract):
plot(plot::Circle3d(br, base, n, Filled = TRUE),
     plot::Circle3d(tr, top, n, Filled = TRUE),
     plot::Cone(br, base, tr, top),
     LinesVisible = FALSE, FillColor = RGB::Red):
```

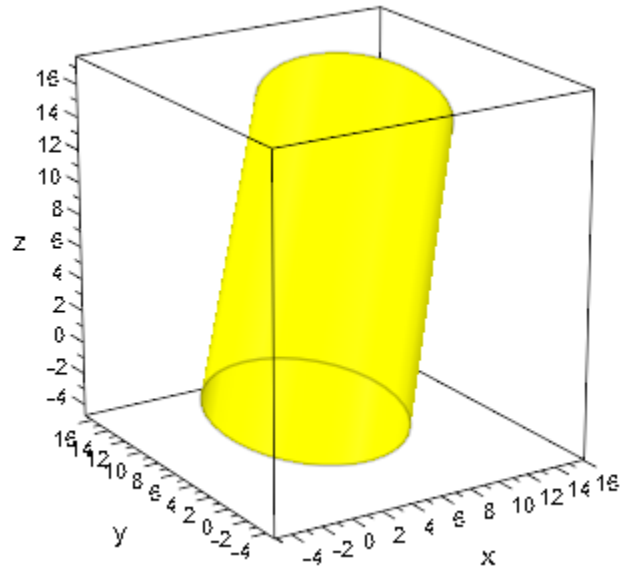



```
delete br, base, tr, top, n:
```

Example 3

A tube or cylinder (in the mathematical sense, i.e., the lateral sides of a physical cylinder) is a special case of a conical frustum with the same top and bottom radius:

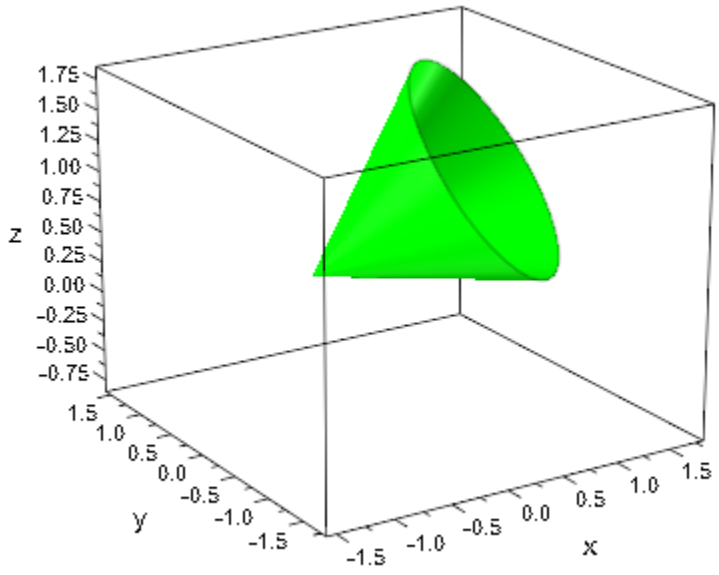
```
plot(plot::Cone(6, [0, 0, 0], 6, [11, 12, 13],
                FillColor = RGB::Yellow,
                LinesVisible = TRUE)):
```



Example 4

Bottom and top radii and centers can be animated:

```
plot(plot::Cone(sin(a)^2, [sin(2*a), cos(2*a), 0],
                cos(a)^2, [cos(2*a), sin(2*a), 1],
                a = 0..PI, FillColor = RGB::Green)):
```



Parameters

`br`

The base radius of the cone. This must be a real numerical value or an arithmetical expression of the animation parameter `a`.

br is equivalent to the attribute `BaseRadius`.

b_x, b_y, b_z

The lower center point. The coordinates b_x, b_y, b_z must be real numerical values or arithmetical expressions of the animation parameter `a`.

b_x, b_y, b_z are equivalent to the attributes `BaseX`, `BaseY`, `BaseZ`.

`tr`

The top radius of the cone/conical frustum. This must be a real numerical value or an arithmetical expression of the animation parameter `a`. If no top radius is specified, a cone with top radius $tr = 0$ is created.

tr is equivalent to the attribute `TopRadius`.

t_x, t_y, t_z

The upper center point. The coordinates t_x, t_y, t_z must be real numerical values orarithmetical expressions of the animation parameter `a`.

t_x, t_y, t_z are equivalent to the attributes `TopX`, `TopY`, `TopZ`.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions

`plot` | `plot::copy`

MuPAD Graphical Primitives

`plot::Cylinder` | `plot::Prism` | `plot::Pyramid`

plot::Conformal

(complex-valued) conformal function plot

Syntax

```
plot::Conformal(f, z = z1 .. z2, <a = amin .. amax>, options)
```

Description

`plot::Conformal(f(z), z = z_1..z_2 )` is a plot of the conformal function *f* over the complex interval `z_1..z_2`.

`plot::Conformal` creates plots of (conformal) complex-valued functions of one complex variable. They are displayed by showing the image of a rectangular grid over an interval.

By default, the attribute `LineColorType = Flat` is set. All curves are displayed with the color given by the attribute `LineColor` (or `Color` for short).

When specifying the attribute `LineColorType = Dichromatic`, a color blend from `LineColor` to `LineColor2` is used ("height coloring").

When specifying the attribute `LineColorType = Functional` without specifying a `LineColorFunction`, all curves parametrized by the real part of the pre-image points are displayed with the flat color `LineColor`, whereas all curves parametrized by the imaginary part of the pre-image points are displayed with the flat color `LineColor2`.

A user defined `LineColorFunction` is a procedure `(z, x, y, flag) -> RGB-color` that will be called with complex floating-point arguments `z` from the range of pre-images of the conformal function `f`, the real floating point values `x = Re(f(z))`, `y = Im(f(z))`, and the integer value `flag` which has the values 1 or 2. The flag value 1 determines the color of the curves parametrized by the real part of `z`, the flag value 2 determines the color of the curves parametrized by the imaginary part of `z`. The color function must return an `RGB` color, i.e., a list of 3 real floating point values between 0.0 and 1.0. For example,

```
LineColorFunction = proc(z, x, y, flag)
  begin
    if flag = 1 then
      return(RGB::Blue)
```

```
    else
      return(RGB::Red)
    end_if;
  end_proc
```

displays all curves parametrized by `Re(z)` in blue, while the orthogonal curves, parametrized by `Im(z)`, are displayed in red.

See the examples in the documentation of `RGB` for another way of displaying complex functions.

Attributes

Attribute	Purpose	Default Value
`AdaptiveMesh`	adaptive sampling	`0`
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`
`Color`	the main color	`RGB::Blue`
`Frames`	the number of frames in an animation	`50`
`Function`	function expression or procedure	
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`TRUE`
`LineColor`	color of lines	`RGB::Blue`
`LineWidth`	width of lines	`0.35`
`LineColor2`	color of lines	`RGB::DeepPink`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LinesVisible`	visibility of lines	`TRUE`
`LineColorType`	line coloring types	`Flat`

Attribute	Purpose	Default Value
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	1
Mesh	number of sample points	[11, 11]
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointsVisible	visibility of mesh points	FALSE
Submesh	density of submesh (additional sample points)	[0, 0]
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center

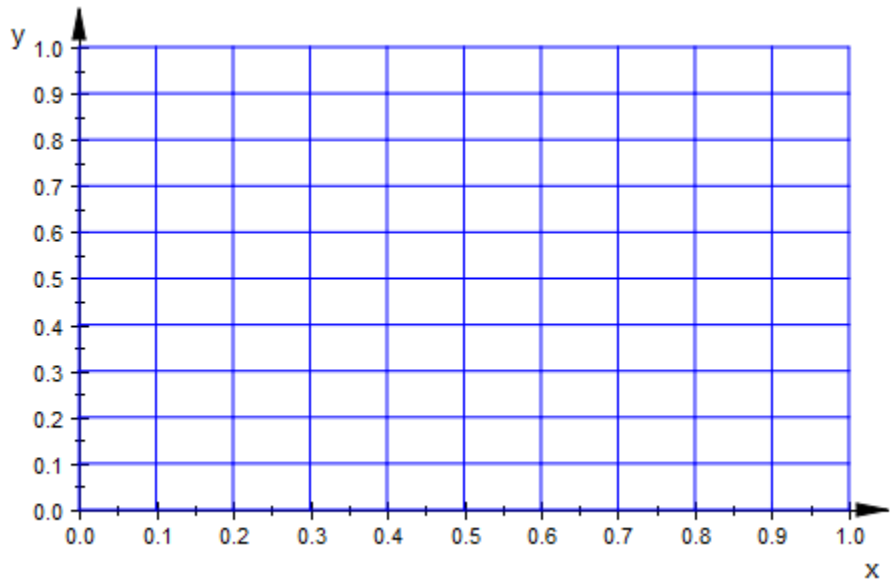
Attribute	Purpose	Default Value
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XMesh	number of sample points for parameter “x”	11
XSubmesh	density of additional sample points for parameter “x”	0
YMesh	number of sample points for parameter “y”	11
YSubmesh	density of additional sample points for parameter “y”	0
ZMax	final value of parameter “z”	
ZMin	initial value of parameter “z”	
ZName	name of parameter “z”	
ZRange	range of parameter “z”	

Examples

Example 1

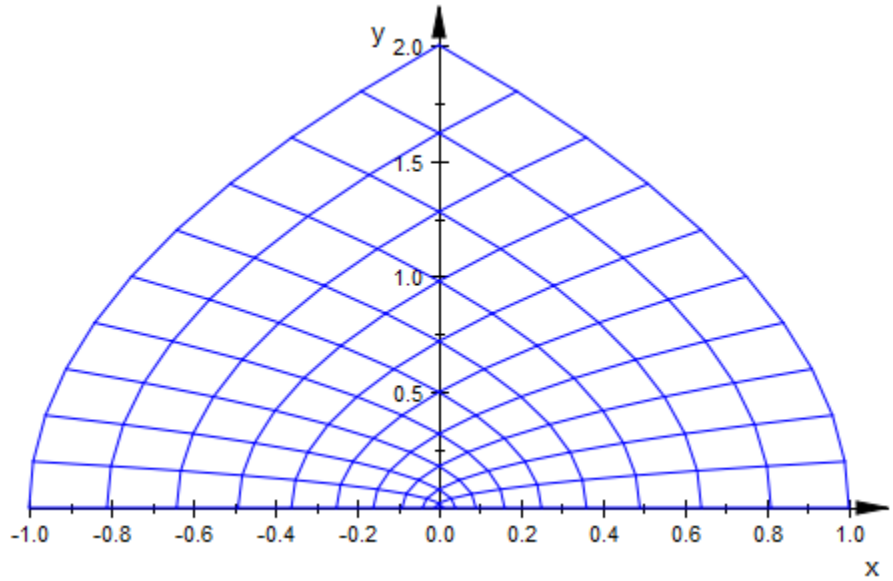
By plotting the identity function, we are presented the pre-image used by `plot::Conformal`:

```
plot(plot::Conformal(z, z = 0..1+I))
```



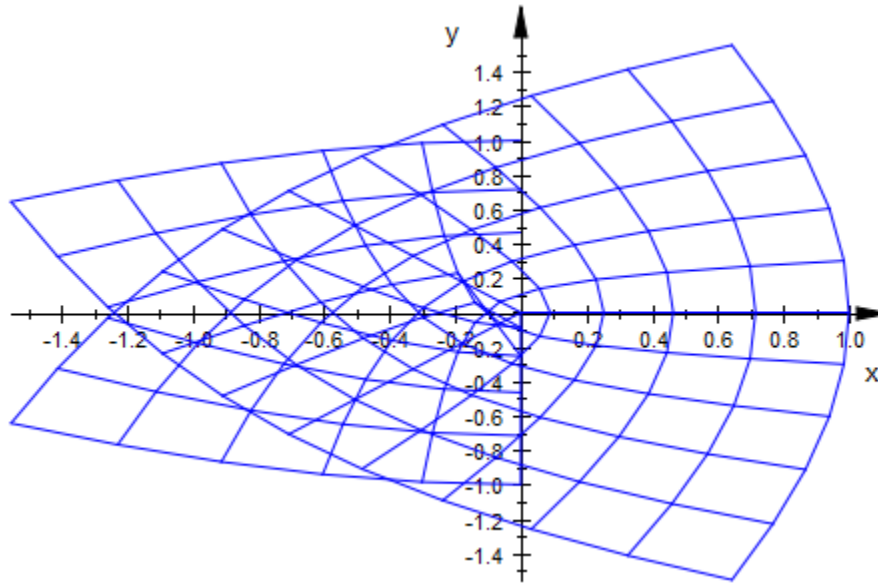
The important property of conformal functions, as far as plots are concerned, is that orthogonal lines are mapped onto curves meeting orthogonally:

```
plot(plot::Conformal(z^2, z = 0..1+I))
```



This property allows to visually detect overlapping regions (in some cases); in the following example this is the case in the left semi-plane:

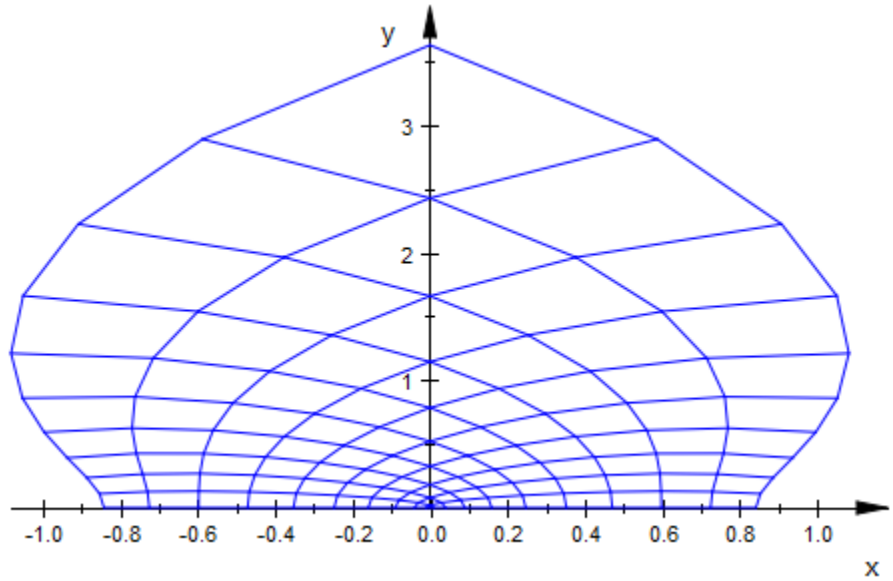
```
plot(plot::Conformal(z^(3/2), z = -1-I..1+I))
```



Example 2

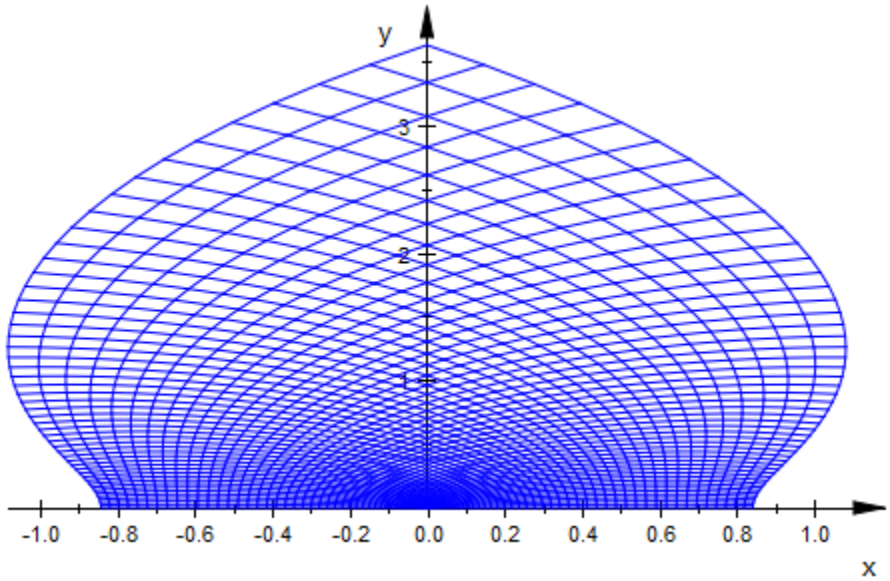
The default mesh may in some cases be too coarse:

```
plot(plot::Conformal(sin(z^2), z = 0..1+I))
```



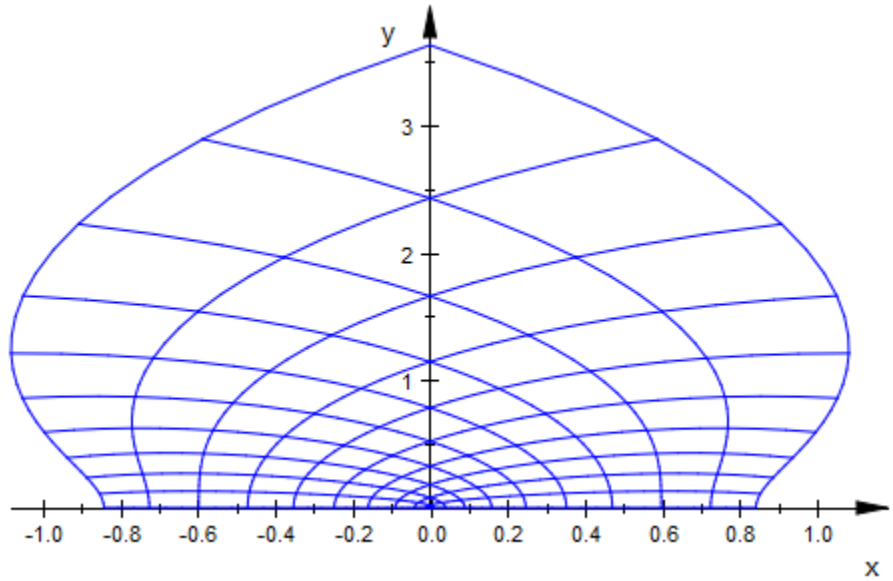
There are at least three ways of improving this plot. Firstly, we can set `Mesh` to a higher value:

```
plot(plot::Conformal(sin(z^2), z = 0..1+I, Mesh = [50, 50]))
```



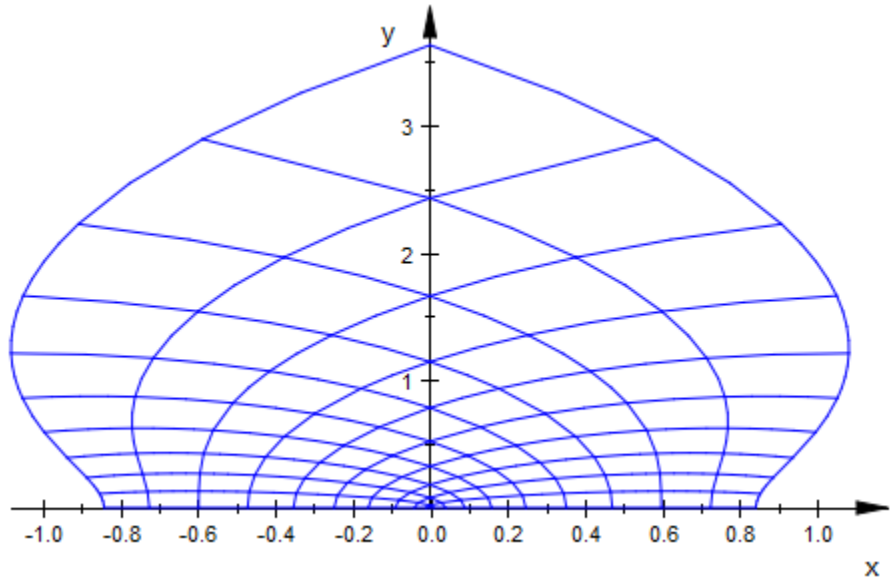
Another option would be to increase `Submesh` to get smoother, not more, lines:

```
plot(plot::Conformal(sin(z^2), z = 0..1+I, Submesh = [2, 2]))
```



Finally, we can also ask for an adaptive refinement of the submesh by setting `AdaptiveMesh` to some positive value:

```
plot(plot::Conformal(sin(z^2), z = 0..1+I, AdaptiveMesh = 2))
```



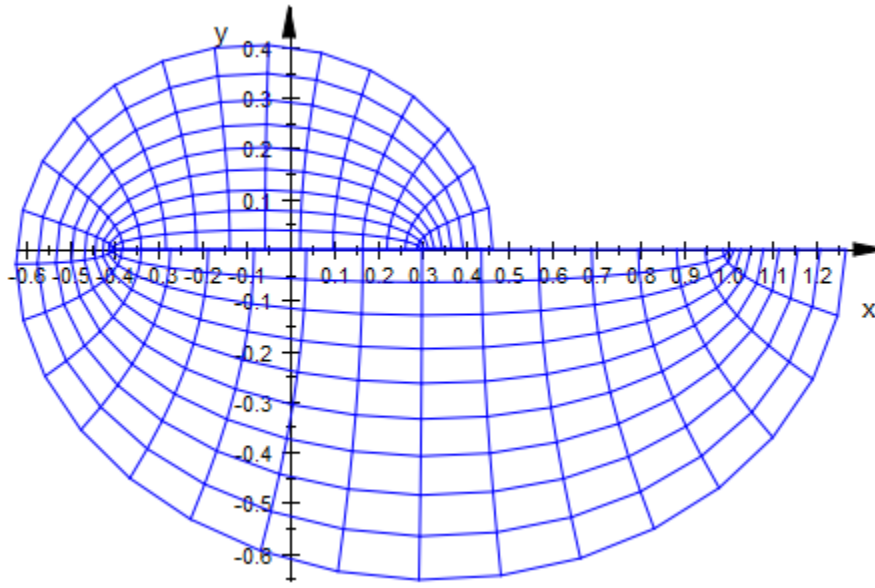
Example 3

Here is the image of the complex rectangle $0 \le Re(z) \le x$, $0 \le \Im(z) \le 1$ under the map `z→besselJ(0, z)`. We choose x as the second positive root of `Im(besselJ(0, x + I))`:

```
numeric::solve(Im(besselJ(0, x + I)), x = i .. i+1) $ i = 0..7
```

$$\{0.0\}, \varnothing, \varnothing, \{3.791394324\}, \varnothing, \varnothing, \{6.993364687\}, \varnothing$$

```
plot(plot::Conformal(besselJ(0, z), z = 0 .. 6.9934 + I,
                     Mesh = [31, 10]))
```



Parameters

`f`

An expression in `z` and the animation parameter, if present. Expected to be conformal in `z`.

f is equivalent to the attribute `Function`.

`z`

The independent variable: An identifier or indexed identifier.

z is equivalent to the attribute `ZName`.

z_1 .. z_2

The (complex) range over which `f` should be plotted: z_1 and z_2 should be complex-valued expressions, possibly in the animation parameter.

z_1 .. z_2 is equivalent to the attribute `ZRange`.

a

Animation parameter, specified as a = a_{min}..a_{max}, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Curve2d` | `plot::Function2d`

plot::Curve2d

Parameterized 2D curves

Syntax

`plot::Curve2d([x, y], t = tmin .. tmax, <a = amin .. amax>, options)`

`plot::Curve2d(A2d, t = tmin .. tmax, <a = amin .. amax>, options)`

`plot::Curve2d(piecewiseF(t), t = tmin .. tmax, <a = amin .. amax>, options)`

Description

`plot::Curve2d([x(t), y(t)], t = tmin.. tmax)` creates the planar curve

$$\{(x(t), y(t)) \mid t_{\min} \le t \le t_{\max}\}.$$

`plot::Curve2d` and `plot::Curve3d` construct curves in one parameter (see “Example 1” on page 24-254), possibly animated (see “Example 2” on page 24-255). The curves may contain poles, in which case automatic clipping is used by default (see “Example 4” on page 24-259).

By default, curves are sampled at equidistant values of the parameter `t`. The attribute `AdaptiveMesh` can be used to change this behavior, such that a denser sampling rate is used in areas of higher curvature. Cf. “Example 5” on page 24-261.

Curves are graphical objects that can be manipulated, see the examples and the documentation of the parameters listed below for details.

Attributes

Attribute	Purpose	Default Value
`AdaptiveMesh`	adaptive sampling	`0`
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`

Attribute	Purpose	Default Value
AntiAliased	antialiased lines and points?	TRUE
Color	the main color	RGB::Blue
DiscontinuitySearch	semi-symbolic search for discontinuities	TRUE
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	TRUE
LineColor	color of lines	RGB::Blue
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	1
Mesh	number of sample points	121
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	

Attribute	Purpose	Default Value
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE
Submesh	density of submesh (additional sample points)	0
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
UMax	final value of parameter “u”	5
UMesh	number of sample points for parameter “u”	121
UMin	initial value of parameter “u”	-5
UName	name of parameter “u”	

Attribute	Purpose	Default Value
URange	range of parameter “u”	-5 .. 5
USubmesh	density of additional sample points for parameter “u”	0
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XFunction	function for x values	
YFunction	function for y values	

Examples

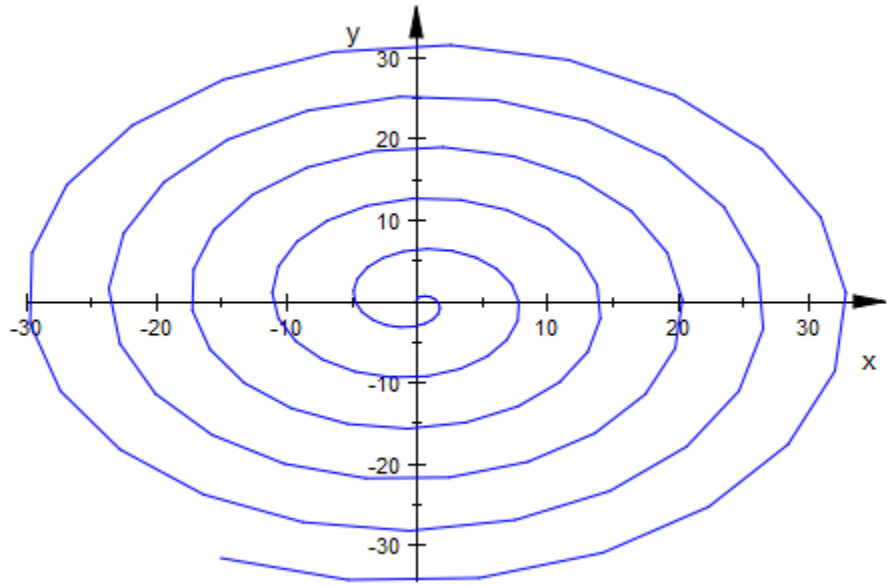
Example 1

Archimedes' Spiral is defined by $f(r) = (r\,sin(r), r\,cos(r))$. The corresponding call to `plot::Curve2d` reads:

```
curve := plot::Curve2d([r*sin(r), r*cos(r)], r = 0..35)
```

$$\text{plot::Curve2d}([r\,\sin(r),\ r\,\cos(r)],\ r = 0..35)$$

```
plot(curve)
```

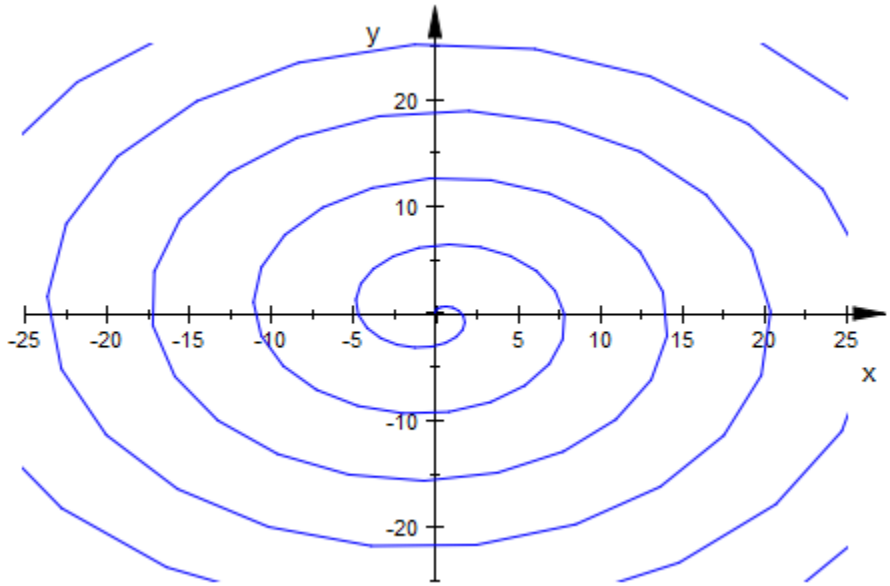


Note that this particular example is even more straightforward to plot using `plot::Polar`.

Example 2

Continuing the example from above, we define an easy animation by making the angular part time-dependent:

```
curve := plot::Curve2d([r*sin(r-t), r*cos(r-t)],
                       r = 0..35, t = 0..2*PI,
                       TimeEnd = 5,
                       ViewingBox = [-25..25, -25..25]):
plot(curve)
```

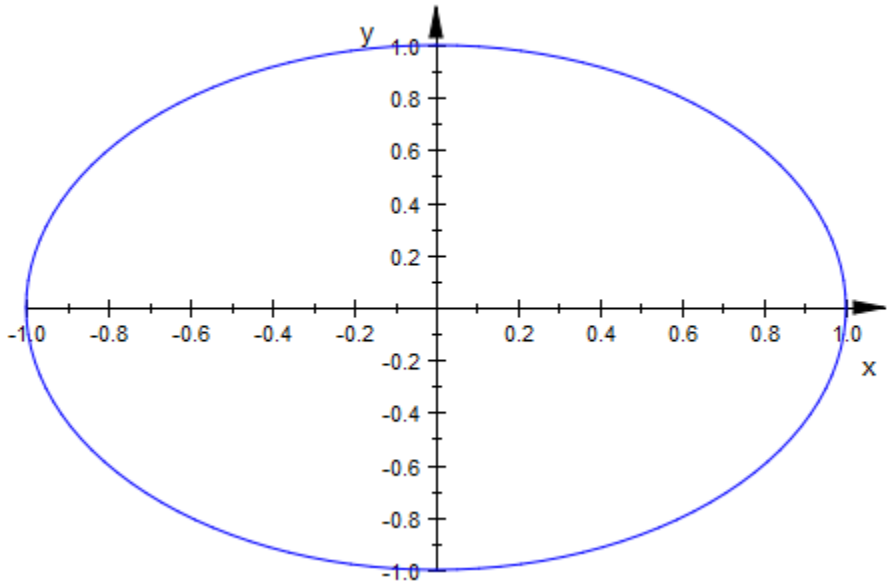


Note that to start the animation, you have to double-click the image in the notebook and choose “Start” from the “Animation” menu.

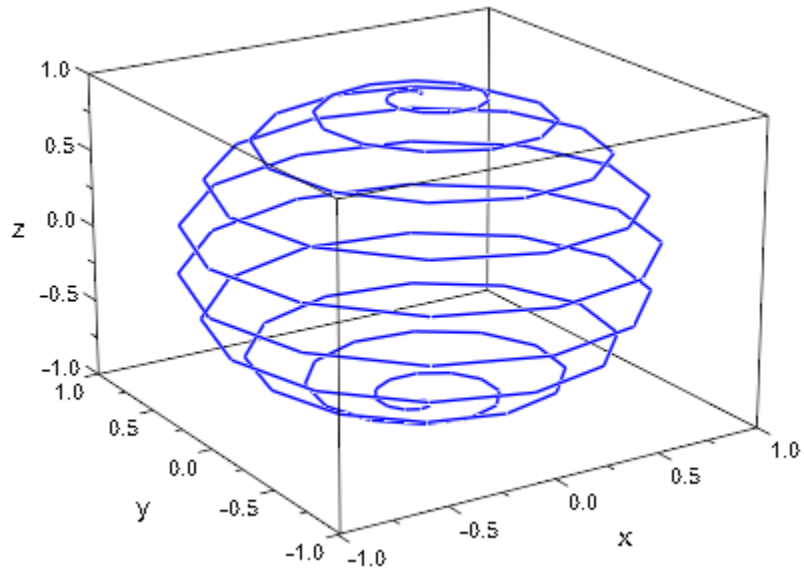
Example 3

Another useful and easy type of animation is achieved by animating the parameter range. This creates the illusion of the curve being drawn in real time:

```
curve := plot::Curve2d([sin(thet), cos(thet)],
                       thet = 0..a,
                       a = 0..2*PI):
plot(curve)
```

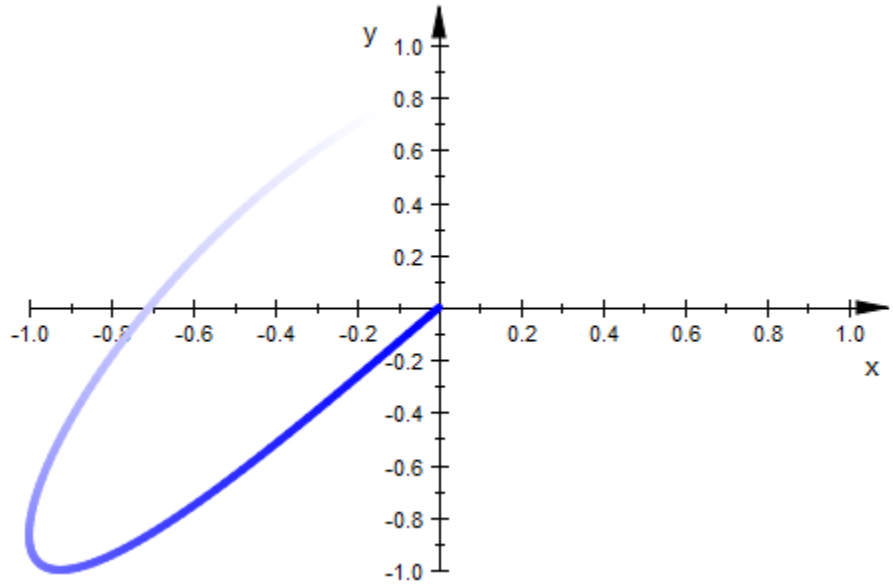



```
curve := plot::Curve3d([sin(thet)*cos(20*thet),
                        sin(thet)*sin(20*thet),
                        cos(thet)],
                       thet = 0..a,
                       a = 0..PI):
plot(curve)
```



Combining this with an animated `LineColorFunction`, you can even simulate motion:

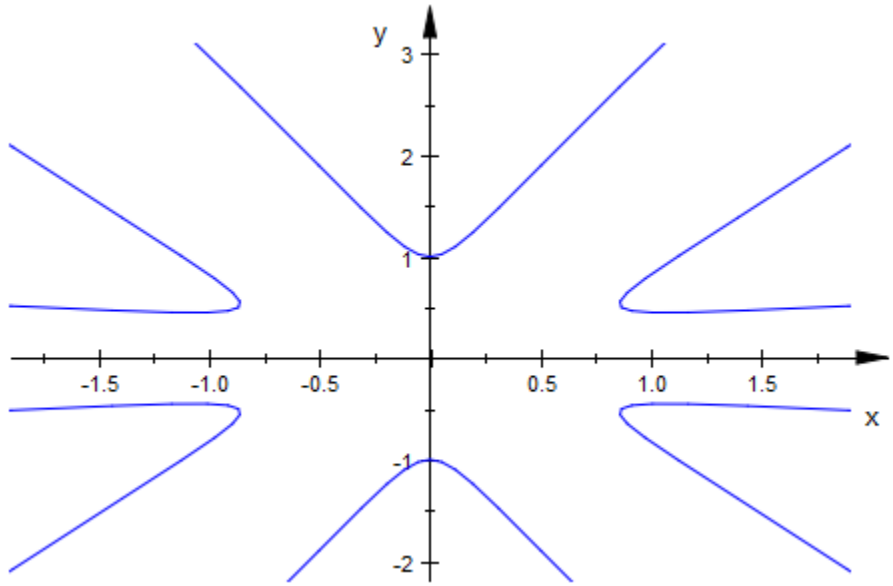
```
colorfunc := (thet, x, y, a) -> [a-thet, a-thet, 1.0]:
curve := plot::Curve2d([sin(3*thet), sin(4*thet)],
                       thet = a-1..a,
                       LineColorFunction = colorfunc,
                       LineWidth = 1,
                       a = 0..2*PI):
plot(curve)
```



Example 4

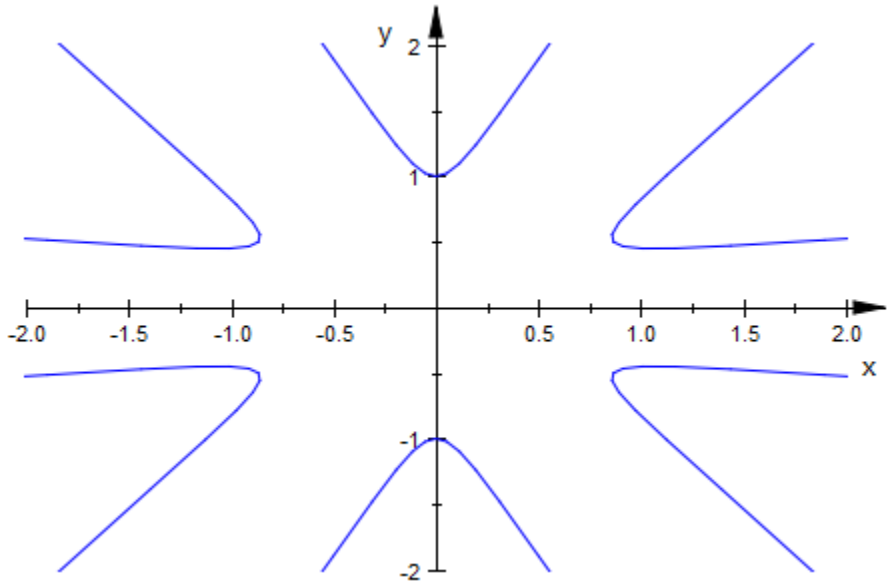
Curves with poles are automatically clipped:

```
curve := plot::Curve2d([(1+tan(3*t)^2)*sin(t),
                        (1+tan(3*t)^2)*cos(t)],
                       t = 0..2*PI):
plot(curve);
```



If the automatically chosen viewing box is not to your liking, you can explicitly set other values:

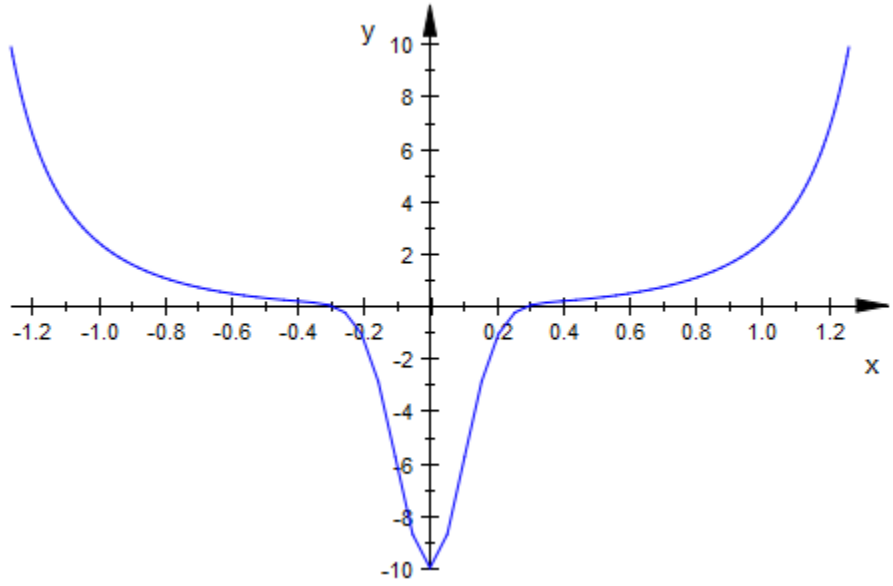
```
curve::ViewingBox := [-2..2, -2..2]:
plot(curve)
```



Example 5

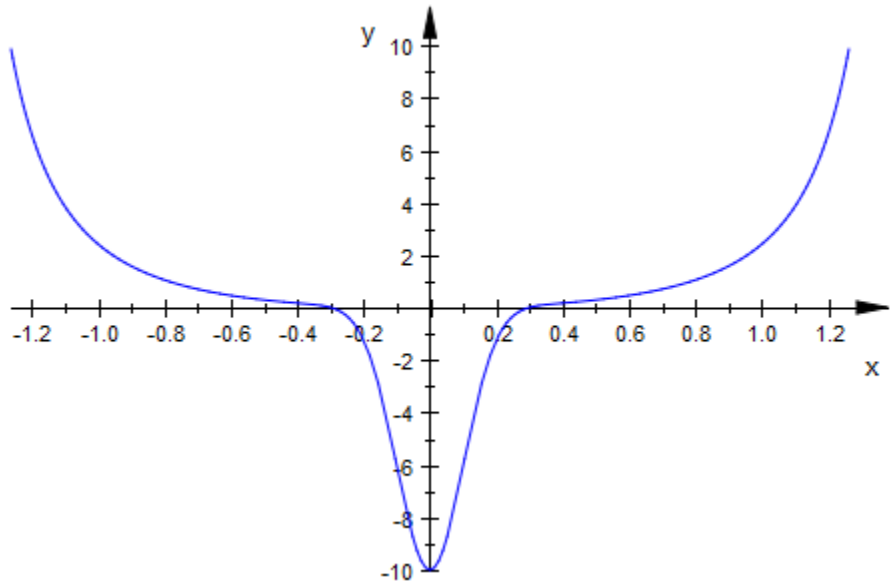
By default, curves are drawn by evaluating at equidistant values of the curve parameter. For curves that have few regions of high curvature, this may be inappropriate:

```
plot(plot::Curve2d([arctan(t), t^2-10*exp(-50*t^2)],
                   t = -PI..PI))
```



Note the hard “kink” at the bottom of the picture. On the other hand, the remainder of the curve is sufficiently smooth, so globally increasing the number of evaluation points is not desirable. `AdaptiveMesh` makes `plot::Curve2d` look for these kinks and adaptively increase the mesh density in problematic areas:

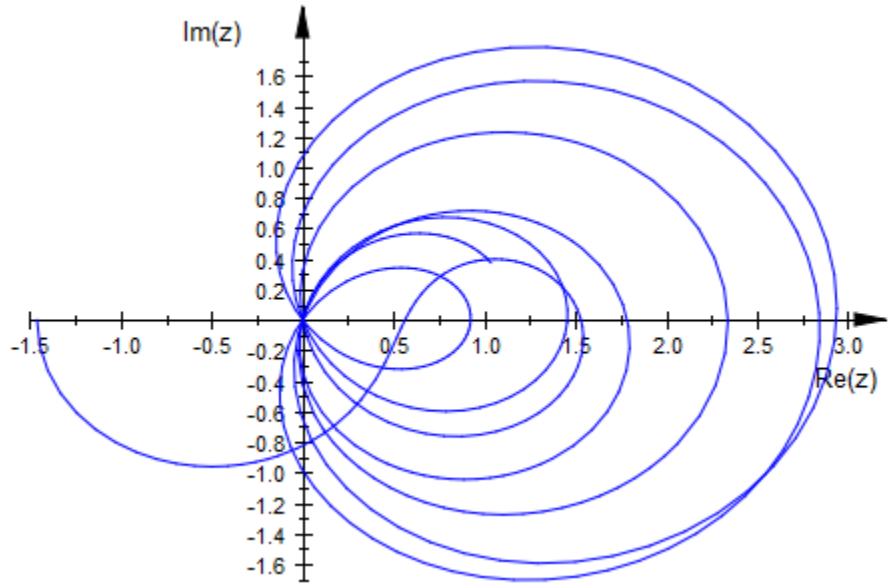
```
plot(plot::Curve2d([arctan(t), t^2-10*exp(-50*t^2)],
                   t = -PI..PI, AdaptiveMesh = 2))
```



Example 6

To display a curve in the complex plane, map the list-valued function `[Re, Im]` to the curve:

```
plot(plot::Curve2d([Re, Im](zeta(I*y+1/2)), y=0..42,
                   AdaptiveMesh=3),
     XAxisTitle = "Re(z)", YAxisTitle = "Im(z)")
```



Example 7

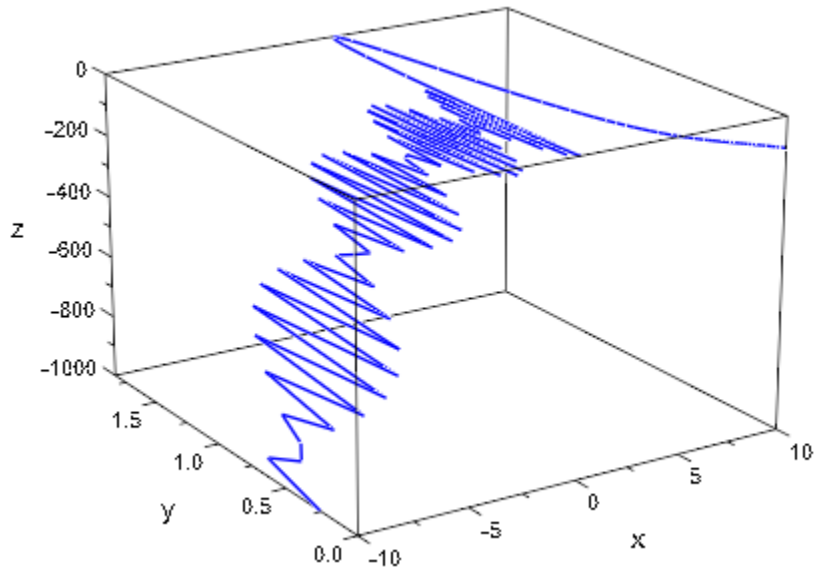
Create the following piecewise function:

```
f := piecewise([t < 0, [t, sin(10*t)^2, t^3]],
               [t >= 0, [t, 5*t/exp(t), -t^2]])
```

$$\begin{cases} [t,\ \sin(10\,t)^2,\ t^3] & \text{if } t < 0 \\ \left[t,\ 5\,t\,\mathrm{e}^{-t},\ -t^2\right] & \text{if } 0 \le t \end{cases}$$

Now, plot this function:

```
plot(plot::Curve3d(f, t = -10..10))
```



Parameters

`x, y`

Real-valued expressions in `t` (and possibly the animation parameter)

A_{2d}

A matrix of category `Cat::Matrix` with two entries that provide the parametrization `x`, `y` of a 2D curve

`piecewiseF(t)`

A `piecewise` object

`t`

An identifier or an indexed identifier

t_{min}, t_{max}

Real-valued expressions (possibly in the animation parameter)

a

Animation parameter, specified as $a = a_{min}..a_{max}$, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Curve3d` | `plot::Function2d` | `plot::Function3d` | `plot::Polygon2d` | `plot::Polygon3d` | `plot::Surface`

plot::Curve3d

Parameterized 3D curves

Syntax

`plot::Curve3d([x, y, z], t = tmin .. tmax, <a = amin .. amax>, options)`

`plot::Curve3d(A3d, t = tmin .. tmax, <a = amin .. amax>, options)`

`plot::Curve3d(piecewiseF(t), t = tmin .. tmax, <a = amin .. amax>, options)`

Description

`plot::Curve3d([x(t), y(t), z(t)], t = tmin.. tmax)` creates the space curve

$$\{(x(t), y(t), z(t)) \mid t_{\min} \le t \le t_{\max}\}.$$

`plot::Curve2d` and `plot::Curve3d` construct curves in one parameter (see “Example 1” on page 24-270), possibly animated (see “Example 2” on page 24-271). The curves may contain poles, in which case automatic clipping is used by default (see “Example 4” on page 24-275).

By default, curves are sampled at equidistant values of the parameter `t`. The attribute `AdaptiveMesh` can be used to change this behavior, such that a denser sampling rate is used in areas of higher curvature. Cf. “Example 5” on page 24-277.

Curves are graphical objects that can be manipulated, see the examples and the documentation of the parameters listed below for details.

Attributes

Attribute	Purpose	Default Value
`AdaptiveMesh`	adaptive sampling	`0`
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`

Attribute	Purpose	Default Value
`Color`	the main color	`RGB::Blue`
`DiscontinuitySearch`	semi-symbolic search for discontinuities	`TRUE`
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`TRUE`
`LineColor`	color of lines	`RGB::Blue`
`LineWidth`	width of lines	`0.35`
`LineColor2`	color of lines	`RGB::DeepPink`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LinesVisible`	visibility of lines	`TRUE`
`LineColorType`	line coloring types	`Flat`
`LineColorFunction`	functional line coloring	
`LineColorDirection`	the direction of color transitions on lines	`[0, 0, 1]`
`LineColorDirectionX`	x-component of the direction of color transitions on lines	`0`
`LineColorDirectionY`	y-component of the direction of color transitions on lines	`0`
`LineColorDirectionZ`	z-component of the direction of color transitions on lines	`1`
`Mesh`	number of sample points	`121`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	

Attribute	Purpose	Default Value
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE
Submesh	density of submesh (additional sample points)	0
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
UMax	final value of parameter “u”	5
UMesh	number of sample points for parameter “u”	121

Attribute	Purpose	Default Value
`UMin`	initial value of parameter "u"	`-5`
`UName`	name of parameter "u"	
`URange`	range of parameter "u"	`-5 .. 5`
`USubmesh`	density of additional sample points for parameter "u"	`0`
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`
`XFunction`	function for x values	
`YFunction`	function for y values	
`ZFunction`	function for z values	

Examples

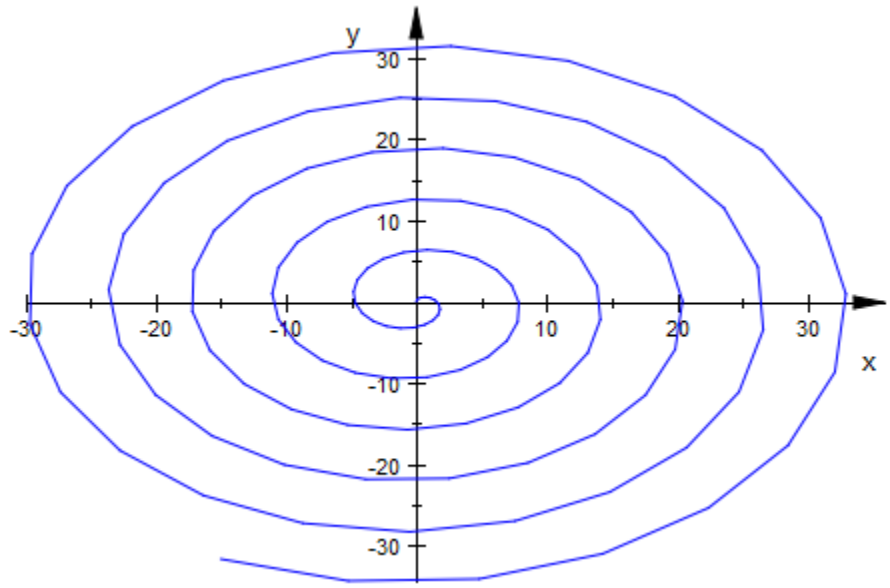
Example 1

Archimedes' Spiral is defined by $f(r) = (r\,sin(r), r\,cos(r))$. The corresponding call to `plot::Curve2d` reads:

```
curve := plot::Curve2d([r*sin(r), r*cos(r)], r = 0..35)
```

$$\text{plot::Curve2d}([r\,\sin(r),\ r\,\cos(r)],\ r = 0..35)$$

```
plot(curve)
```

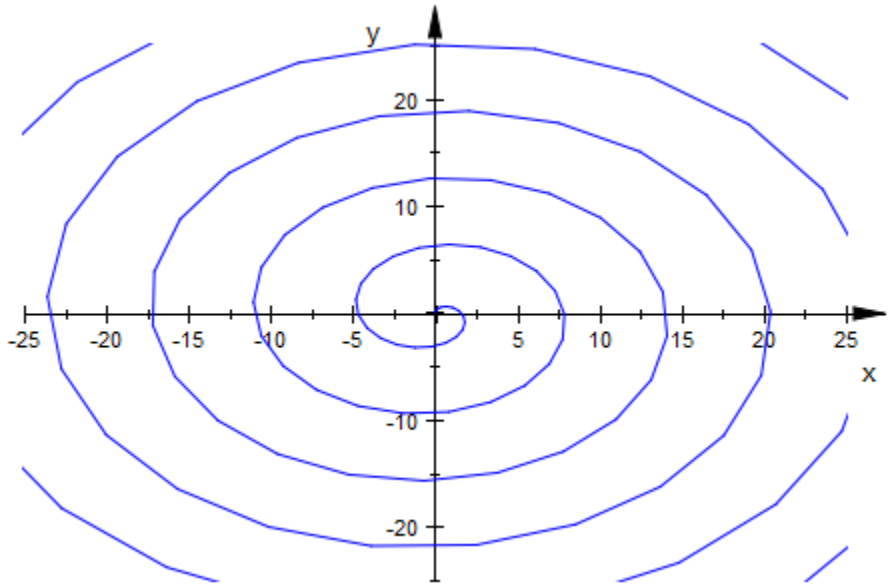


Note that this particular example is even more straightforward to plot using `plot::Polar`.

Example 2

Continuing the example from above, we define an easy animation by making the angular part time-dependent:

```
curve := plot::Curve2d([r*sin(r - t), r*cos(r - t)],
                       r = 0..35, t = 0..2*PI,
                       TimeEnd = 5,
                       ViewingBox = [-25..25, -25..25]):
plot(curve)
```

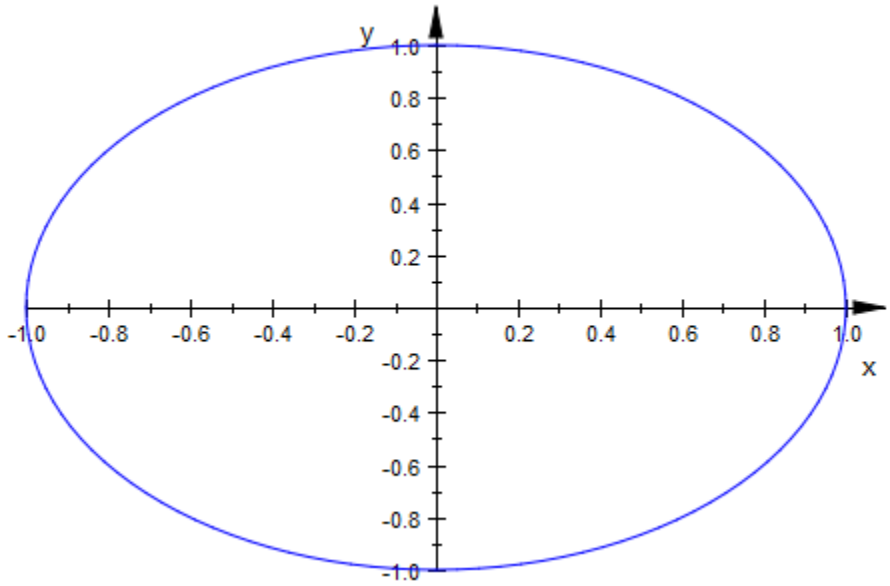


Note that to start the animation, you have to double-click the image in the notebook and choose “Start” from the “Animation” menu.

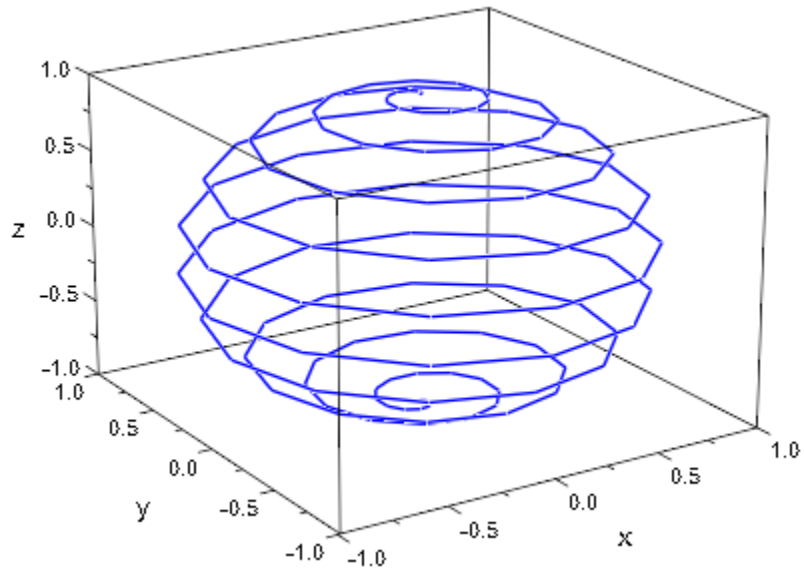
Example 3

Another useful and easy type of animation is achieved by animating the parameter range. This creates the illusion of the curve being drawn in real time:

```
curve := plot::Curve2d([sin(thet), cos(thet)],
                       thet = 0..a,
                       a = 0..2*PI):
plot(curve)
```

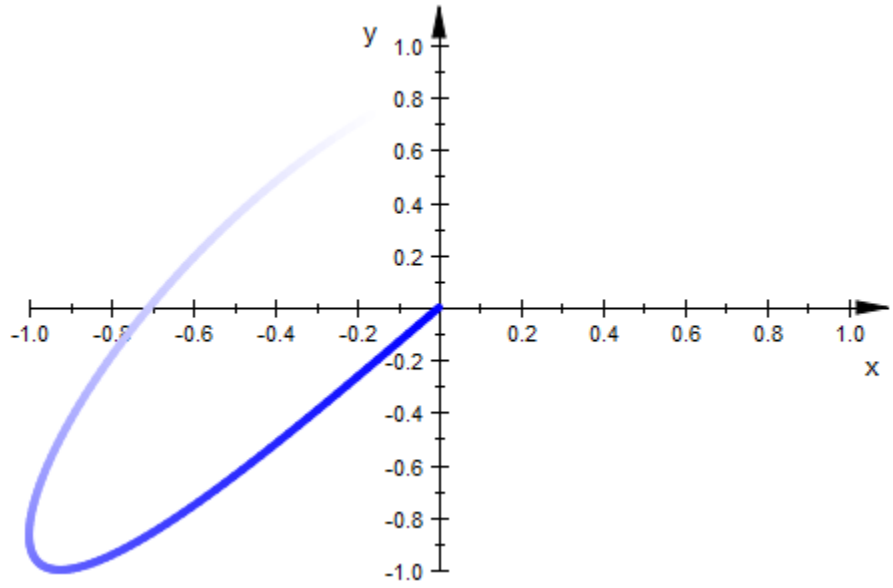



```
curve := plot::Curve3d([sin(thet)*cos(20*thet),
                        sin(thet)*sin(20*thet),
                        cos(thet)],
                       thet = 0..a,
                       a = 0..PI):
plot(curve)
```



Combining this with an animated `LineColorFunction`, you can even simulate motion:

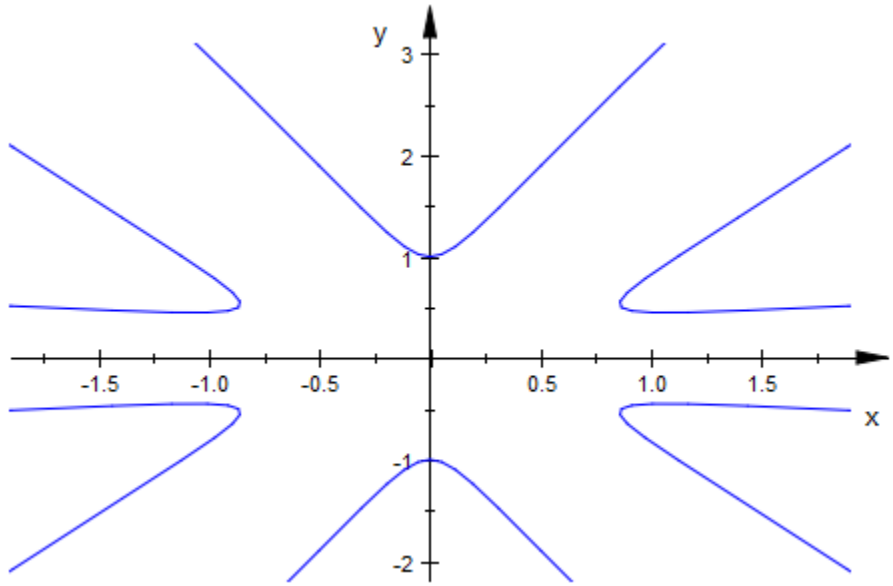
```
colorfunc := (thet, x, y, a) -> [a-thet, a-thet, 1.0]:
curve := plot::Curve2d([sin(3*thet), sin(4*thet)],
                       thet = a - 1..a,
                       LineColorFunction = colorfunc,
                       LineWidth = 1,
                       a = 0..2*PI):
plot(curve)
```



Example 4

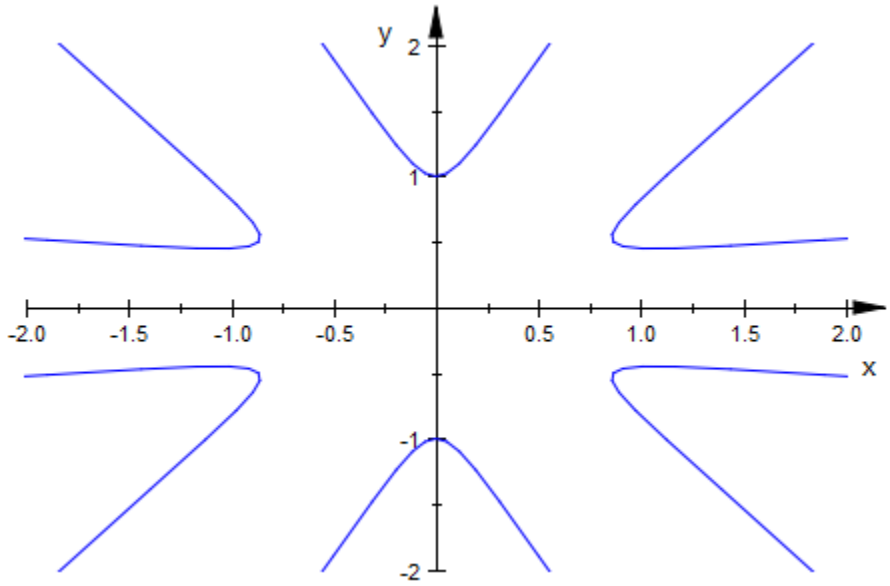
Curves with poles are automatically clipped:

```
curve := plot::Curve2d([(1 + tan(3*t)^2)*sin(t),
                        (1 + tan(3*t)^2)*cos(t)],
                                    t = 0..2*PI):
plot(curve);
```



If the automatically chosen viewing box is not to your liking, you can explicitly set other values:

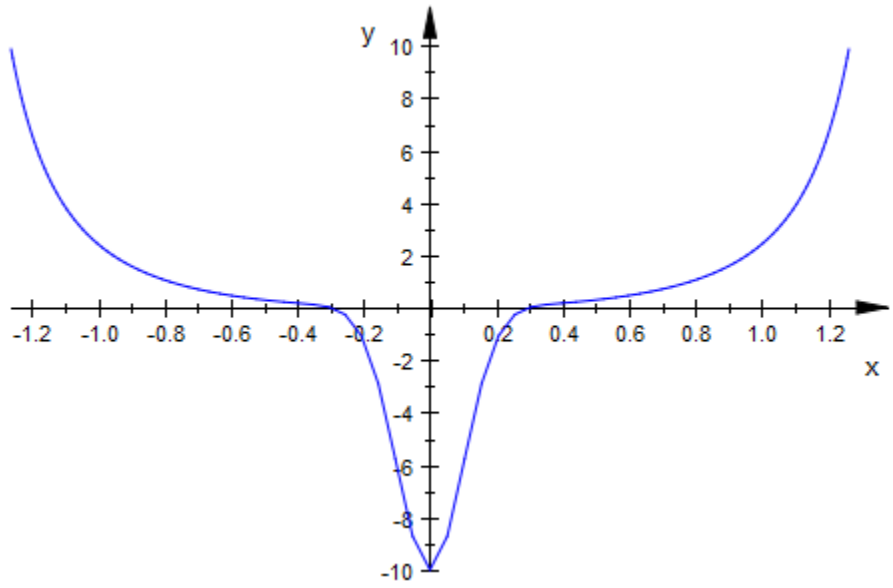
```
curve::ViewingBox := [-2..2, -2..2]:
plot(curve)
```



Example 5

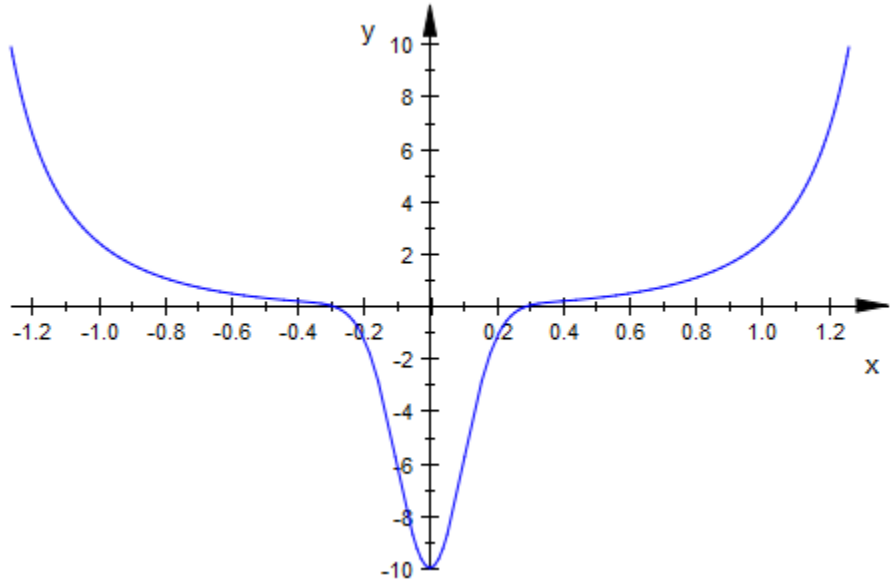
By default, curves are drawn by evaluating at equidistant values of the curve parameter. For curves that have few regions of high curvature, this may be inappropriate:

```
plot(plot::Curve2d([arctan(t), t^2 - 10*exp(-50*t^2)],
                   t = -PI..PI))
```



Note the hard “kink” at the bottom of the picture. On the other hand, the remainder of the curve is sufficiently smooth, so globally increasing the number of evaluation points is not desirable. `AdaptiveMesh` makes `plot::Curve2d` look for these kinks and adaptively increase the mesh density in problematic areas:

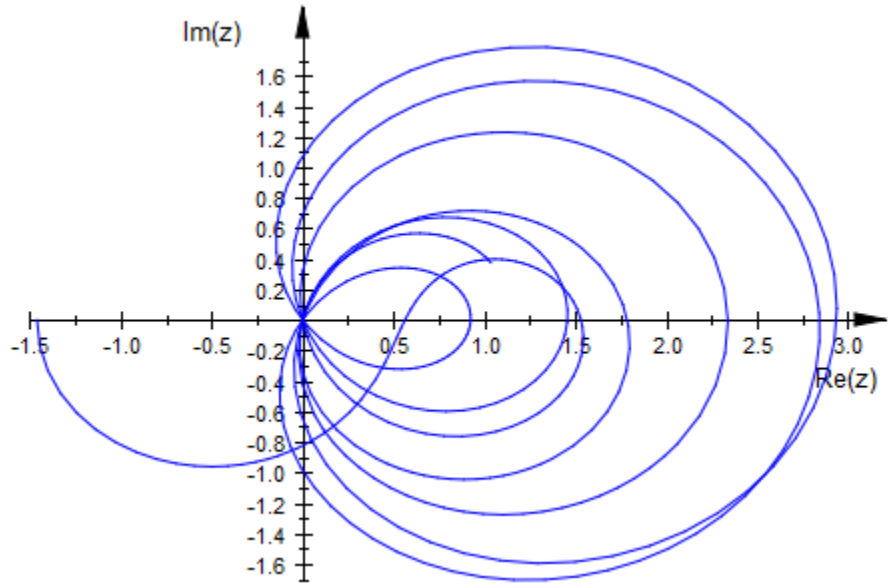
```
plot(plot::Curve2d([arctan(t), t^2 - 10*exp(-50*t^2)],
                   t = -PI..PI, AdaptiveMesh = 2))
```



Example 6

To display a curve in the complex plane, map the list-valued function `[Re, Im]` to the curve:

```
plot(
     plot::Curve2d([Re, Im](zeta(I*y + 1/2)), y = 0..42,
                                        AdaptiveMesh = 3
     ),
     XAxisTitle = "Re(z)", YAxisTitle = "Im(z)")
```



Example 7

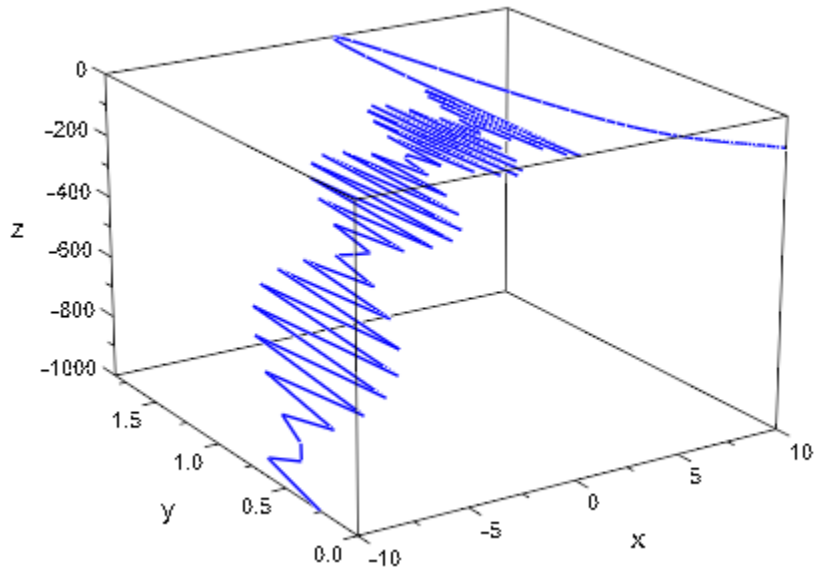
Create the following piecewise function:

```
f := piecewise([t < 0, [t, sin(10*t)^2, t^3]],
               [t >= 0, [t, 5*t/exp(t), -t^2]])
```

$$\begin{cases} [t,\ \sin(10\,t)^2,\ t^3] & \text{if } t < 0 \\ \left[t,\ 5\,t\,\mathrm{e}^{-t},\ -t^2\right] & \text{if } 0 \le t \end{cases}$$

Now, plot this function:

```
plot(plot::Curve3d(f, t = -10..10))
```



Parameters

`x, y, z`

Real-valued expressions in `t` (and possibly the animation parameter)

A_{3d}

A matrix of category `Cat::Matrix` with three entries that provide the parametrization `x`, `y`, `z` of a 3D curve

`piecewiseF(t)`

A `piecewise` object

`t`

An identifier or an indexed identifier

t_{min}, t_{max}

Real-valued expressions (possibly in the animation parameter)

a

Animation parameter, specified as a = a_{min}..a_{max}, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Curve2d` | `plot::Function2d` | `plot::Function3d` | `plot::Polygon2d` | `plot::Polygon3d` | `plot::Surface`

plot::Cylinder

Cylinders

Syntax

```
plot::Cylinder(r, [x1, y1, z1], [x2, y2, z2], <a = amin .. amax>, options)
```

Description

`plot::Cylinder(r, [x1, y1, z1] , [x2, y2, z2] )` creates a cylinder of radius *r* with an axis from the point `[x1, y1, z1]` to the point `[x2, y2, z2]`.

The base center and top center of the cylinder can also be passed as vectors.

A cylinder created by `plot::Cylinder` consists of the lateral surface and the "lids" (discs with centers `[x1, y1, z1]` and `[x2, y2, z2]`, respectively.)

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Base`	base center of cones, cylinders, pyramids and prisms	[0, 0, 0]
`BaseX`	x-coordinate of top center of cones, cylinders, pyramids and prisms	0
`BaseY`	y-coordinate of top center of cones, cylinders, pyramids and prisms	0
`BaseZ`	z-coordinate of top center of cones, cylinders, pyramids and prisms	0

Attribute	Purpose	Default Value
`Filled`	filled or transparent areas and surfaces	`TRUE`
`FillColor`	color of areas and surfaces	`RGB::LightBlue`
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`LineColor`	color of lines	`RGB::Black.[0.25]`
`LineWidth`	width of lines	`0.35`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LinesVisible`	visibility of lines	`TRUE`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`Radius`	radius of circles, spheres etc.	`1`
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`

Attribute	Purpose	Default Value
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Top	top center of cones, cylinders, pyramids and prisms	[0, 0, 1]
TopX	base and top center of cones, cylinders, pyramids and prisms	0
TopY	base and top center of cones, cylinders, pyramids and prisms	0
TopZ	base and top center of cones, cylinders, pyramids and prisms	1
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE

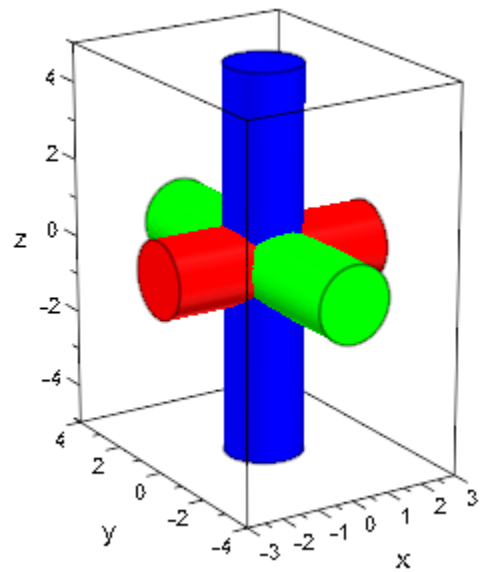
Attribute	Purpose	Default Value
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

We plot three cylinders with symmetry axes given by the coordinate axes:

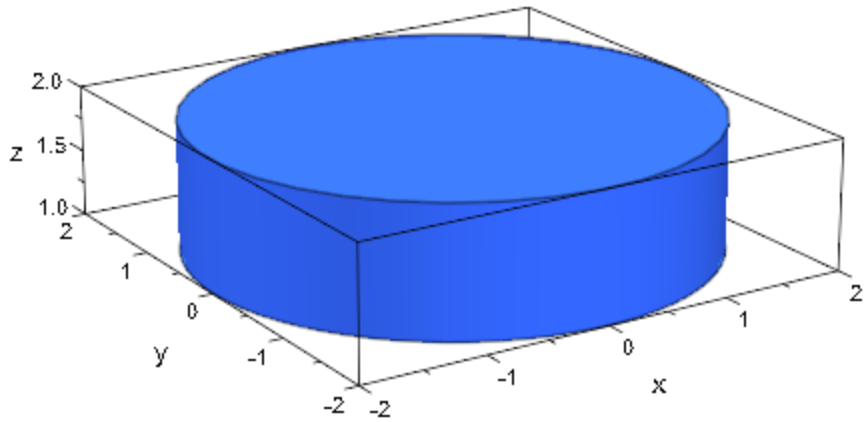
```
plot(plot::Cylinder(1, [-3, 0, 0], [3, 0, 0],
                    Color = RGB::Red),
     plot::Cylinder(1, [0, -4, 0], [0, 4, 0],
                    Color = RGB::Green),
     plot::Cylinder(1, [0, 0, -5], [0, 0, 5],
                    Color = RGB::Blue)):
```



Example 2

All parameters of a cylinder can be animated:

```
plot(plot::Cylinder(a, [0, 0, a], [0, 0, 3 - a],
                    a = 1 .. 2))
```



Parameters

r

The radius of the cylinder: a real numerical value or an arithmetical expression of the animation parameter `a`.

r is equivalent to the attribute `Radius`.

x_1, y_1, z_1

Components of the base center: real numerical values or expressions of the animation parameter `a`.

x_1, y_1, z_1 are equivalent to the attributes `BaseX`, `BaseY`, `BaseZ`.

x_2, y_2, z_2

Components of the top center: real numerical values or expressions of the animation parameter `a`.

x_2, y_2, z_2 are equivalent to the attributes `TopX`, `TopY`, `TopZ`.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Cone` | `plot::Prism` | `plot::Pyramid`

plot::Cylindrical

Surfaces in 3D parameterized in cylindrical coordinates

Syntax

```
plot::Cylindrical([r, ϕ, z], u = u_min .. u_max, v = v_min .. v_max, <a = a_min .. a_max>, options)
```

Description

`plot::Cylindrical` creates surfaces parametrized in cylindrical coordinates.

The surface given by a mapping ("parametrization") $(u, v) \to (r(u, v), \phi(u, v), z(u, v))$ is the set of all image points

$$\left\{ \begin{pmatrix} r(u, v) \\ \phi(u, v) \\ z(u, v) \end{pmatrix} \;\middle|\; u \in [u_{\min}, u_{\max}], v \in [v_{\min}, v_{\max}] \right\}$$

in cylindrical coordinates, which translate to the usual "Cartesian" coordinates as

$$\begin{aligned} x &= r\cos(\phi) \\ y &= r\sin(\phi) \\ z &= z \end{aligned}$$

.

r is referred to as "radius," ϕ as "polar angle," and z as the "height" of a point.

The functions r, ϕ, z are evaluated on a regular equidistant mesh of sample points in the u-v plane. This mesh is determined by the attributes `UMesh`, `VMesh`. By default, the attribute `AdaptiveMesh = 0` is set, i.e., no adaptive refinement of the equidistant mesh is used.

If the standard mesh does not suffice to produce a sufficiently detailed plot, one may either increase the value of `UMesh`, `VMesh` or `USubmesh`, `VSubmesh`,or set `AdaptiveMesh = n` with some (small) positive integer `n`. If necessary, up to 2^n - 1

additional points are placed in each direction of the u-v plane between adjacent points of the initial equidistant mesh. Cf. "Example 2" on page 24-297.

"Coordinate lines" ("parameter lines") are curves on the surface.

The phrase "ULines" refers to the curves ($r(u, v_0)$, $\phi(u, v_0)$, $z(u, v_0)$) with the parameter u running from u_{min} to u_{max}, while v_0 is some fixed value from the interval $[v_{min}, v_{max}]$.

The phrase "VLines" refers to the curves ($r(u_0, v)$, $\phi(u_0, v)$, $z(u_0, v)$) with the parameter v running from v_{min} to v_{max}, while u_0 is some fixed value from the interval $[u_{min}, u_{max}]$.

By default, the parameter curves are visible. They may be switched off by specifying `ULinesVisible = FALSE` and `VLinesVisible = FALSE`, respectively.

The coordinate lines controlled by `ULinesVisible = TRUE/FALSE` and `VLinesVisible = TRUE/FALSE` indicate the equidistant mesh in the u-v plane set via the `UMesh`, `VMesh` attributes. If the mesh is refined by the `USubmesh`, `VSubmesh` attributes, or by the adaptive mechanism controlled by `AdaptiveMesh = n`, no additional parameter lines are drawn.

As far as the numerical approximation of the surface is concerned, the settings

`UMesh =` n_u, `VMesh =` n_v, `USubmesh =` m_u, `VSubmesh =` m_v

and

`UMesh = (`n_u `- 1) (`m_u `+ 1) + 1`, `VMesh = (`n_v `- 1) (`m_v `+ 1) + 1`,

`USubmesh = 0`, `VSubmesh = 0`

are equivalent. However, in the first setting, n_u parameter lines are visible in the u direction, while in the latter setting `(`n_u `- 1) (`m_u `+ 1) + 1` parameter lines are visible. Cf. "Example 2" on page 24-297.

Use `Filled = FALSE` to obtain a wireframe representation of the surface.

If the expressions/functions r and/or z contain singularities, it is recommended (but not strictly necessary) to use the attribute `ViewingBox` to set a suitable viewing box. No such precautions are necessary for ϕ, although singularities in this function may result in poorly rendered surfaces – in many cases setting the attributes `Mesh` and/or `AdaptiveMesh` to higher values will help. Cf. "Example 3" on page 24-301.

Attributes

Attribute	Purpose	Default Value
AdaptiveMesh	adaptive sampling	0
AffectViewingBox	influence of objects on the ViewingBox of a scene	TRUE
Color	the main color	RGB::Red
Filled	filled or transparent areas and surfaces	TRUE
FillColor	color of areas and surfaces	RGB::Red
FillColor2	second color of areas and surfaces for color blends	RGB::CornflowerBlue
FillColorType	surface filling types	Dichromatic
FillColorFunction	functional area/surface coloring	
FillColorDirection	the direction of color transitions on surfaces	[0, 0, 1]
FillColorDirectionX	x-component of the direction of color transitions on surfaces	0
FillColorDirectionY	y-component of the direction of color transitions on surfaces	0
FillColorDirectionZ	z-component of the direction of color transitions on surfaces	1
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	TRUE

Attribute	Purpose	Default Value
LineColor	color of lines	RGB::Black.[0.25]
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	0
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Mesh	number of sample points	[25, 25]
MeshVisible	visibility of irregular mesh lines in 3D	FALSE
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles

Attribute	Purpose	Default Value
PointsVisible	visibility of mesh points	FALSE
Submesh	density of submesh (additional sample points)	[0, 0]
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
ULinesVisible	visibility of parameter lines (u lines)	TRUE
UMax	final value of parameter “u”	
UMesh	number of sample points for parameter “u”	25
UMin	initial value of parameter “u”	
UName	name of parameter “u”	
URange	range of parameter “u”	
USubmesh	density of additional sample points for parameter “u”	0
VLinesVisible	visibility of parameter lines (v lines)	TRUE

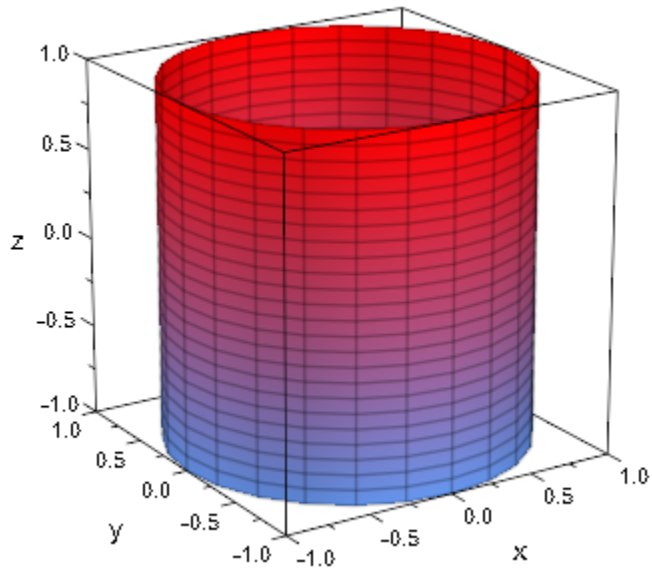
Attribute	Purpose	Default Value
VMax	final value of parameter “v”	
VMesh	number of sample points for parameter “v”	25
VMin	initial value of parameter “v”	
VName	name of parameter “v”	
VRange	range of parameter “v”	
VSubmesh	density of additional sample points for parameter “v”	0
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XContours	contour lines at constant x values	[]
XFunction	function for x values	
YContours	contour lines at constant y values	[]
YFunction	function for y values	
ZContours	contour lines at constant z values	[]
ZFunction	function for z values	

Examples

Example 1

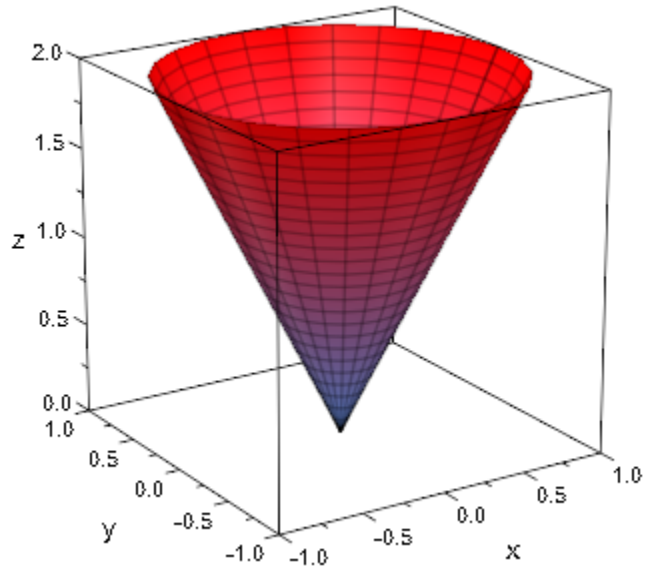
Using a constant radius for `plot::Cylindrical`, with the other two functions straight from the surface parameters, results in a right cylinder. This explains the name "cylindrical coordinates":

```
plot(plot::Cylindrical([1, phi, z], phi = 0..2*PI, z = -1..1))
```

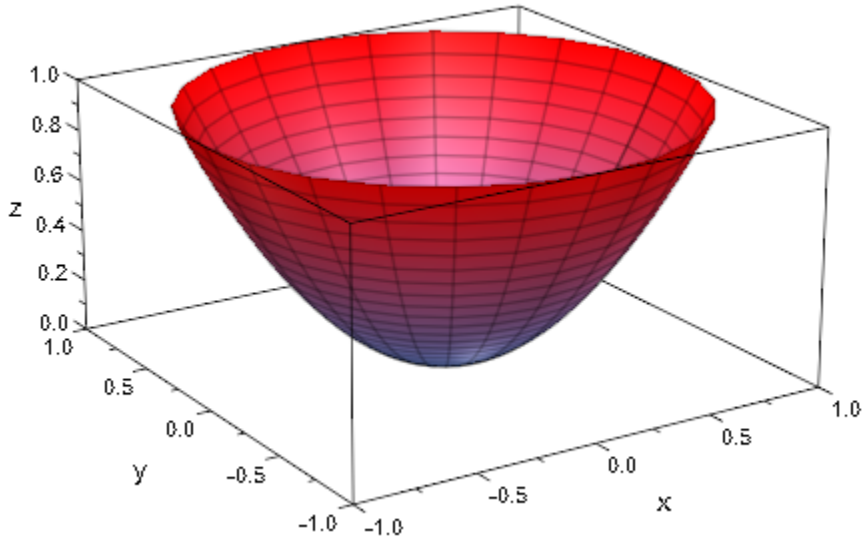


Other straightforward examples include cones and paraboloids of revolution:

```
plot(plot::Cylindrical([r, phi, 2*r], r = 0..1, phi = 0..2*PI))
```



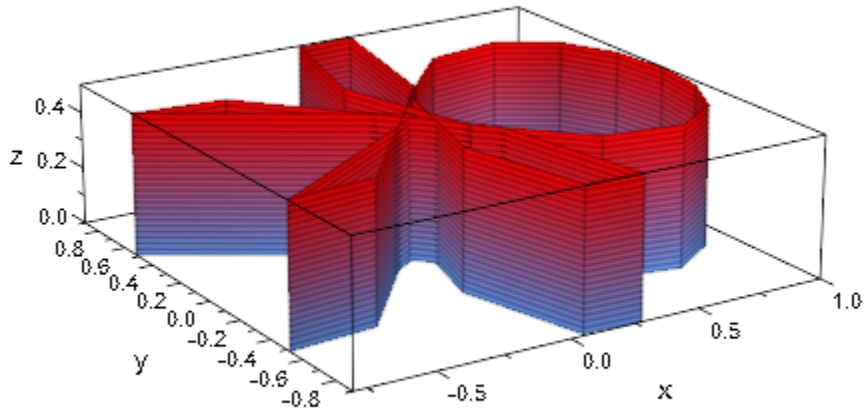

```
plot(plot::Cylindrical([r, phi, r^2], r = 0..1, phi = 0..2*PI))
```



Example 2

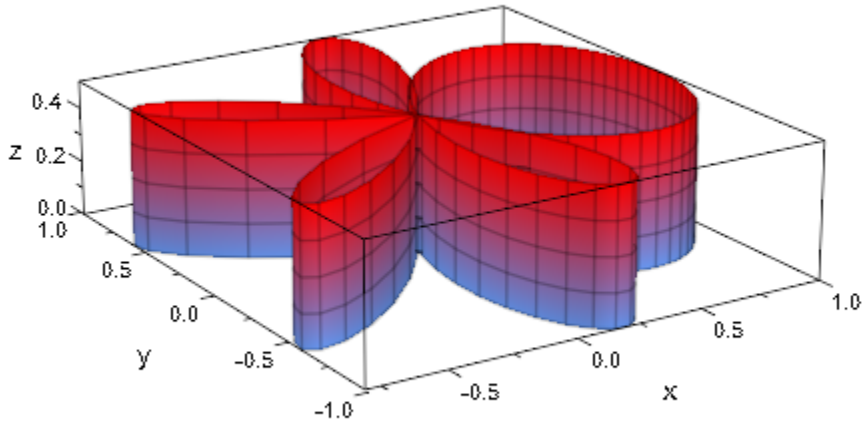
Cylindrical surfaces are drawn from evaluations on an equidistant mesh of points. In some cases, the default mesh density is insufficient or otherwise inappropriate:

```
plot(plot::Cylindrical([cos(phi^2), phi, z],
                       phi=-2.8..2.8, z=0..1/2))
```



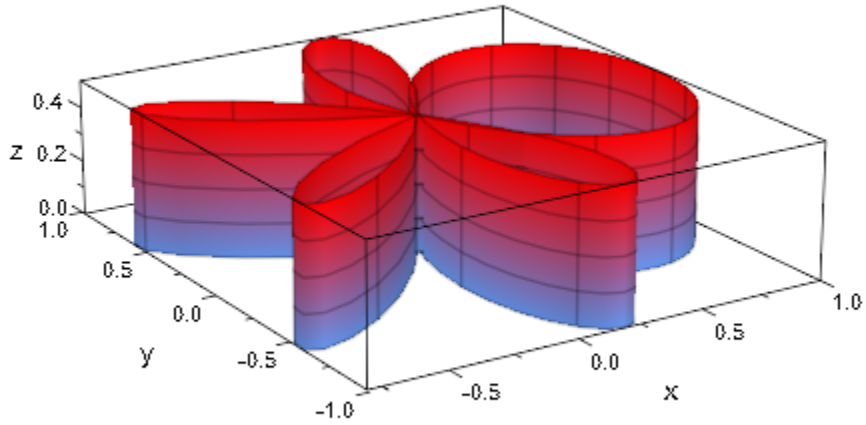
One possible change to this plot command is to explicitly set the mesh with the attribute `Mesh`. Note that this setting influences the density of parameter lines:

```
plot(plot::Cylindrical([cos(phi^2), phi, z],
                       phi=-2.8..2.8, z=0..1/2,
                       Mesh = [100, 5]))
```



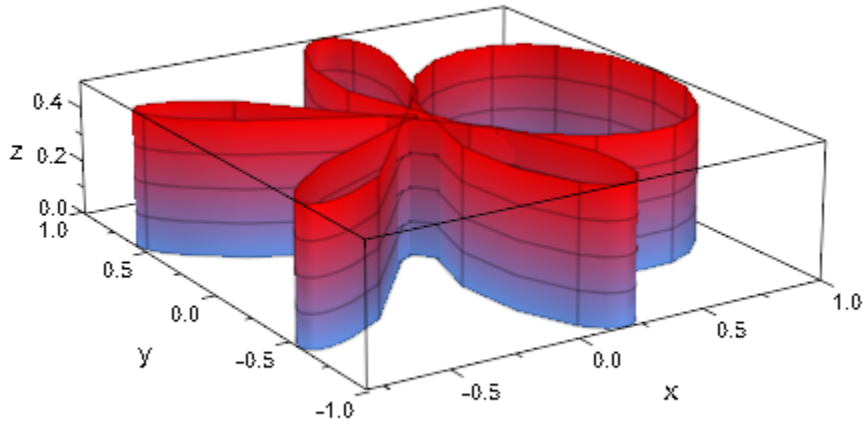
To increase the mesh density without introducing additional parameter lines, you can use submesh settings:

```
plot(plot::Cylindrical([cos(phi^2), phi, z],
                       phi=-2.8..2.8, z=0..1/2,
                       VMesh = 5, USubmesh = 3))
```



Finally, we can also ask `plot::Cylindrical` to refine the mesh only in areas of higher curvature. In the following example, we allow for $2^3 = 8$ additional points between each two neighboring points of the initial mesh:

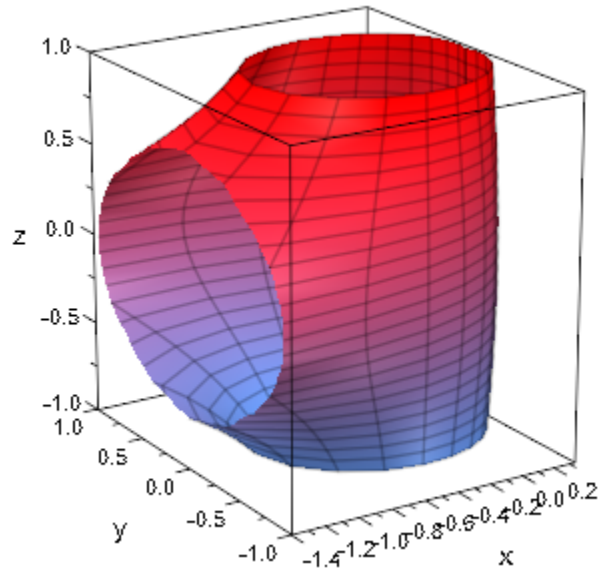
```
plot(plot::Cylindrical([cos(phi^2), phi, z],
                       phi=-2.8..2.8, z=0..1/2,
                       VMesh = 5, AdaptiveMesh = 3))
```



Example 3

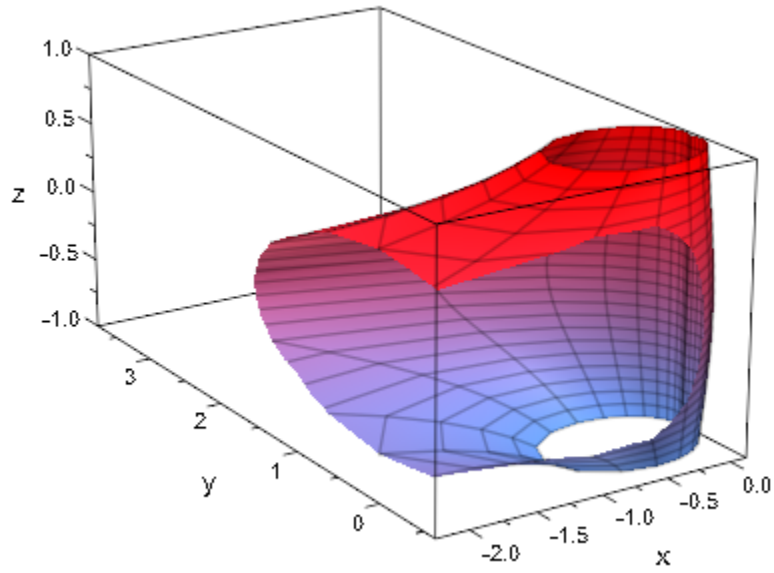
If the radius- or the *z*-function/expression contains singularities, `plot::Cylindrical` employs heuristic clipping to select a range to display:

```
plot(plot::Cylindrical([1/sqrt((phi - PI)^2 + z^2), phi, z],
                       phi = 0..2*PI, z = -1..1))
```



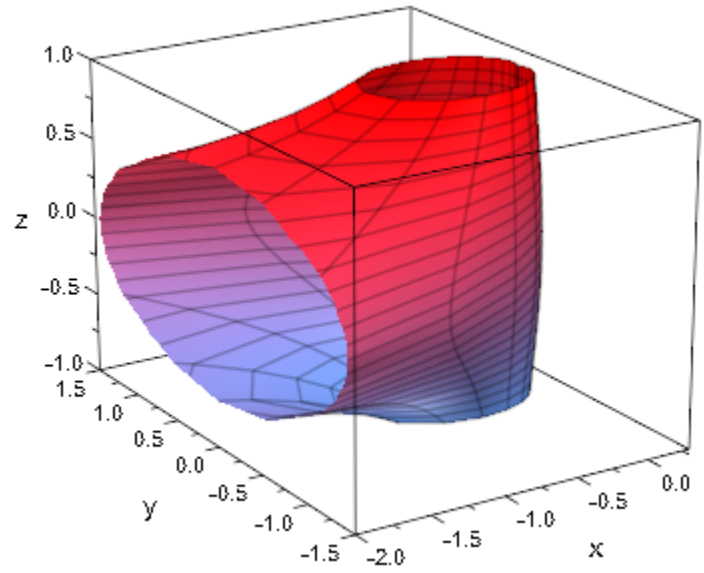
While these heuristics work well in many cases, there are also examples where they do not select a useful box:

```
plot(plot::Cylindrical([1/((phi - PI)^2 + z^2), phi, z],
                       phi = 0.. 2*PI, z = -1..1))
```



In these cases, the user should set the range to display explicitly:

```
plot(plot::Cylindrical([1/((phi - PI)^2+z^2), phi, z],
                       phi = 0..2*PI, z = -1..1),
     ViewingBox = [-2..0.3, -1.5..1.5, -1..1])
```



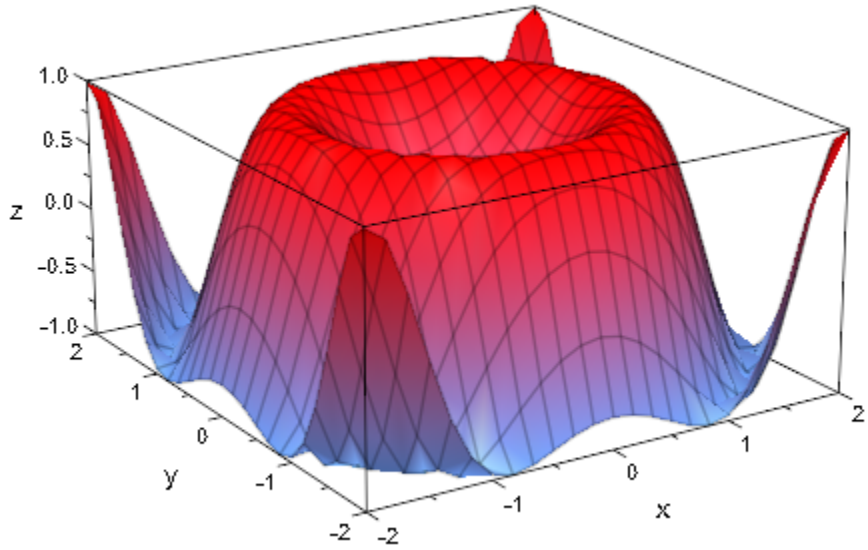
Example 4

Since the transformation from cylindrical to orthogonal coordinates is reversible (up to reducing the angle to the range [0, 2 π], it is possible to plot any surface with `plot::Cylindrical` (although this is probably more a curiosity than really useful):

```
trans := linalg::ogCoordTab[Cylindrical, InverseTransformation]:
cyl   := trans(x, y, sin(x^2+y^2))
```

$$\left[\sqrt{x^2+y^2},\ \arccos\left(\frac{x}{\sqrt{x^2+y^2}}\right) + \mathrm{sign}(y)\ (\mathrm{sign}(y)-1)\ \left(\pi - \arccos\left(\frac{x}{\sqrt{x^2+y^2}}\right)\right),\ \sin(x^2+y^2)\right]$$

```
plot(plot::Cylindrical(cyl, x = -2..2, y = -2..2))
```



Parameters

`r, ϕ, z`

The coordinate functions: arithmetical expressions or `piecewise` objects depending on the surface parameters u, v and the animation parameter a. Alternatively, procedures that accept 2 input parameters u, v or 3 input parameters u, v, a and return a real numerical value when the input parameters are numerical.

r, ϕ, z are equivalent to the attributes `XFunction`, `YFunction`, `ZFunction`.

`u`

The first surface parameter: an identifier or an indexed identifier.

u is equivalent to the attributes `UName`, `UMin`, `UMax`.

u_{min} .. u_{max}

The plot range for the parameter u: u_{min}, u_{max} must be numerical real values or expressions of the animation parameter a.

u_{min} .. u_{max} is equivalent to the attributes `URange`, `UMin`, `UMax`.

v

The second surface parameter: an identifier or an indexed identifier.

v is equivalent to the attribute `VName`.

v_{min} .. v_{max}

The plot range for the parameter *v*: v_{min}, v_{max} must be numerical real values or expressions of the animation parameter *a*.

v_{min} .. v_{max} is equivalent to the attributes `VRange`, `VMin`, `VMax`.

a

Animation parameter, specified as a = a_{min}..a_{max}, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions

`linalg::ogCoordTab` | `plot` | `plot::copy`

MuPAD Graphical Primitives

`plot::Polar` | `plot::Spherical` | `plot::Tube`

plot::Density

Density plot

Syntax

```
plot::Density(f, x = x_min .. x_max, y = y_min .. y_max, <a = a_min .. a_max>, options)
plot::Density(A, <x = x_min .. x_max, y = y_min .. y_max>, <a = a_min .. a_max>, options)
plot::Density(L, <x = x_min .. x_max, y = y_min .. y_max>, <a = a_min .. a_max>, options)
```

Description

`plot::Density(f(x, y), x = `x_{min}`..`x_{max}` , y = `y_{min}`..`y_{max}` )` generates a regular 2D mesh of rectangles extending from the lower left corner (x_{min}, y_{min}) to the upper right corner (x_{max}, y_{max}). The rectangle with midpoint (x, y) is colored according to a color scheme based on the "density" value $f(x, y)$.

`plot::Density` serves for the visualization of 3D data (x, y, $f(x, y)$) by a 2D plot. Roughly speaking, it corresponds to a colored 3D function graph of the density function $f(x, y)$ viewed from above. However, in contrast to the 3D function graph, `plot::Density` does not use smooth interpolation ("shading") of the color between adjacent rectangles.

If the density data are provided by an array or matrix `A` or by a list `L`, the number of rectangles in the density plot is given automatically by the format of `A` or `L`, respectively.

If the density data are given by an expression or function `f`, the attribute `Mesh = [m, n]` serves for advising `plot::Density` to create a grid of $m \times n$ rectangles. Alternatively, one may set `XMesh = m`, `YMesh = n`.

With the default `FillColorType = Dichromatic`, the rectangle with density value $f(x, y)$ at the midpoint (x, y) is colored with the color

$$\frac{f(x, y) - f_{\min}}{f_{\max} - f_{\min}} \times \text{fillcolor} + \frac{f_{\max} - f(x, y)}{f_{\max} - f_{\min}} \times \text{fillcolor2},$$

where $\frac{f_{min}}{f_{max}}$ are the minimal/maximal density values in the graphics and `fillcolor`, `fillcolor2` are the `RGB` values of the attributes `FillColor` and `FillColor2`, respectively. Thus, `fillcolor` indicates high density values whereas `fillcolor2` indicates low density values.

If $f_{min} = f_{max}$, a flat coloring with `fillcolor` is used.

With `FillColorType` = `Monochrome`, the rectangle with density value $f(x, y)$ at the midpoint (x, y) is colored with the color

$$\frac{f(x, y) - f_{min}}{f_{max} - f_{min}} \times \text{fillcolor}$$

.

The user may specify a fill color function via `FillColorFunction = mycolorfunction` to override the density coloring described above. The procedure `mycolorfunction` will be called with the arguments

```
mycolorfunction(x, y, f(x, y, a ) a ),
```

where (x, y) are the midpoints of the rectangles and a is the animation parameter. The color function must return an `RGB` or `RGBa` color value.

When density values are specified by an array or a matrix A, the low indices correspond to the lower left corner of the graphics. The high indices correspond to the upper right corner.

Arrays/matrices do not need to be indexed from 1. E.g.,

```
A = array( `i_{min}` .. `i_{max}` , `j_{min}` .. `j_{max}` ,
[..density values..])
```

yields a graphical array with

$\texttt{XMesh} = j_{max} - j_{min} + 1, \texttt{YMesh} = i_{max} - i_{min} + 1.$

If no plot range `` `x_{min}` .. `x_{max}` ``, `` `y_{min}` .. `y_{max}` `` is specified,

$x_{min} = j_{min} - 1, x_{max} = j_{max}, y_{min} = i_{min} - 1, y_{max} = i_{max}$

is used.

When density values are specified by a list of lists L, the first entries in the list correspond to the lower left corner of the graphics. The last entries correspond to the upper right corner.

If no plot range `` `x_{min}` .. `x_{max}` ``, `` `y_{min}` .. `y_{max}` `` is specified,

$x_{min} = 0, x_{max} = m, y_{min} = 0, y_{max} = n$

is used, where n is the length of L and m is the (common) length of the sublists in L. All sublists ("rows") must have the same length.

Animations are triggered by specifying a range `a = `a_{min}` .. `a_{max}`` for a parameter `a` that is different from the variables `x`, `y`. Thus, in animations, both the ranges `x = `x_{min}` .. `x_{max}``, `y = `y_{min}` .. `y_{max}`` as well as the animation range `a = `a_{min}` .. `a_{max}`` must be specified.

The related plot routine `plot::Raster` provides a similar functionality. However, `plot::Raster` does not use an automatic color scheme based on density values. The user must provide `RGB` or `RGBa` values instead.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`FALSE`
`Color`	the main color	`RGB::Red`
`DensityData`	density values for a density plot	
`DensityFunction`	density function for a density plot	
`FillColor`	color of areas and surfaces	`RGB::Red`
`FillColor2`	second color of areas and surfaces for color blends	`RGB::CornflowerBlue`
`FillColorType`	surface filling types	`Dichromatic`
`FillColorFunction`	functional area/surface coloring	

Attribute	Purpose	Default Value
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`LineColor`	color of lines	`RGB::Blue`
`LineWidth`	width of lines	`0.35`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LinesVisible`	visibility of lines	`FALSE`
`Mesh`	number of sample points	[25, 25]
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`
`Title`	object title	
`TitleFont`	font of object titles	`[" sans-serif ", 11]`
`TitlePosition`	position of object titles	

Attribute	Purpose	Default Value
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XMax	final value of parameter “x”	
XMesh	number of sample points for parameter “x”	25
XMin	initial value of parameter “x”	
XName	name of parameter “x”	
XRange	range of parameter “x”	
YMax	final value of parameter “y”	
YMesh	number of sample points for parameter “y”	25
YMin	initial value of parameter “y”	
YName	name of parameter “y”	
YRange	range of parameter “y”	

Examples

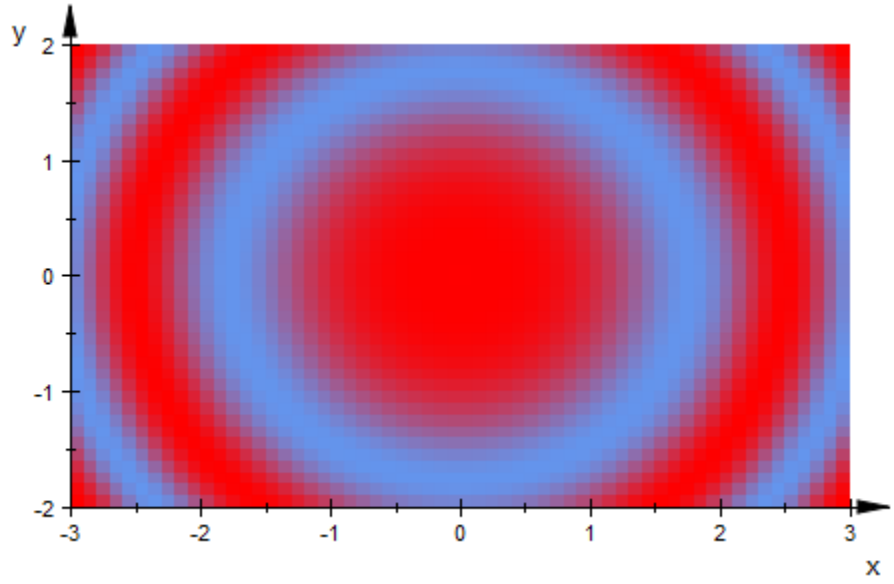
Example 1

We generate a density plot:

```
p := plot::Density(cos(x^2 + y^2), x = -3..3, y = -2..2,
                   Mesh = [60, 40]):
```

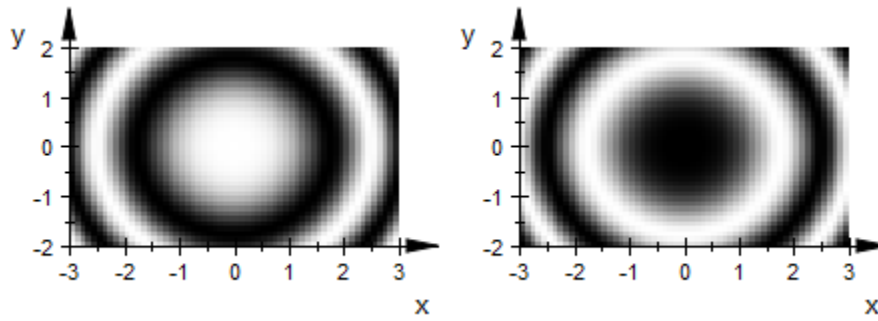
The plot object is rendered:

```
plot(p, Axes = Frame):
```



This turns into a black and white graphics when suitable colors are specified:

```
plot(plot::Scene2d(p, FillColor = RGB::White,
                      FillColor2 = RGB::Black),
     plot::Scene2d(p, FillColor = RGB::Black,
                      FillColor2 = RGB::White),
     Width = 120*unit::mm, Height = 45*unit::mm,
     Layout = Horizontal, Axes = Frame):
```

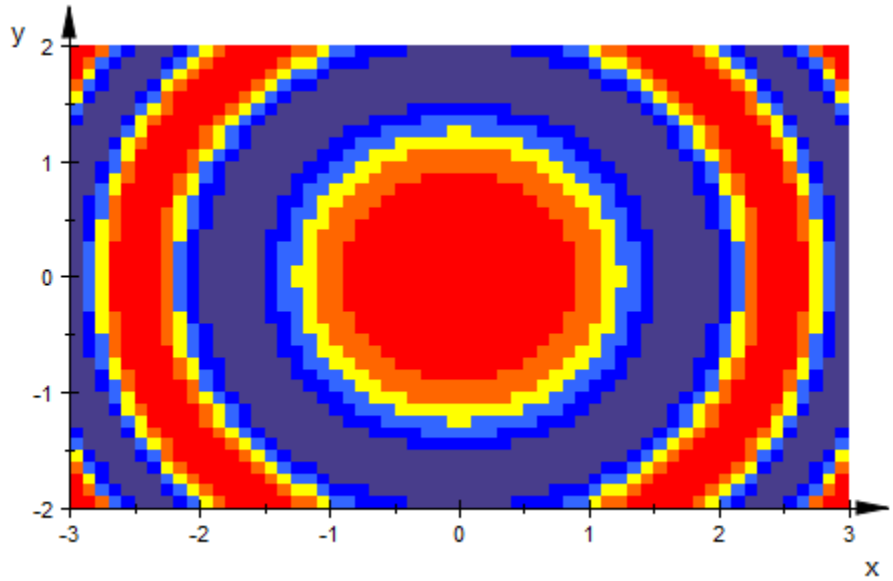



```
delete p:
```

Example 2

We demonstrate the use of a user-defined color function:

```
mycolor := proc(x, y, f)
begin
   if   f >=  2/3 then RGB::Red
   elif f >=  1/3 then RGB::Orange;
   elif f >=   0  then RGB::Yellow;
   elif f >= -1/3 then RGB::BlueLight;
   elif f >= -2/3 then RGB::Blue;
   else RGB::SlateBlueDark;
   end_if;
end_proc:
plot(plot::Density(cos(x^2 + y^2), x = -3..3, y = -2..2,
                   Mesh = [60, 40],
                   FillColorFunction = mycolor),
     Axes = Frame):
```




```
delete mycolor:
```

Example 3

In this example, we demonstrate how `plot::Density` can be used to plot gray data from an external source. Assume, there is an external PortableGrayMap text file `Norton.pgm` containing data such as

```
P2
240 180
255
249 237 228 231 245 218 229 195 ...
```

The first line contains the "magic value" `P2` indicating that this is a PGM text file. The second line contains the pixel width and pixel height of the picture. The number `255` in the third line is the scale of the following gray values.

The remaining data consist of integers between 0 (black) and 255 (white), each representing the gray value of a pixel (row by row).

We import the text data via `import::readdata`:

```
graydata := import::readdata("Norton.pgm", NonNested):
```

This is a long list of all data items in the file. We extract the 4 items in the first three lines:

```
[magicvalue, xmesh, ymesh, maxgray] := graydata[1..4]
```

```
[P2, 240, 180, 255]
```

We delete the header from the pixel data. (If there are comments in the PGM file, they must be deleted, too).

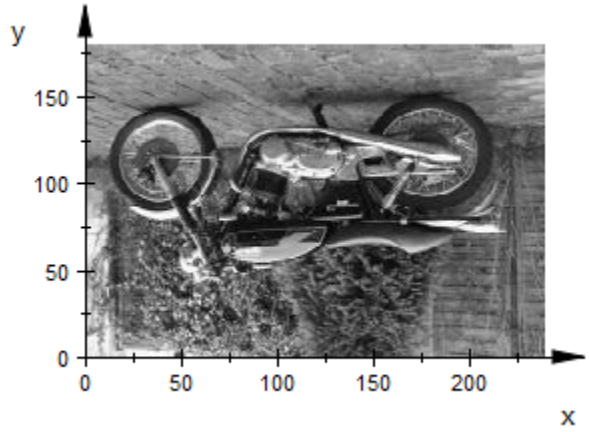
```
for i from 1 to 4 do
  delete graydata[1];
end_for:
```

We transform the plain data list to a nested list containing the gray data of the rows as sublists. (The call to `level` is not really necessary, but it speeds up the conversion considerably on the interactive level.)

```
L := level([graydata[(i - 1)*xmesh + 1 .. i*xmesh] $ i=1..ymesh], 1):
```

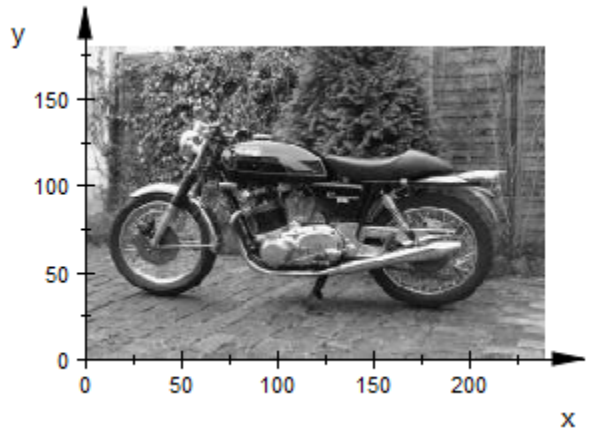
This list can be passed to `plot::Density`:

```
plot(plot::Density(L, FillColor = RGB::White,
                   FillColor2 = RGB::Black),
     Width = 80*unit::mm, Height = 60*unit::mm):
```



The image is upside down, because the PGM files stores the pixel data row by row in the usual reading order starting with the upper left corner of the image. The MuPAD routine `plot::Density`, however, follows the mathematical orientation of the coordinate axes, i.e., the first pixel value is interpreted as the lower left corner of the image. We have to re-order the rows in the `graydata` list via `revert`:

```
plot(plot::Density(revert(L), FillColor = RGB::White,
                   FillColor2= RGB::Black),
     Width = 80*unit::mm, Height = 60*unit::mm):
```



The routines `import::readbitmap` and `plot::Raster` provide an alternative way to import and display the bitmap image. See the help page of plot::Raster for examples. This, however, takes more memory, because the bitmap data are imported as `RGB` color values, whereas only density values (gray data) are needed for `plot::Density`.

```
delete graydata, magicvalue, xmesh, ymesh, maxgray, i, L:
```

Example 4

The Mandelbrot set is one of the best-known fractals. It arises when considering the iteration $z_{n+1} = z_n^2 + c$, $z_0 = 0$ in the complex plane. For sufficiently large values $|c|$ of the complex parameter c, the sequence z_n diverges to infinity; it converges for sufficiently small values of $|c|$. The boundary of the region of those c values that lead to divergence of z_n is of particular interest: this border is highly complicated and of a fractal nature.

In particular, it is known that the series z_n diverges to infinity, whenever one of the iterates satisfies $|z_n| > 2$. This fact is used by the following procedure `f` as stopping criterion. The return value provides information, how many iterates $z_0, \ldots, z_n$ it takes to escape from the region $|z| \le 2$ of (potential) convergence. These data are to be used to color the complex c plane (i.e., the (x,y) plane) by a density plot:

```
f := proc(x, y)
local c, z, n;
begin
   c := x + I*y:
   z := 0.0:
   for n from 0 to 100 do
       z := z^2 + c:
       if abs(z) > 2 then
          break;
       end_if;
   end_for:
   if n < 70 then
        n mod 5;
   else n - 70;
   end_if;
end_proc:
```

Depending on your computer, the following computations may take some time. On a very fast machine, you can increase the following values of `xmesh`, `ymesh`. This will use up more computing time but will lead to better graphical results:

```
xmesh :=  100: ymesh :=  100:
```

The following region in the x-y plane is to be considered:

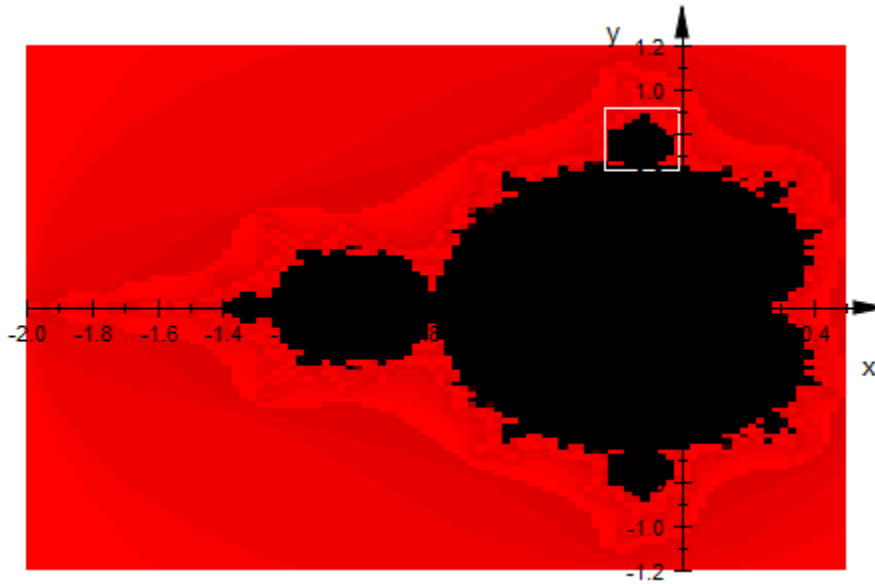
```
xmin[1] := -2.0: xmax[1] := 0.5:
ymin[1] := -1.2: ymax[1] := 1.2:
```

The region $xmin_1 \le x \le xmax_1$, $ymin_1 \le y \le ymax_1$ is divided into $xmesh \times ymesh$ rectangles. Each rectangle is colored by a density plot according to the "escape times" computed by the procedure `f`. This procedure can be passed directly to `plot::Density`:

```
p1 := plot::Density(f, x = xmin[1].. xmax[1],
                    y = ymin[1] .. ymax[1],
                    Mesh = [xmesh, ymesh],
                    FillColor = RGB::Black,
                    FillColor2 = RGB::Red):
```

In addition, a rectangle is produced that indicates a region that is to be magnified in the following:

```
xmin[2] := -0.24: xmax[2] := -0.01:
ymin[2] :=  0.63: ymax[2] :=  0.92:
r1 := plot::Rectangle(xmin[2] .. xmax[2], ymin[2] .. ymax[2],
                      LineColor = RGB::White):

plot(p1, r1):
```

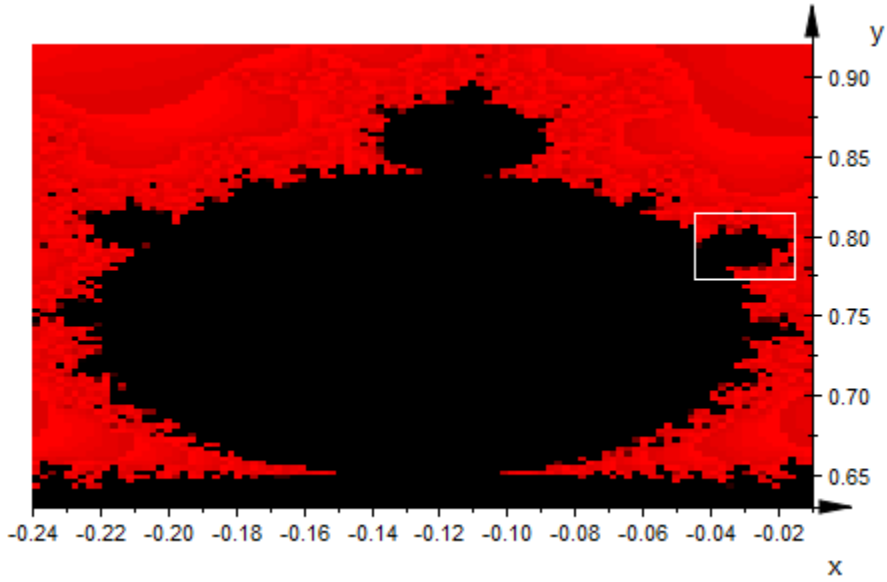


The density values of the blow-up are not computed directly by `plot::Density`. They are computed separately and stored in an array *A*:

```
dx := (xmax[2] - xmin[2])/xmesh:
dy := (ymax[2] - ymin[2])/ymesh:
A := array(1..ymesh, 1..xmesh,
           [[f(xmin[2]+ (j - 1/2)*dx, ymin[2] + (i - 1/2)*dy)
             $ j = 1..xmesh] $ i = 1..ymesh]):
p2 := plot::Density(A, x = xmin[2] .. xmax[2], y = ymin[2] .. ymax[2],
                    FillColor = RGB::Black, FillColor2 = RGB::Red):
```

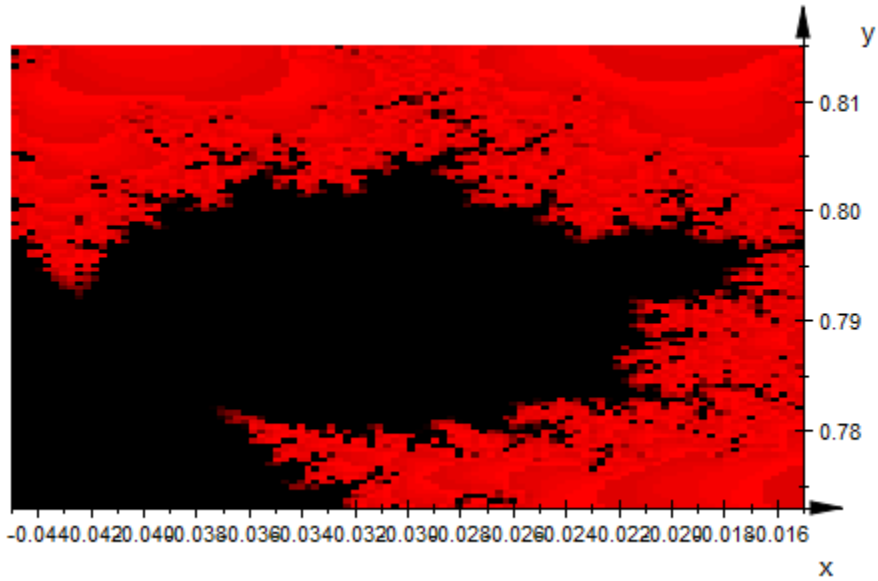
In addition, a further rectangle is produced to indicate a region of interest to be blown up lateron:

```
xmin[3] := -0.045: xmax[3] := -0.015:
ymin[3] :=  0.773: ymax[3] :=  0.815:
r2 := plot::Rectangle(xmin[3] .. xmax[3], ymin[3] .. ymax[3],
                      LineColor = RGB::White):

plot(p2, r2):
```



The density values of the next blow-up are again computed separately and stored in a nested list *L*:

```
dx := (xmax[3] - xmin[3])/xmesh:
dy := (ymax[3] - ymin[3])/ymesh:
L := [[f(xmin[3] + (j - 1/2)*dx, ymin[3] + (i - 1/2)*dy)
      $ j= 1..xmesh] $ i = 1..ymesh]:
p3 := plot::Density(L, x = xmin[3] .. xmax[3], y = ymin[3] .. ymax[3],
                    FillColor = RGB::Black, FillColor2 = RGB::Red):

plot(p3):
```



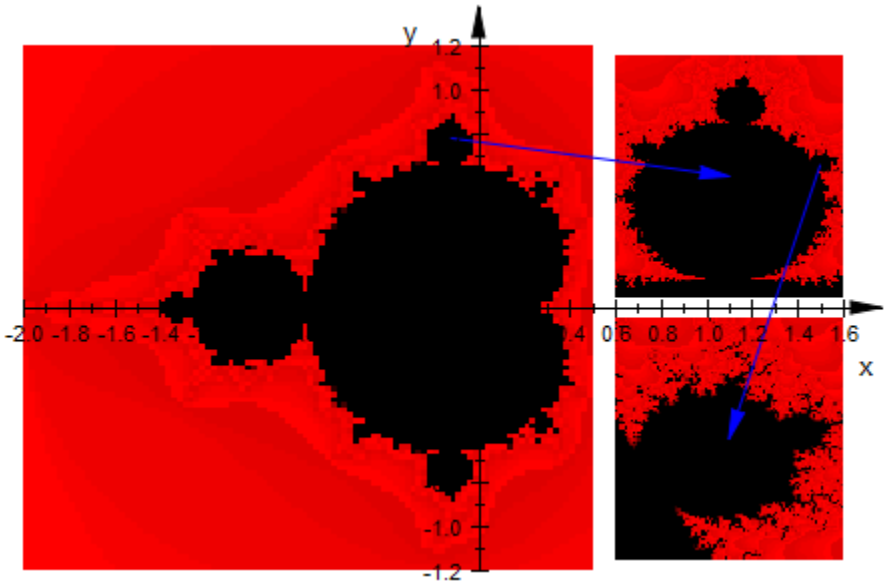
The density objects are to be placed in a single graphics. It consists of the Mandelbrot set `p1` as computed above and of modifications of the density plots `p2` and `p3`. Redefining the attributes `XRange`, `YRange`, we move `p2`, `p3` to places in the *x*-*y* plane where they are not overlapped by `p1`. Note that this does not change the graphical content of `p2`, `p3`, because it is given by the data *A* and *L*, respectively, which remain unchanged. (If the ranges were changed in `p1`, another `plot` call of `p1` would call the procedure `f` at different points of the plane resulting in a different graphics.)

```
p2::XRange :=  0.60 .. 1.60: p2::YRange :=  0.05 ..  1.15:
p3::XRange :=  0.60 .. 1.60: p3::YRange := -1.15 .. -0.05:
```

The Mandelbrot set and the two blow-ups are placed in one scene. In addition, some arrows are added to indicate the origin of the blow-ups. Note that it is quite important here that the arrows are passed to the `plot` command after the density plots. Otherwise, they would be hidden by the density plots: graphical objects are painted in the ordering in which they are passed to `plot`:

```
plot(p1, p2, p3,
     plot::Arrow2d([(xmin[2] + xmax[2])/2,
                    (ymin[2] + ymax[2])/2],
                   [(p2::XMin + p2::XMax)/2,
```

```
                 (p2::YMin + p2::YMax)/2],
                LineColor = RGB::Blue),
    plot::Arrow2d([1.50, 0.65],
                [(p3::XMin + p3::XMax)/2,
                 (p3::YMin + p3::YMax)/2],
                LineColor = RGB::Blue)
):
```




```
delete f, xmesh, ymesh, xmin, xmax, ymin, ymax,
       dx, dy, p1, p2, p3, r1, r2, A, L:
```

Parameters

f

The density values: an arithmetical expression in 2 variables x, y and the animation parameter a. Alternatively, a procedure that accepts 2 input parameters x, y or 3 input parameters x, y, a and returns a real density value.

f is equivalent to the attribute `DensityFunction`.

x

Name of the horizontal variable: an identifier or an indexed identifier.

x is equivalent to the attribute `XName`.

x_{min} .. x_{max}

The range of the horizontal variable: x_{min}, x_{max} must be numerical real value or expressions of the animation parameter *a*.

x_{min} .. x_{max} is equivalent to the attributes `XRange`, `XMin`, `XMax`.

y

Name of the vertical variable: an identifier or an indexed identifier.

y is equivalent to the attribute `YName`.

y_{min} .. y_{max}

The range of the vertical variable: y_{min}, y_{max} must be numerical real value or expressions of the animation parameter *a*.

y_{min} .. y_{max} is equivalent to the attributes `YRange`, `YMin`, `YMax`.

A

An array of domain type `DOM_ARRAY` or a matrix of category `Cat::Matrix` (e.g., of type `matrix` or `densematrix`) providing numerical density values or expressions of the animation parameter *a*. Rows/columns of the array, respectively matrix, correspond to rows/columns of the graphical array.

A is equivalent to the attribute `DensityData`.

L

A list of lists of numerical density values or expressions of the animation parameter *a*. Each sublist of *L* represents a row of the graphical array. The number of sublists in *L* yields the value of the attribute `XMesh`. The (common) length of the sublists yields the value of the attribute `YMesh`.

L is equivalent to the attribute `DensityData`.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`import::readbitmap` | `plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Inequality` | `plot::Raster`

plot::Ellipse2d

Ellipses in 2D

Syntax

```
plot::Ellipse2d(r1, r2, <[cx, cy]>, <a = amin .. amax>, options)
```

Description

`plot::Ellipse2d( r1, r2, [cx, cy] )` creates a 2D ellipse with center point `[cx, cy]` and semi-axes of lengths `r1` and `r2` for the horizontal and the vertical axis, respectively.

The symmetry axes of the ellipse are parallel to the coordinate axes. Use `plot::Rotate2d` to create ellipses of different orientation.

If no center point is specified, an ellipse with center [0, 0] is created.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`
`Center`	center of objects, rotation center	[0, 0]
`CenterX`	center of objects, rotation center, x-component	0
`CenterY`	center of objects, rotation center, y-component	0
`Color`	the main color	`RGB::Blue`
`Filled`	filled or transparent areas and surfaces	`FALSE`

Attribute	Purpose	Default Value
FillColor	color of areas and surfaces	RGB::Red
FillPattern	type of area filling	DiagonalLines
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Blue
LineWidth	width of lines	0.35
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
LineColorDirection	the direction of color transitions on lines	[0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	1
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	

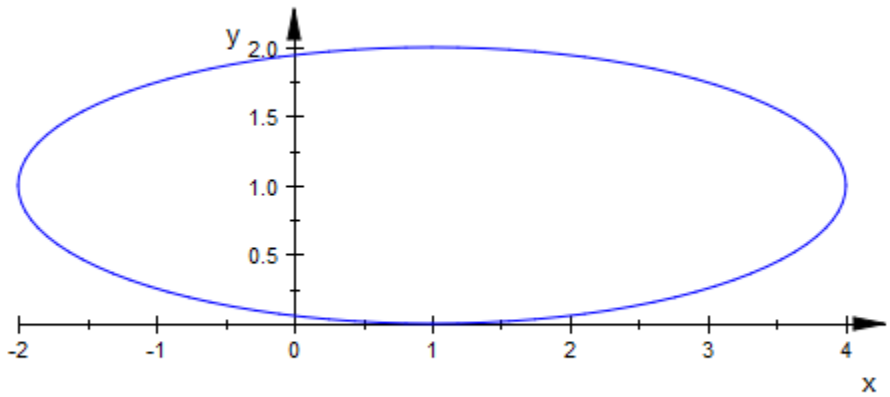
Attribute	Purpose	Default Value
SemiAxes	semi axes of ellipses and ellipsods	[2, 1]
SemiAxisX	first semi axis of ellipses and ellipsods	2
SemiAxisY	second semi axis of ellipses and ellipsods	1
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE

Examples

Example 1

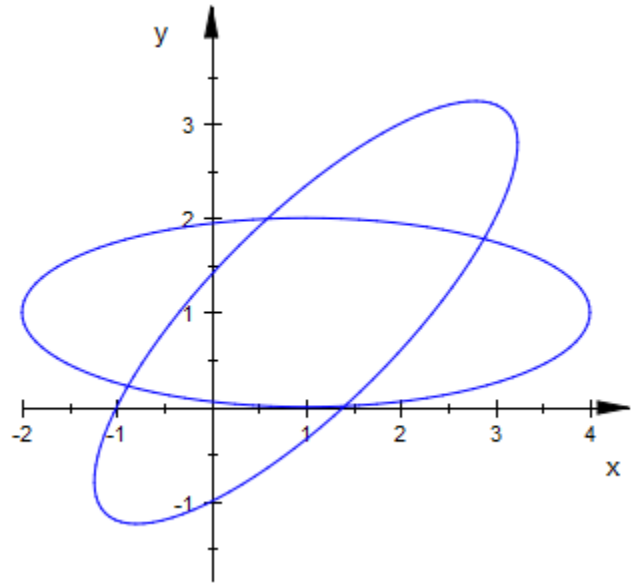
We create a plot of an ellipse with center point (1, 1) and semi-axes of lengths 3 and 1:

```
ellipse := plot::Ellipse2d(3, 1, [1, 1]):
plot(ellipse)
```



We apply a rotation:

```
plot(ellipse, plot::Rotate2d(PI/4, [1, 1], ellipse))
```

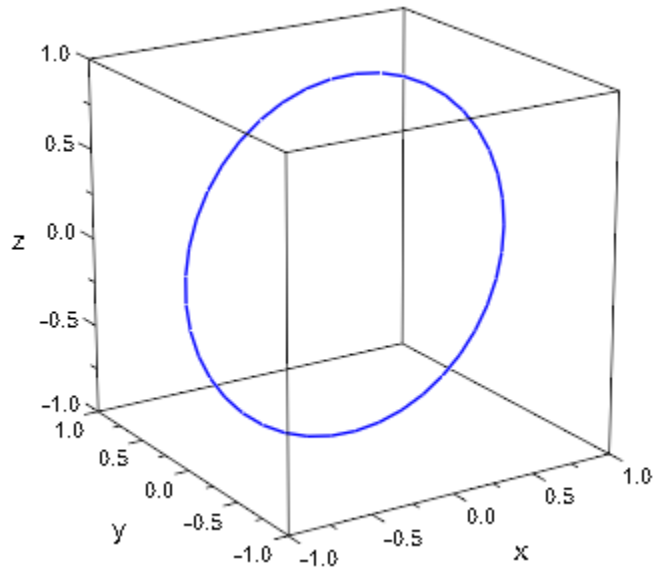



```
delete ellipse:
```

Example 2

We plot an animated 3D ellipse:

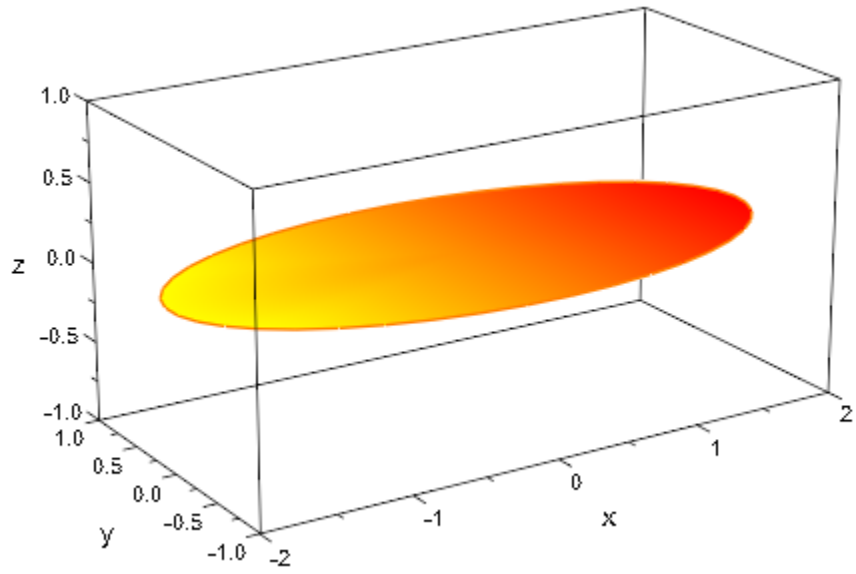
```
plot(plot::Ellipse3d(1, 1, [0,0,0], [0,a,1-a], a = 0..1))
```



Example 3

We plot a colored 3D ellipse:

```
plot(plot::Ellipse3d(2, 1, [0,0,0], Filled,
                     LineColor=RGB::Yellow, LineColor2=RGB::Red,
                     LineColorType = Dichromatic,
                     FillColorDirection=[+1,0,0],
                     FillColor=RGB::Yellow, FillColor2=RGB::Red,
                     FillColorType = Dichromatic,
                     FillColorDirection=[-1,0,0]
))
```



Parameters

r_1, r_2

The semi-axes of an ellipse. They must be real numerical values or arithmetical expressions of the animation parameter `a`.

r_1, r_2 are equivalent to the attributes `SemiAxisX`, `SemiAxisY`.

c_x, c_y

The center. The coordinates c_x, c_y must be real numerical values or arithmetical expressions of the animation parameter `a`. If no center is specified, the ellipse is centered at the origin.

c_x, c_y are equivalent to the attribute `Center`.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Arc2d` | `plot::Arc3d` | `plot::Circle2d` | `plot::Ellipse3d`

plot::Ellipse3d

Ellipses in 3D

Syntax

```
plot::Ellipse3d(r1, r2, <[cx, cy, cz], <[cx, cy, nz]>>, <a = amin .. amax>, options)
```

Description

`plot::Ellipse3d( r1, r2, [cx, cy, cz] , [nx, ny, nz] )` creates a 3D ellipse with center point `[cx, cy, cz]` and semi-axes of lengths `r1` and `r2` in the plane with the normal vector `[nx, ny, nz]`.

The symmetry axes of the ellipse are parallel to the coordinate axes. Use `plot::Rotate2d` to create ellipses of different orientation.

If no center point is specified, an ellipse with center [0, 0, 0] is created.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Center`	center of objects, rotation center	[0, 0, 0]
`CenterX`	center of objects, rotation center, x-component	0
`CenterY`	center of objects, rotation center, y-component	0
`CenterZ`	center of objects, rotation center, z-component	0
`Filled`	filled or transparent areas and surfaces	`FALSE`

Attribute	Purpose	Default Value
FillColor	color of areas and surfaces	RGB::LightBlue
FillColor2	second color of areas and surfaces for color blends	RGB::CornflowerBlue
FillColorType	surface filling types	Flat
FillColorDirection	the direction of color transitions on surfaces	[0, 0, 1]
FillColorDirectionX	x-component of the direction of color transitions on surfaces	0
FillColorDirectionY	y-component of the direction of color transitions on surfaces	0
FillColorDirectionZ	z-component of the direction of color transitions on surfaces	1
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Blue
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
LineColorType	line coloring types	Flat
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]

Attribute	Purpose	Default Value
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	0
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Name	the name of a plot object (for browser and legend)	
Normal	normal vector of circles and discs, etc. in 3D	[0, 0, 1]
NormalX	normal vector of circles and discs, etc. in 3D, x-component	0
NormalY	normal vector of circles and discs, etc. in 3D, y-component	0
NormalZ	normal vector of circles and discs, etc. in 3D, z-component	1
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
SemiAxes	semi axes of ellipses and ellipsods	
SemiAxisX	first semi axis of ellipses and ellipsods	2
SemiAxisY	second semi axis of ellipses and ellipsods	1

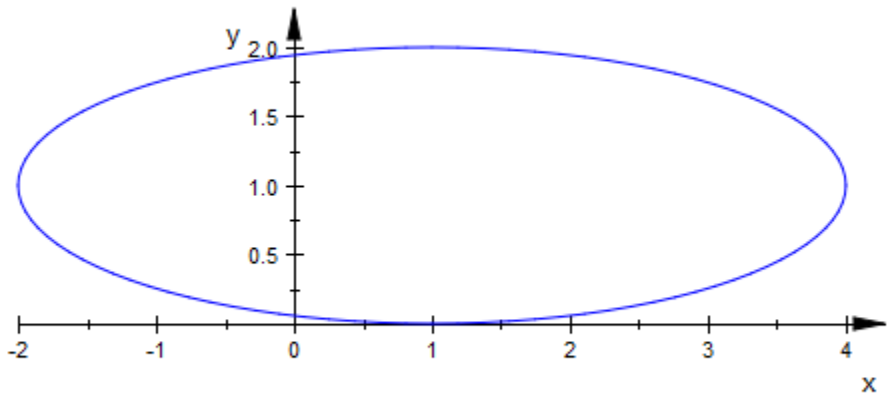
Attribute	Purpose	Default Value
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE

Examples

Example 1

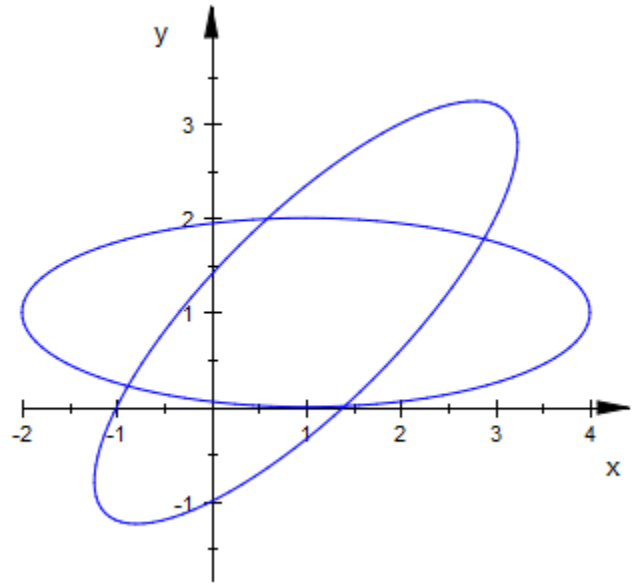
We create a plot of an ellipse with center point (1, 1) and semi-axes of lengths 3 and 1:

```
ellipse := plot::Ellipse2d(3, 1, [1, 1]):
plot(ellipse)
```



We apply a rotation:

```
plot(ellipse, plot::Rotate2d(PI/4, [1, 1], ellipse))
```

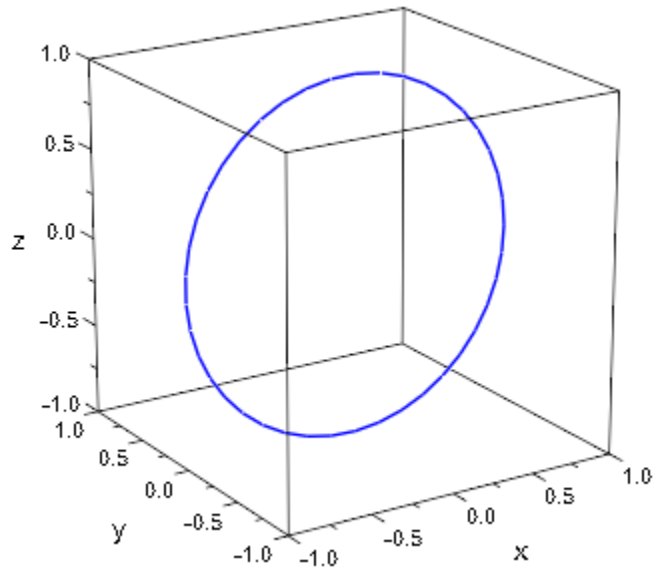



```
delete ellipse:
```

Example 2

We plot an animated 3D ellipse:

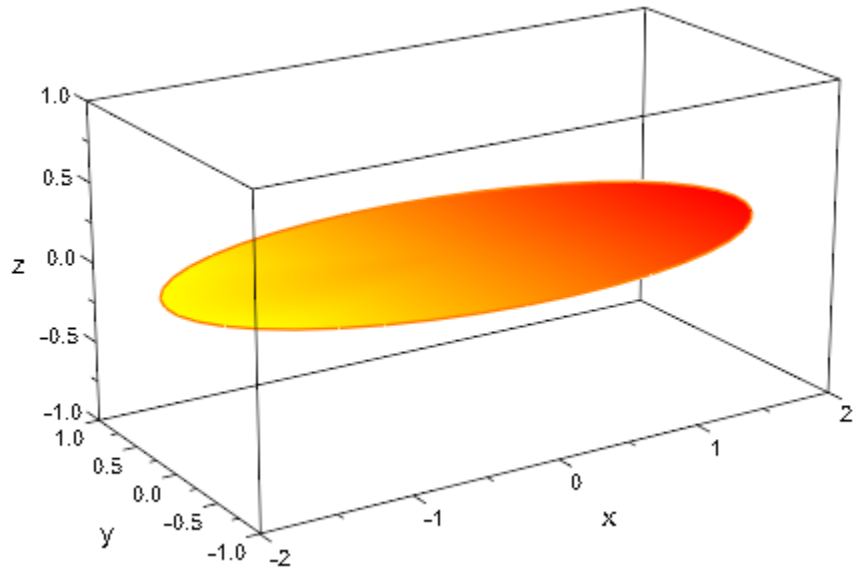
```
plot(plot::Ellipse3d(1, 1, [0,0,0], [0,a,1-a], a = 0..1))
```



Example 3

We plot a colored 3D ellipse:

```
plot(plot::Ellipse3d(2, 1, [0,0,0], Filled,
                     LineColor=RGB::Yellow, LineColor2=RGB::Red,
                     LineColorType = Dichromatic, FillColorDirection=[+1,0,0],
                     FillColor=RGB::Yellow, FillColor2=RGB::Red,
                     FillColorType = Dichromatic, FillColorDirection=[-1,0,0]
))
```



Parameters

r_1, r_2

The semi-axes of an ellipse. They must be real numerical values or arithmetical expressions of the animation parameter `a`.

r_1, r_2 are equivalent to the attributes `SemiAxisX`, `SemiAxisY`.

c_x, c_y, c_z

The center. The coordinates c_x, c_y, c_z must be real numerical values or arithmetical expressions of the animation parameter `a`. If no center is specified, the ellipse is centered at the origin.

c_x, c_y, c_z are equivalent to the attribute `Center`.

n_x, n_y, n_z

The normal vector. The coordinates n_x, n_y, n_z must be real numerical values or arithmetical expressions of the animation parameter `a`. If no normal vector is specified, the ellipse is created in the xy-plane.

n_x, n_y, n_z are equivalent to the attribute `Normal`.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Arc2d` | `plot::Arc3d` | `plot::Circle2d` | `plot::Ellipse2d`

plot::Function2d

2D function graphs

Syntax

```
plot::Function2d(f, options)
```

```
plot::Function2d(f, x = x_min .. x_max, <a = a_min .. a_max>, options)
```

Description

`plot::Function2d` creates the 2D graph of a univariate function.

The graphics ignores all points, where the expression/function `f` does not produce a numerical real value. See "Example 2" on page 24-348.

The expression/function `f` may have singularities in the plot range. Although a heuristics is used to find a reasonable *y* range when singularities are present, it is highly recommended to specify a *y* range via `ViewingBoxYRange = `y_{min}` .. `y_{max}`` with suitable numerical real values y_{min}, y_{max}. See "Example 3" on page 24-349.

Animations are triggered by specifying a range `a = `a_{min}` .. `a_{max}`` for a parameter `a` that is different from the independent variable `x`. Thus, in animations, both the *x*-range `x = `x_{min}` .. `x_{max}`` as well as the animation range `a = `a_{min}` .. `a_{max}`` must be specified. See "Example 4" on page 24-351.

The function `f` is evaluated on an equidistant mesh of sample points determined by the attribute `XMesh` (or the shorthand notation `Mesh`). By default, the attribute `AdaptiveMesh = 0` is set, i.e., no adaptive refinement of the equidistant mesh is used.

If the standard mesh does not suffice to produce a sufficiently detailed plot, one may either increase the value of `XMesh` or set `AdaptiveMesh = n` with some (small) positive integer `n`. If necessary, up to 2^n additional points are placed between adjacent points of the initial equidistant mesh. See "Example 5" on page 24-352.

By default, the attribute `DiscontinuitySearch = TRUE` is set. This triggers a semi-symbolic preprocessing of the expression `f` to search for discontinuities and singularities.

At each singular point, the function graph is split into disjoint branches to the left and to the right of the singularity. This avoids graphical artifacts such as lines connecting points to the left and to the right of a singularity.

If the function is known to be regular in the plot range, the semi-symbolic search may be disabled by specifying `DiscontinuitySearch = FALSE`. This will improve the efficiency of the plot commands.

Singular points are highlighted by a vertical line unless `VerticalAsymptotesVisible = FALSE` is specified. Its style may be set by the attributes `VerticalAsymptotesStyle`, `VerticalAsymptotesWidth`, and `VerticalAsymptotesColor`.

Note: This functionality is only available if the function is specified by a an arithmetical expression or a procedure that accepts symbolic arguments. It is not available if the function is specified by a `piecewise` object or by a procedure that accepts only numerical arguments.

See “Example 6” on page 24-356.

`plot::Hatch` allows to hatch areas between function graphs. See “Example 7” on page 24-358.

Attributes

Attribute	Purpose	Default Value
`AdaptiveMesh`	adaptive sampling	`2`
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`
`Color`	the main color	`RGB::Blue`
`DiscontinuitySearch`	semi-symbolic search for discontinuities	`TRUE`
`Frames`	the number of frames in an animation	`50`

Attribute	Purpose	Default Value
`Function`	function expression or procedure	
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`TRUE`
`LineColor`	color of lines	`RGB::Blue`
`LineWidth`	width of lines	`0.35`
`LineColor2`	color of lines	`RGB::DeepPink`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LinesVisible`	visibility of lines	`TRUE`
`LineColorType`	line coloring types	`Flat`
`LineColorFunction`	functional line coloring	
`LineColorDirection`	the direction of color transitions on lines	[0, 1]
`LineColorDirectionX`	x-component of the direction of color transitions on lines	`0`
`LineColorDirectionY`	y-component of the direction of color transitions on lines	`1`
`Mesh`	number of sample points	`121`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	

Attribute	Purpose	Default Value
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE
Submesh	density of submesh (additional sample points)	0
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
VerticalAsymptotesColo	color of vertical asymptotes indicating poles	RGB::Grey50
VerticalAsymptotesWidt	line width of vertical asymptotes indicating poles	0.2
VerticalAsymptotesStyl	line style of vertical asymptotes indicating poles	Dashed
VerticalAsymptotesVisi	vertical asymptotes indicating poles	TRUE
Visible	visibility	TRUE

Attribute	Purpose	Default Value
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`
`XMax`	final value of parameter “x”	`5`
`XMesh`	number of sample points for parameter “x”	`121`
`XMin`	initial value of parameter “x”	`-5`
`XName`	name of parameter “x”	
`XRange`	range of parameter “x”	`-5 .. 5`
`XSubmesh`	density of additional sample points for parameter “x”	`0`

Examples

Example 1

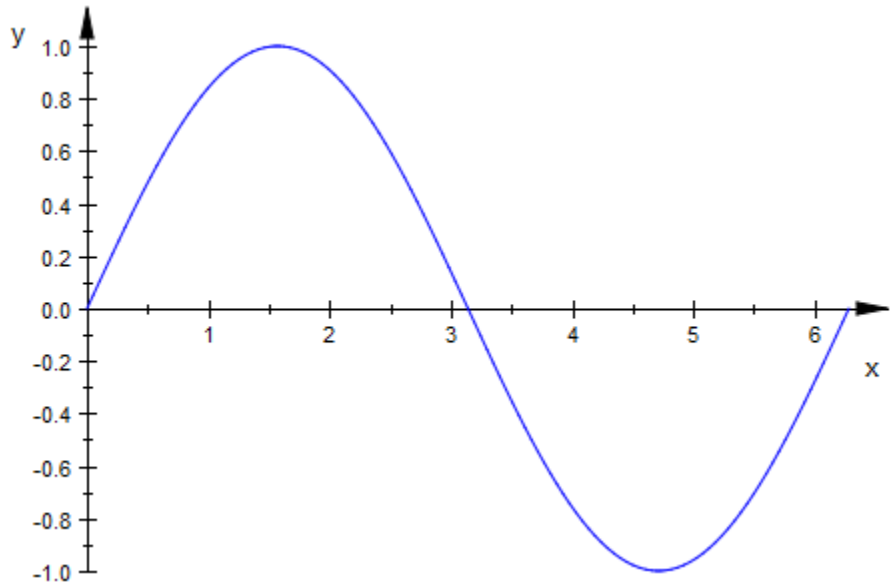
The following call returns an object representing the graph of the sine function over the interval [0, 2 π]:

```
f := plot::Function2d(sin(x), x = 0 .. 2*PI)
```

$$\text{plot::Function2d}(\sin(x),\ x = 0..2\,\pi)$$

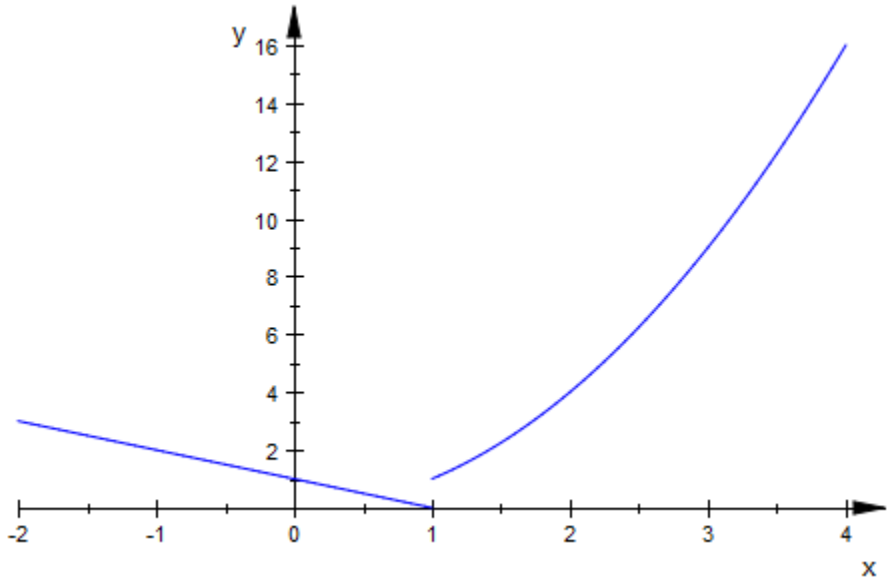
Call `plot` to plot the graph:

```
plot(f):
```

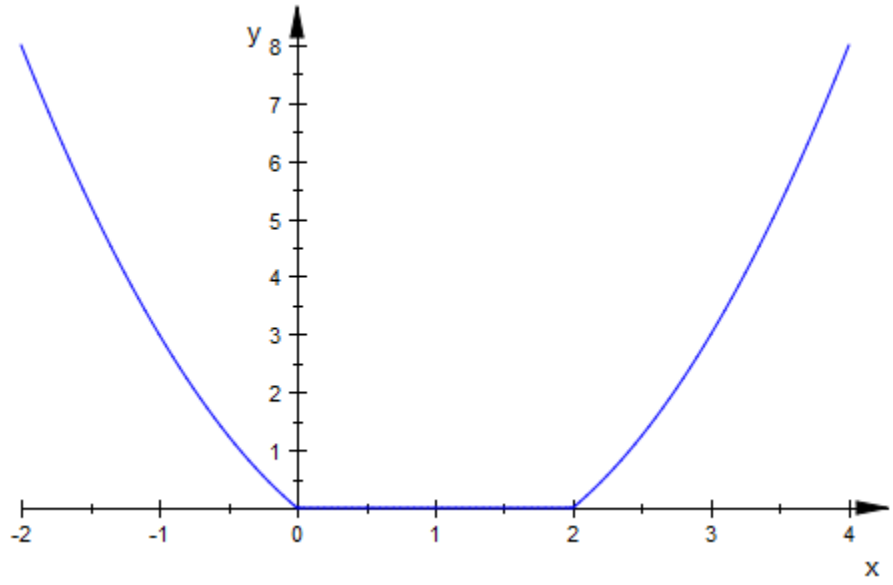


Functions can also be specified by `piecewise` objects or procedures:

```
f := piecewise([x < 1, 1 - x], [x >= 1, x^2]):
plot(plot::Function2d(f, x = -2 .. 4)):
```




```
f := proc(x)
begin
  if x^2 - 2*x < 0 then
     0
  else
     x^2 - 2*x
  end_if:
end_proc:
plot(plot::Function2d(f, x = -2 .. 4)):
```

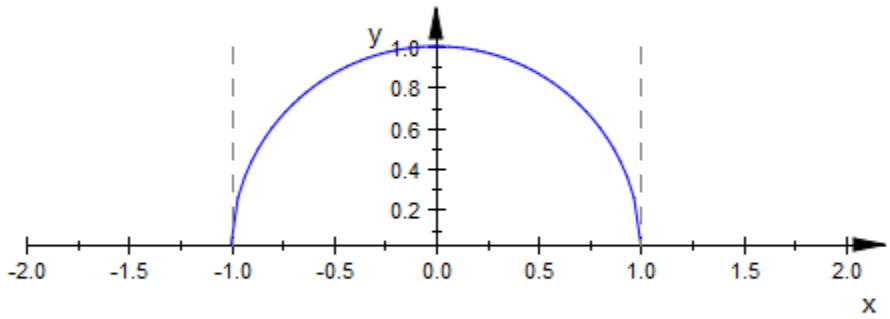



```
delete f:
```

Example 2

Non-real values are ignored in a plot:

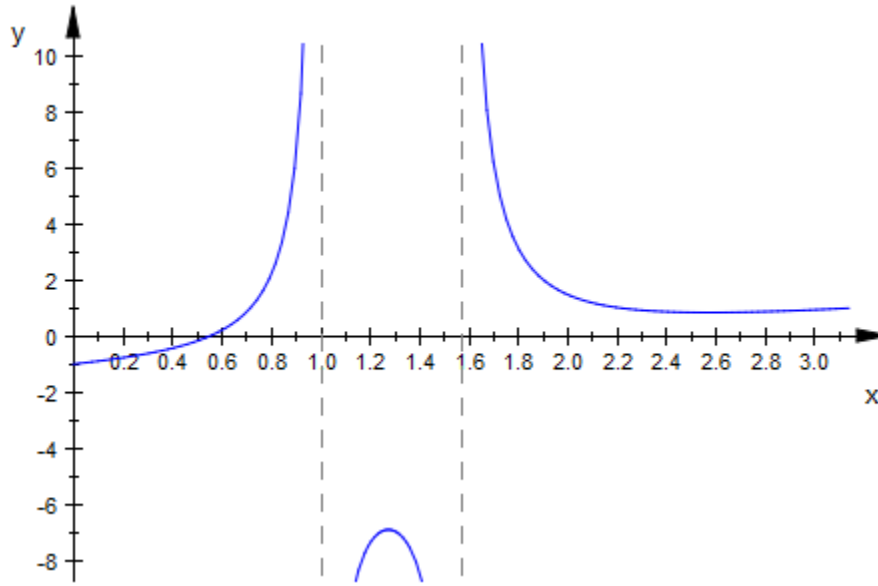
```
plot(plot::Function2d(sqrt(1 - x^2), x = -2 .. 2),
     Scaling = Constrained):
```



Example 3

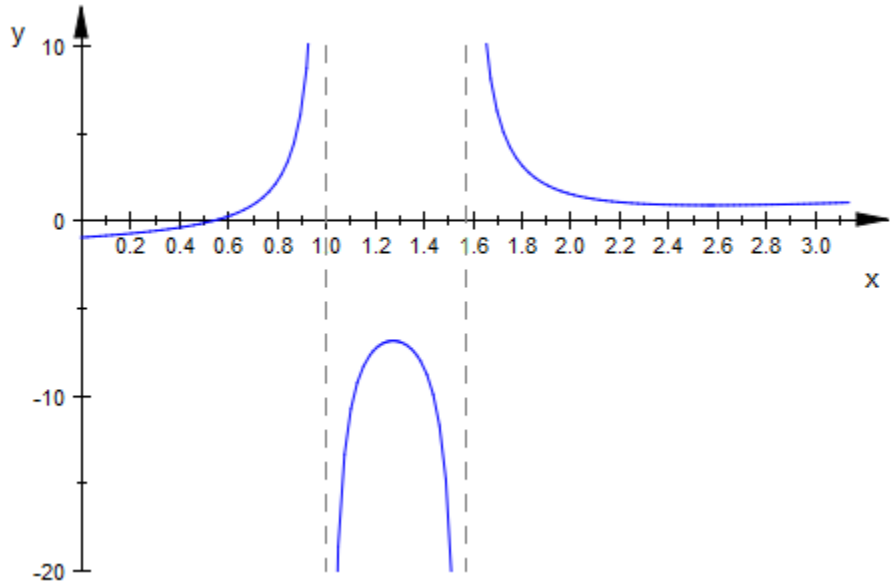
We plot a function with singularities:

```
f := plot::Function2d(sin(x)/(1 - x) - 1/cos(x), x = 0 .. PI):
plot(f):
```



We specify an explicit viewing range for the *y* direction:

```
plot(f, ViewingBoxYRange = -20 .. 10):
```

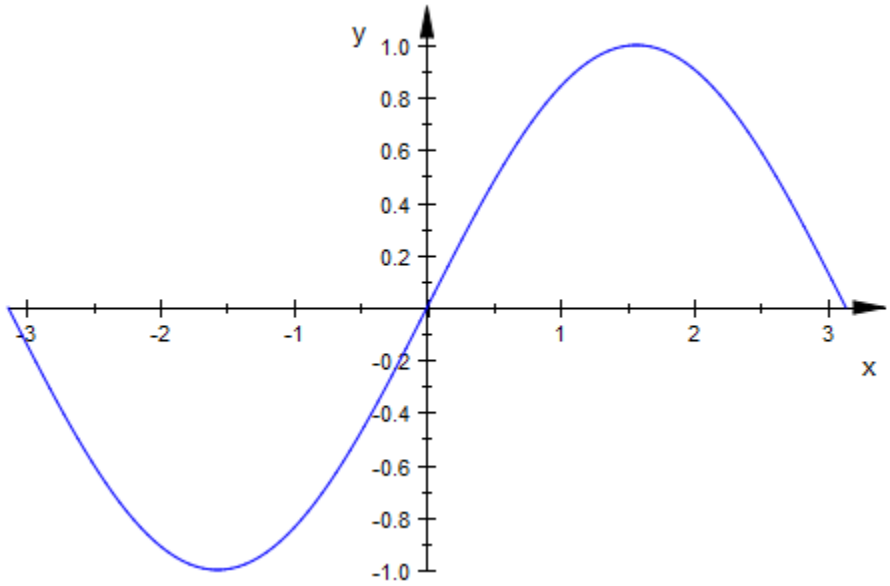



```
delete f:
```

Example 4

We generate an animation of a parametrized function:

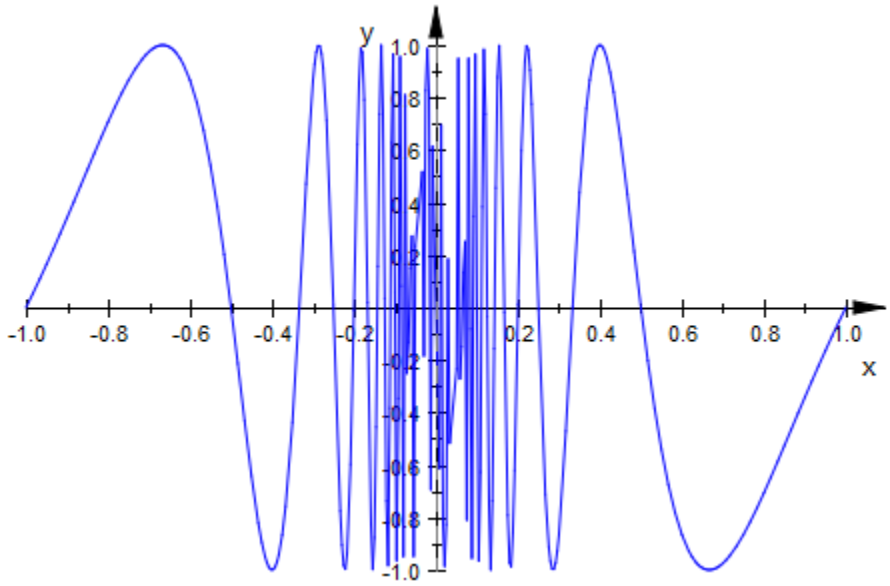
```
plot(plot::Function2d(a*sin(x) + (1 - a)*cos(x),
                      x = -PI .. PI, a = 0 .. 1)):
```



Example 5

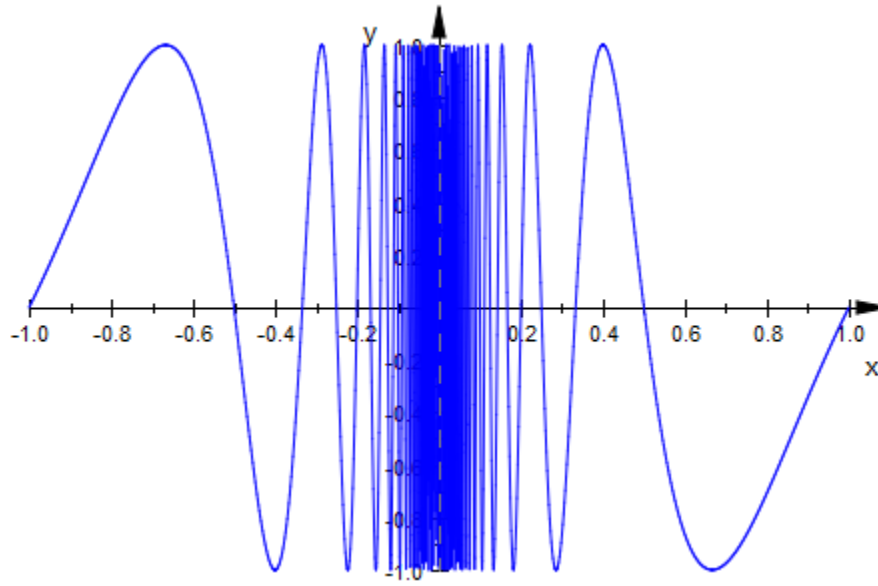
The standard mesh for the numerical evaluation of a function graph does not suffice to generate a satisfying graphics in the following case:

```
plot(plot::Function2d(sin(PI/x), x = -1 .. 1)):
```



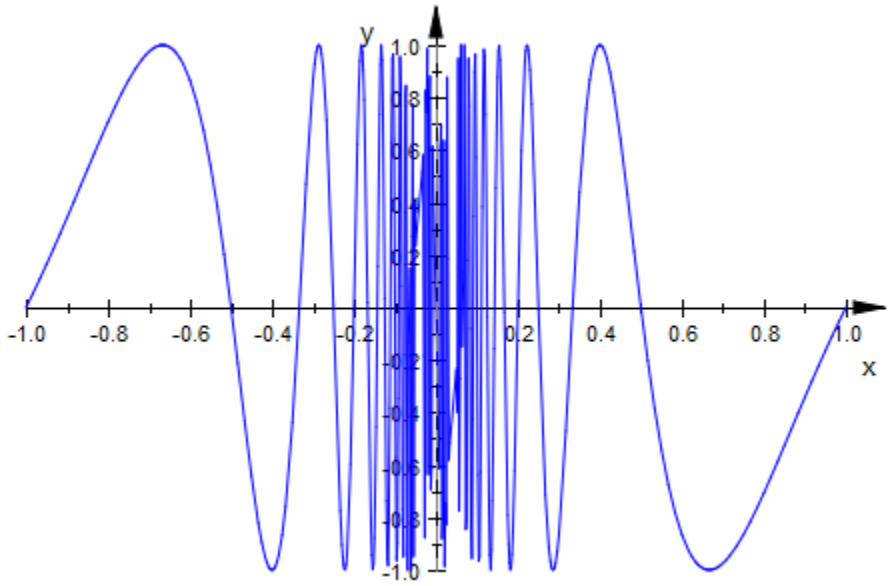
We increase the number of mesh points:

```
plot(plot::Function2d(sin(PI/x), x = -1 .. 1, XMesh = 1000)):
```



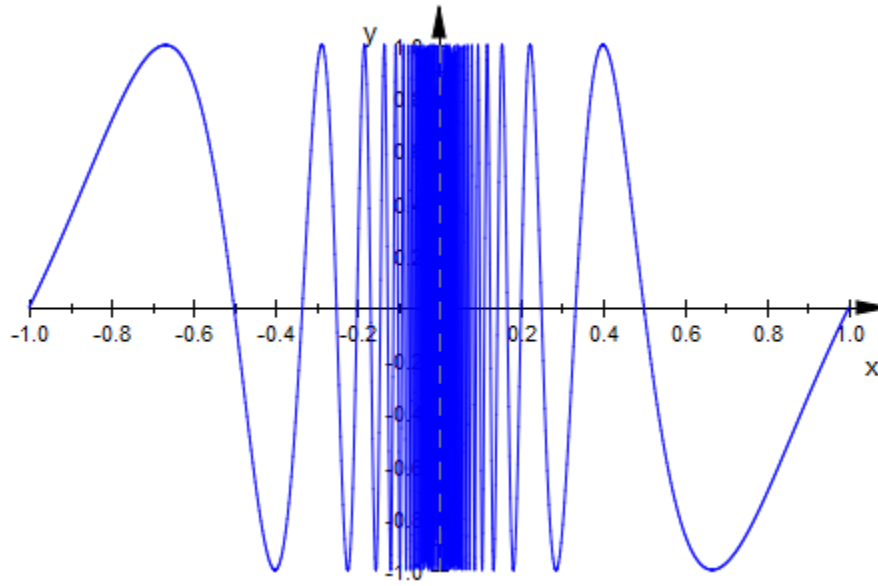
Alternatively, we enable adaptive sampling by setting `AdaptiveMesh` to some positive value:

```
plot(plot::Function2d(sin(PI/x), x = -1 .. 1, AdaptiveMesh = 3)):
```



Finally, we increase the `XMesh` value and use adaptive sampling:

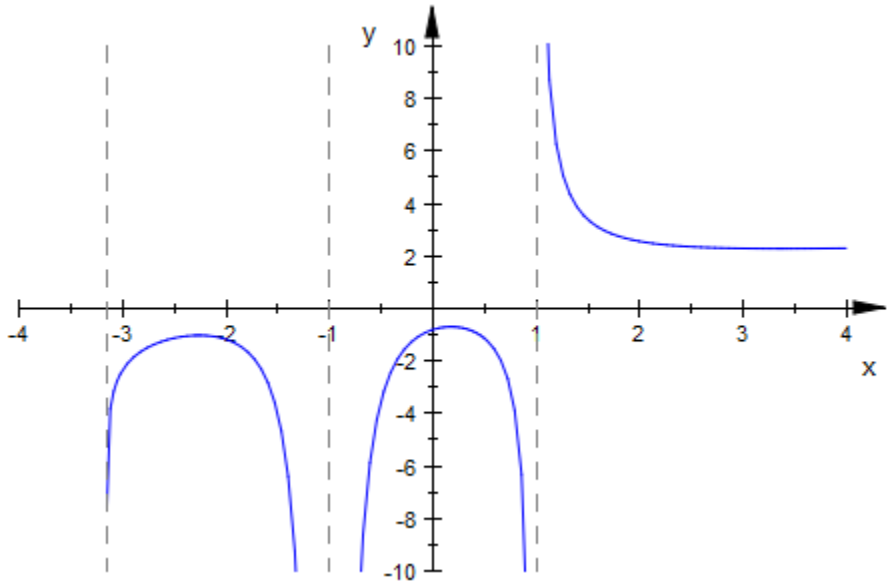
```
plot(plot::Function2d(sin(PI/x), x = -1 .. 1, XMesh = 1000,
                      AdaptiveMesh = 3)):
```



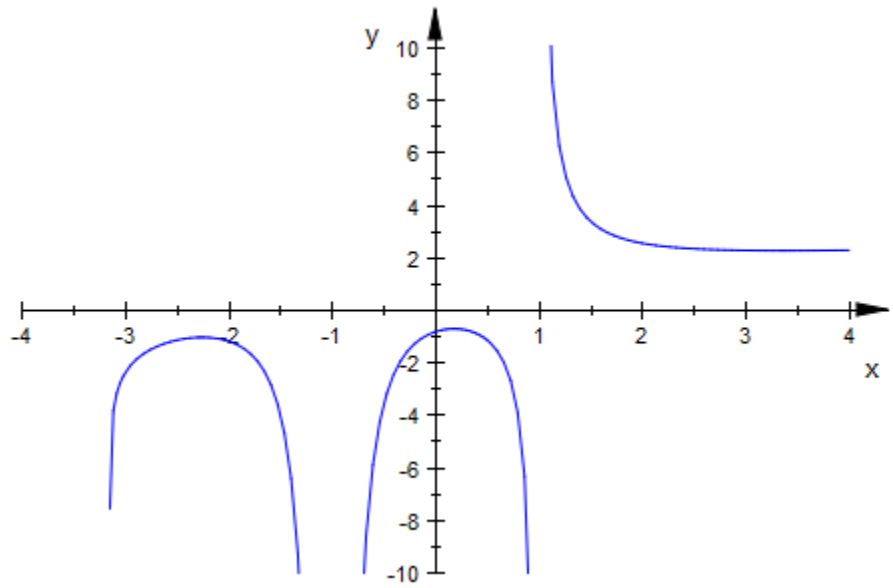
Example 6

With `VerticalAsymptotesVisible = TRUE/FALSE`, singular points are highlighted by a vertical asymptote, or this highlighting is switched off, respectively:

```
plot(plot::Function2d(ln(x + PI) + 1/(x - 1) - 1/(x + 1)^2,
                      x = -4 .. 4,
                      VerticalAsymptotesVisible = TRUE,
                      ViewingBoxYRange = -10 .. 10)):
```



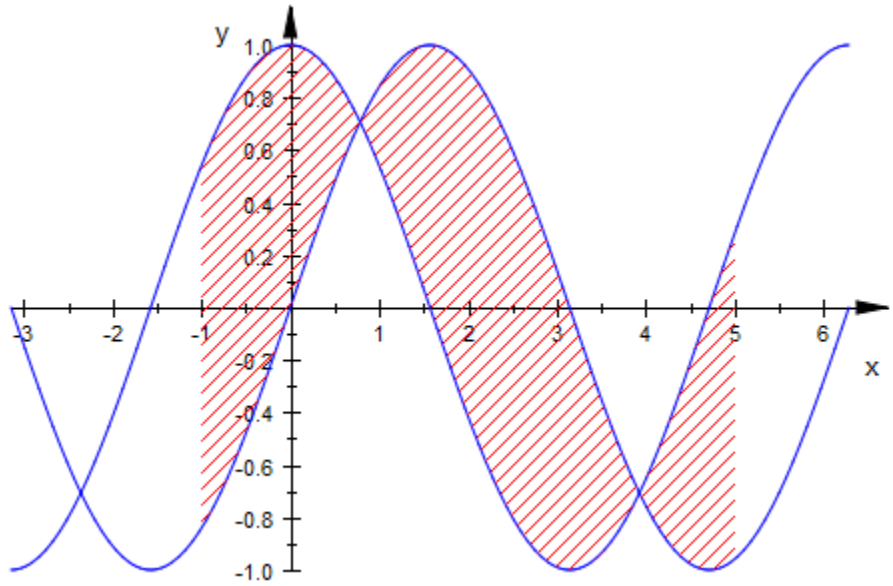

```
plot(plot::Function2d(ln(x + PI) + 1/(x - 1) - 1/(x + 1)^2,
                      x = -4 .. 4,
                      VerticalAsymptotesVisible = FALSE,
                      ViewingBoxYRange = -10 .. 10)):
```



Example 7

The `plot::Hatch` object allows to hatch regions between functions. It expects graphical objects of type `plot::Function2d` or `plot::Curve2d` as boundaries:

```
f1:= plot::Function2d(sin(x), x = -PI .. 2*PI):
f2:= plot::Function2d(cos(x), x = -PI .. 2*PI):
plot(f1, f2, plot::Hatch(f1, f2, -1 .. 5)):
```




```
delete f1, f2:
```

Parameters

f

The function: an arithmetical expression or a `piecewise` object in the independent variable x and the animation parameter a. Alternatively, a procedure that accepts 1 input parameter x or 2 input parameters x, a and returns a real numerical value when the input parameters are numerical.

f is equivalent to the attribute `Function`.

x

The independent variable: an identifier or an indexed identifier.

x is equivalent to the attribute `XName`.

x_{min} .. x_{max}

The plot range: x_{min}, x_{max} must be numerical real values or expressions of the animation parameter a. If not specified, the default range `x = -5 .. 5` is used.

x_{min} .. x_{max} is equivalent to the attributes `XRange`, `XMin`, `XMax`.

a

Animation parameter, specified as a `= a`$_{min}$`..a`$_{max}$, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions

`plot` | `plot::copy` | `plotfunc2d` | `plotfunc3d`

MuPAD Graphical Primitives

`plot::Function3d`

plot::Function3d

3D function graphs

Syntax

```
plot::Function3d(f, options)
```

```
plot::Function3d(f, x = x_min .. x_max, y = y_min .. y_max, <a = a_min .. a_max>, options)
```

Description

`plot::Function3d` creates the 3D graph of a function in 2 variables.

The expression `f(x, y)` is evaluated at finitely many points x, y in the plot range. There may be singularities. Although a heuristics is used to find a reasonable z range when singularities are present, it is highly recommended to specify a z range via `ViewingBoxZRange = `z_{min}` .. `z_{max}`` with suitable numerical real values z_{min}, z_{max}. See "Example 2" on page 24-369.

Animations are triggered by specifying a range `a = `a_{min}` .. `a_{max}`` for a parameter `a` that is different from the indedependent variables `x`, `y`. Thus, in animations, the $x$-range `x = `x_{min}` .. `x_{max}``, the y-range `y = `y_{min}` .. `y_{max}`` as well as the animation range `a = `a_{min}` .. `a_{max}`` must be specified. See "Example 3" on page 24-371.

The function `f` is evaluated on a regular equidistant mesh of sample points determined by the attributes `XMesh` and `YMesh` (or the shorthand-notation for both, `Mesh`). By default, the attribute `AdaptiveMesh = 0` is set, i.e., no adaptive refinement of the equidistant mesh is used.

If the standard mesh does not suffice to produce a sufficiently detailed plot, one may either increase the value of `XMesh` and `YMesh` or set `AdaptiveMesh = n` with some (small) positive integer `n`. This may result in up to 4^n times as many triangles as used with `AdaptiveMesh = 0`, potentially more when `f` has non-isolated singularities. See "Example 4" on page 24-372.

The "coordinate lines" ("parameter lines") are curves on the function graph.

The phrase "XLines" refers to the curves $(x, y_0, f(x, y_0))$ with the parameter x running from x_{min} to x_{max}, while y_0 is some fixed value from the interval $[y_{min}, y_{max}]$.

The phrase "YLines" refers to the curves $(x_0, y, f(x_0, y))$ with the parameter y running from ymin to ymax, while x_0 is some fixed value from the interval $[x_{min}, x_{max}]$.

By default, the parameter lines are visible. They may be "switched off" by specifying XLinesVisible = FALSE and YLinesVisible = FALSE, respectively.

The coordinate lines controlled by XLinesVisible = TRUE/FALSE and YLinesVisible = TRUE/FALSE indicate the equidistant regular mesh set via the Mesh attributes. If the mesh is refined by the Submesh attributes or by the adaptive mechanism controlled by AdaptiveMesh = n, no additional parameter lines are drawn.

As far as the numerical approximation of the function graph is concerned, the settings

$\texttt{Mesh} = [n_x, n_y], \texttt{Submesh} = [m_x, m_y]$

and

$\texttt{Mesh} = [(n_x - 1)\ (m_x + 1) + 1, (n_y - 1)\ (m_y + 1) + 1], \texttt{Submesh} = [0, 0]$

are equivalent. However, in the first setting, nx parameter lines are visible in the x direction, while in the latter setting $(n_x - 1)\ (m_x + 1) + 1$ parameter lines are visible. See "Example 5" on page 24-375.

Attributes

Attribute	Purpose	Default Value
AdaptiveMesh	adaptive sampling	0
AffectViewingBox	influence of objects on the ViewingBox of a scene	TRUE
Color	the main color	RGB::Red
Filled	filled or transparent areas and surfaces	TRUE
FillColor	color of areas and surfaces	RGB::Red
FillColor2	second color of areas and surfaces for color blends	RGB::CornflowerBlue

Attribute	Purpose	Default Value
FillColorType	surface filling types	Dichromatic
FillColorFunction	functional area/surface coloring	
FillColorDirection	the direction of color transitions on surfaces	[0, 0, 1]
FillColorDirectionX	x-component of the direction of color transitions on surfaces	0
FillColorDirectionY	y-component of the direction of color transitions on surfaces	0
FillColorDirectionZ	z-component of the direction of color transitions on surfaces	1
Frames	the number of frames in an animation	50
Function	function expression or procedure	
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	TRUE
LineColor	color of lines	RGB::Black.[0.25]
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	

Attribute	Purpose	Default Value
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	0
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Mesh	number of sample points	[25, 25]
MeshVisible	visibility of irregular mesh lines in 3D	FALSE
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE
Shading	smooth color blend of surfaces	Smooth
Submesh	density of submesh (additional sample points)	[0, 0]
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0

Attribute	Purpose	Default Value
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XLinesVisible	visibility of parameter lines (x lines)	TRUE
XMax	final value of parameter “x”	5
XMesh	number of sample points for parameter “x”	25
XMin	initial value of parameter “x”	-5

Attribute	Purpose	Default Value
XName	name of parameter “x”	
XRange	range of parameter “x”	-5 .. 5
XSubmesh	density of additional sample points for parameter “x”	0
YLinesVisible	visibility of parameter lines (y lines)	TRUE
YMax	final value of parameter “y”	5
YMesh	number of sample points for parameter “y”	25
YMin	initial value of parameter “y”	-5
YName	name of parameter “y”	
YRange	range of parameter “y”	-5 .. 5
YSubmesh	density of additional sample points for parameter “y”	0
ZContours	contour lines at constant z values	[]

Examples

Example 1

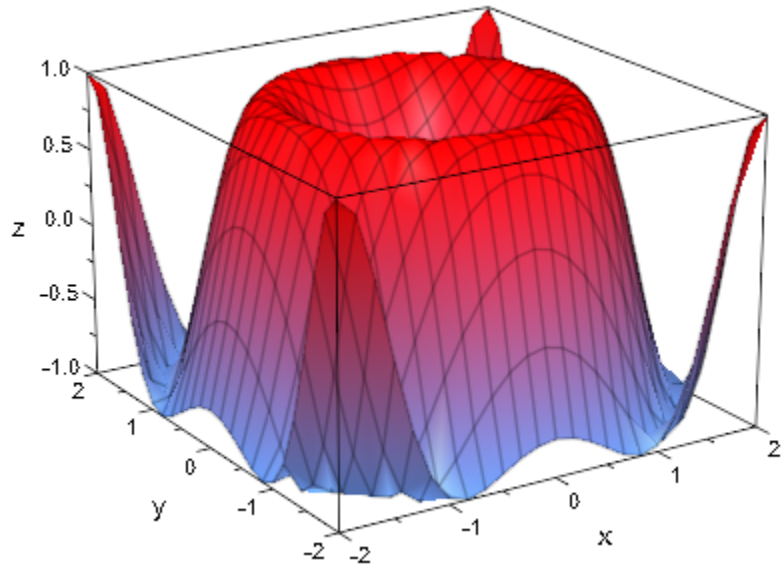
The following call returns an object representing the graph of the function $sin(x^2 + y^2)$ over the region $-2 \le x \le 2$, $-2 \le y \le 2$:

```
g := plot::Function3d(sin(x^2 + y^2), x = -2..2, y = -2..2)
```

$$\text{plot::Function3d}(\sin(x^2 + y^2), x = -2..2, y = -2..2)$$

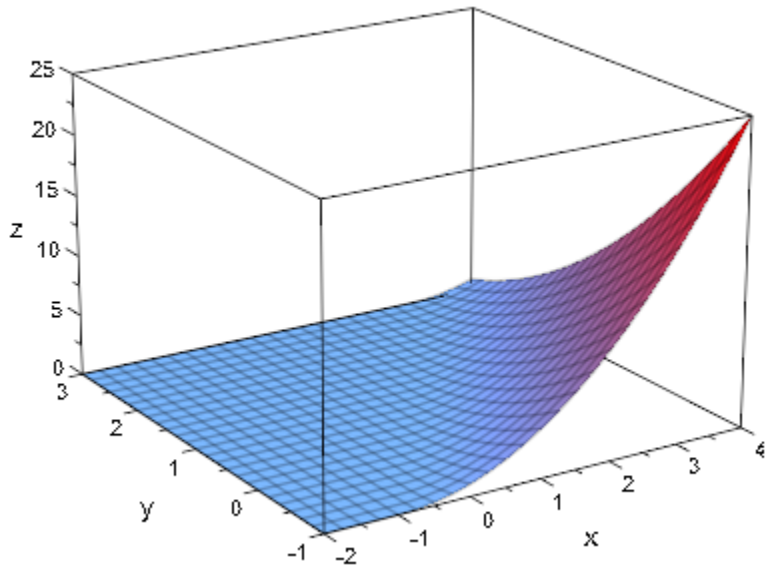
Call `plot` to plot the graph:

```
plot(g)
```

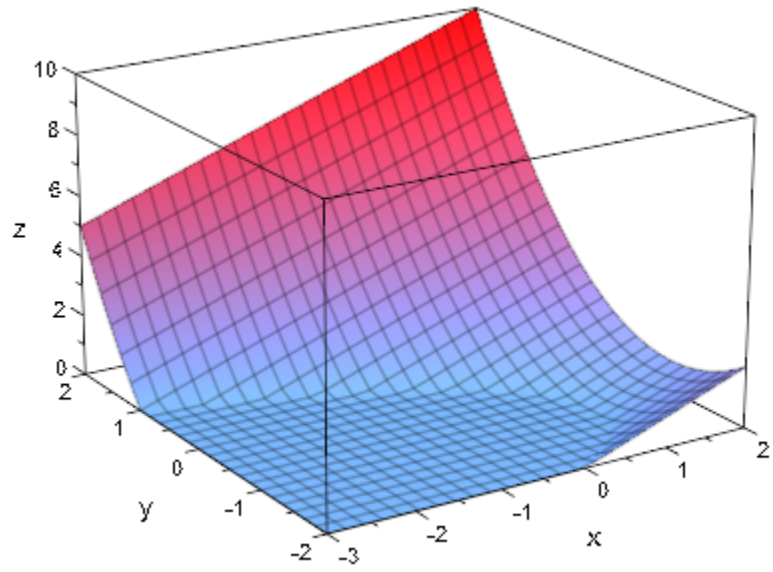


Functions can also be specified by `piecewise` objects or procedures:

```
f := piecewise([x < y, 0], [x >= y, (x - y)^2]):
plot(plot::Function3d(f, x = -2 .. 4, y = -1 .. 3))
```




```
f := proc(x, y)
begin
  if x + y^2 + 2*y < 0 then
     0
  else
     x + y^2 + 2*y
  end_if:
end_proc:
plot(plot::Function3d(f, x = -3 .. 2, y = -2 .. 2))
```

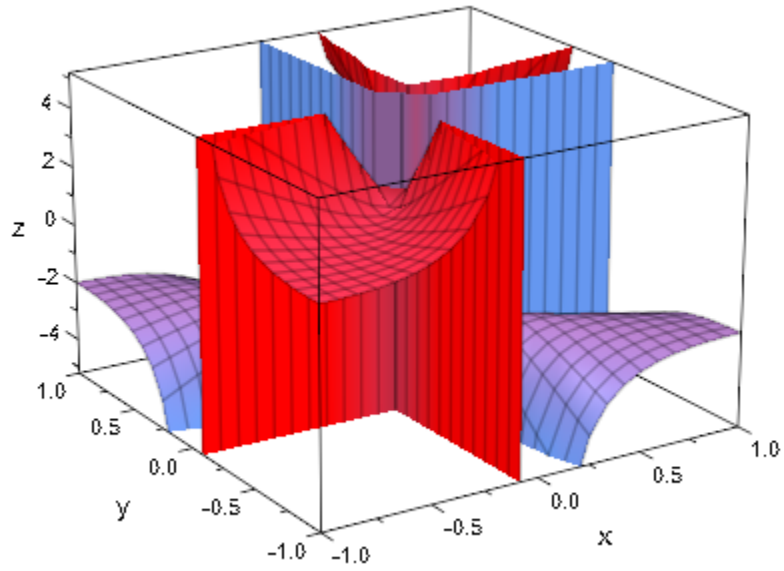



```
delete g, f
```

Example 2

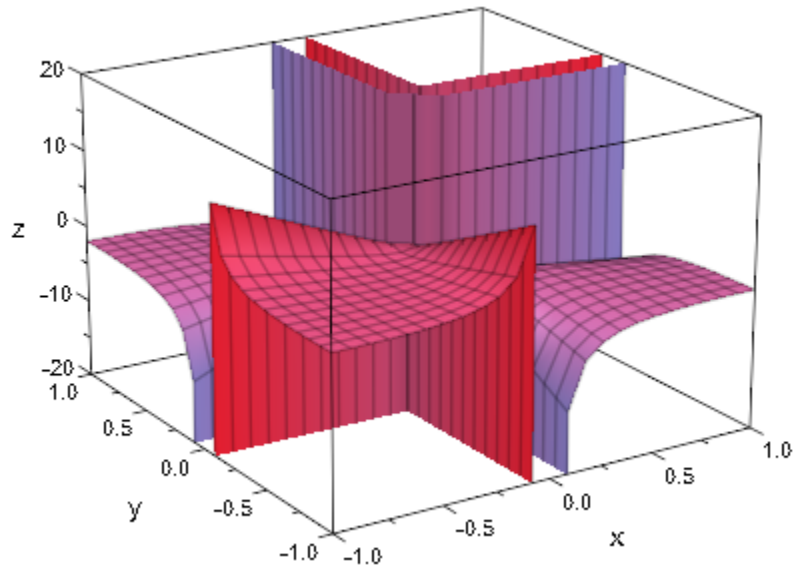
We plot a function with singularities:

```
f := plot::Function3d(x/y + y/x, x = -1 .. 1, y = - 1 .. 1):
plot(f)
```



We specify an explicit viewing range for the *z* direction:

```
plot(f, ViewingBoxZRange = -20 .. 20)
```

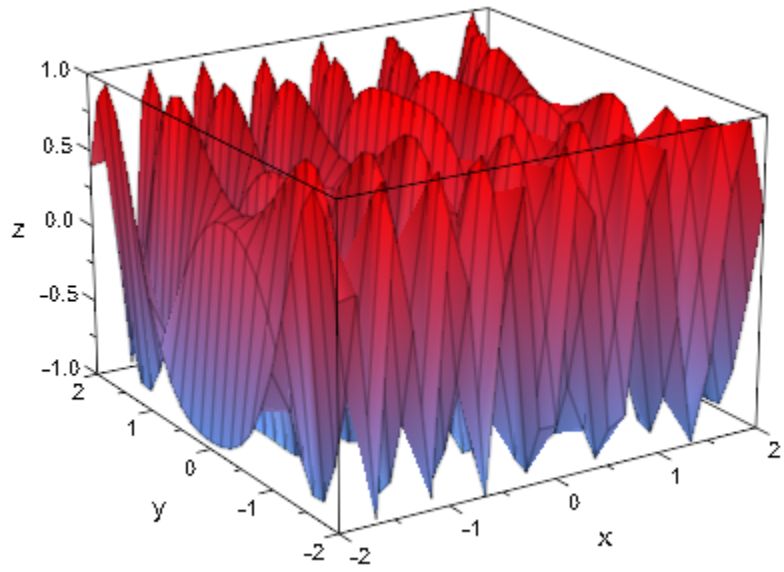



```
delete f
```

Example 3

We generate an animation of a parametrized function:

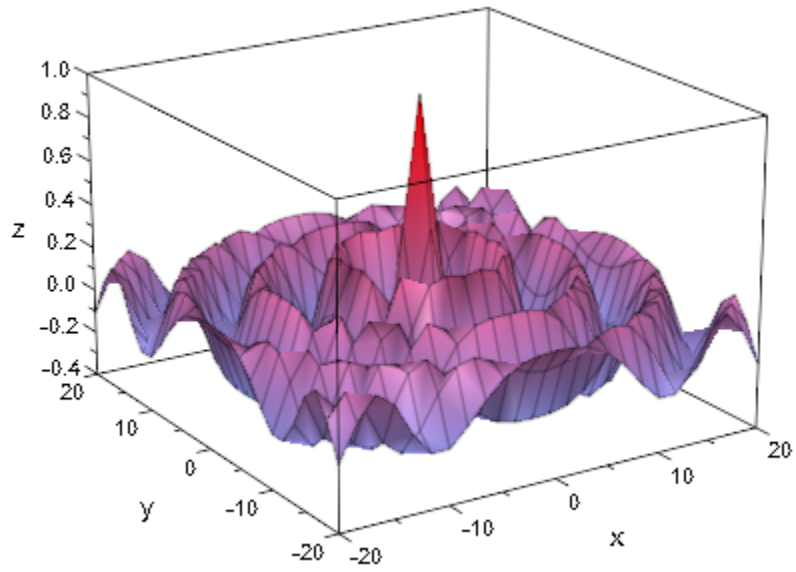
```
plot(plot::Function3d(sin((x - a)^2 + y^2),
                      x = -2 .. 2, y = -2 .. 2, a = 0 .. 5))
```



Example 4

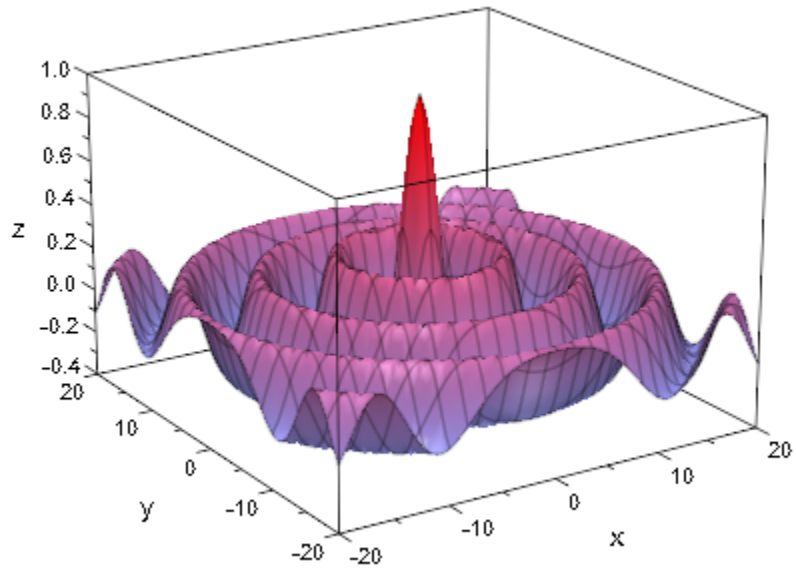
The standard mesh for the numerical evaluation of a function graph does not suffice to generate a satisfying graphics in the following case:

```
plot(plot::Function3d(besselJ(0, sqrt(x^2 + y^2)),
                      x = -20 .. 20, y = -20 .. 20))
```



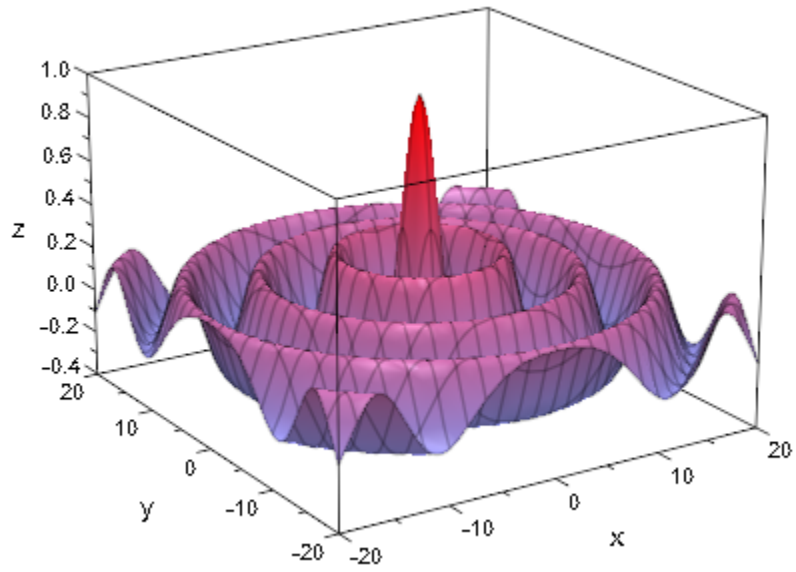
We increase the number of mesh points. Here, we use `XSubmesh` and `YSubmesh` to place 2 additional points in each direction between each pair of neighboring points of the default mesh. This increases the runtime by a factor of 9:

```
plot(plot::Function3d(besselJ(0, sqrt(x^2 + y^2)),
                      x = -20 .. 20, y = -20 .. 20,
                      Submesh = [2, 2]))
```



Alternatively, we enable adaptive sampling by setting the value of `AdaptiveMesh` to some positive value:

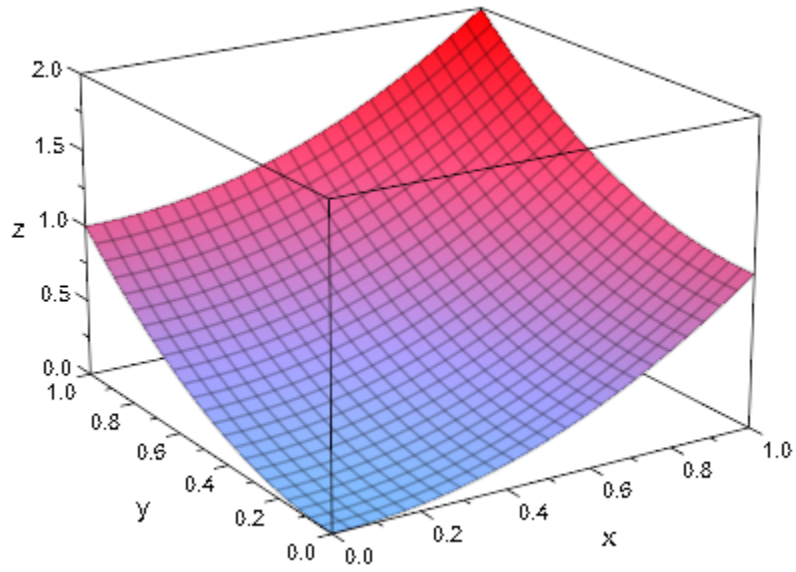
```
plot(plot::Function3d(besselJ(0, sqrt(x^2 + y^2)),
                      x = -20 .. 20, y = -20 .. 20,
                      AdaptiveMesh = 2))
```



Example 5

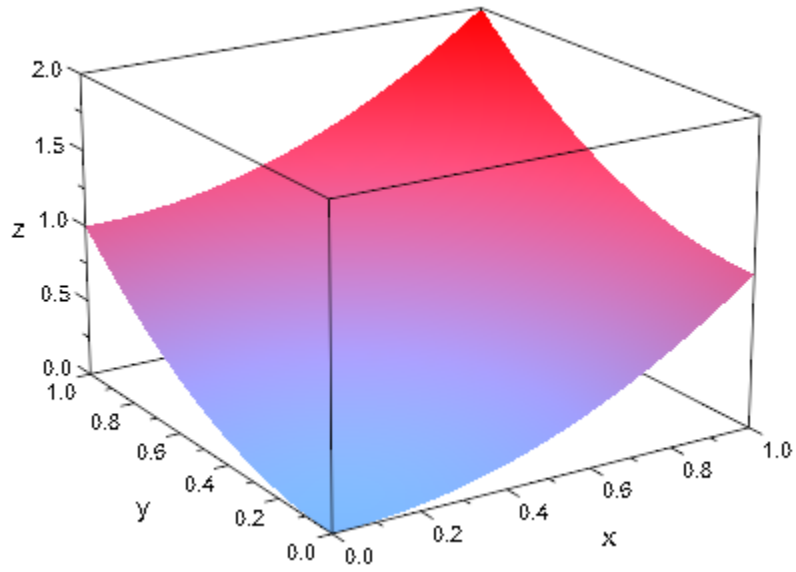
By default, the parameter lines of a function graph are “switched on”:

```
plot(plot::Function3d(x^2 + y^2, x = 0 .. 1, y = 0 .. 1))
```



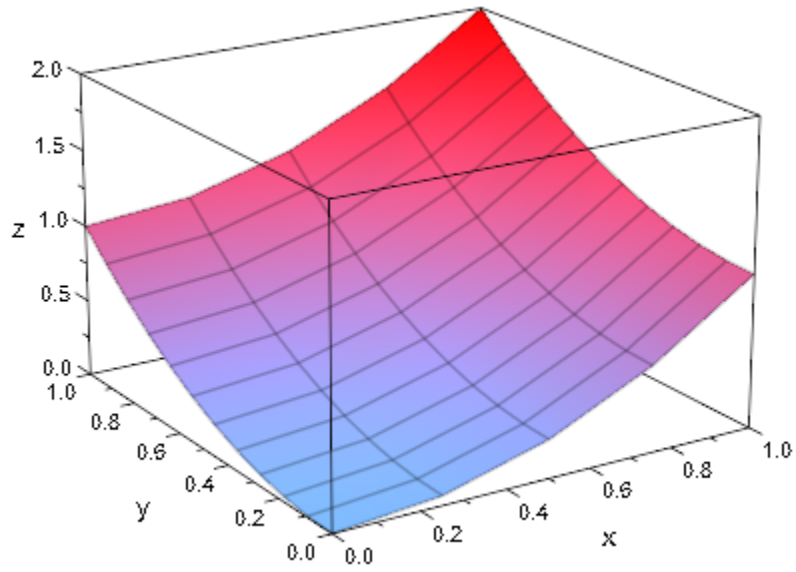
The parameter lines are "switched off" by setting `XLinesVisible`, `YLinesVisible`:

```
plot(plot::Function3d(x^2 + y^2, x = 0 .. 1, y = 0 .. 1,
                      XLinesVisible = FALSE,
                      YLinesVisible = FALSE))
```



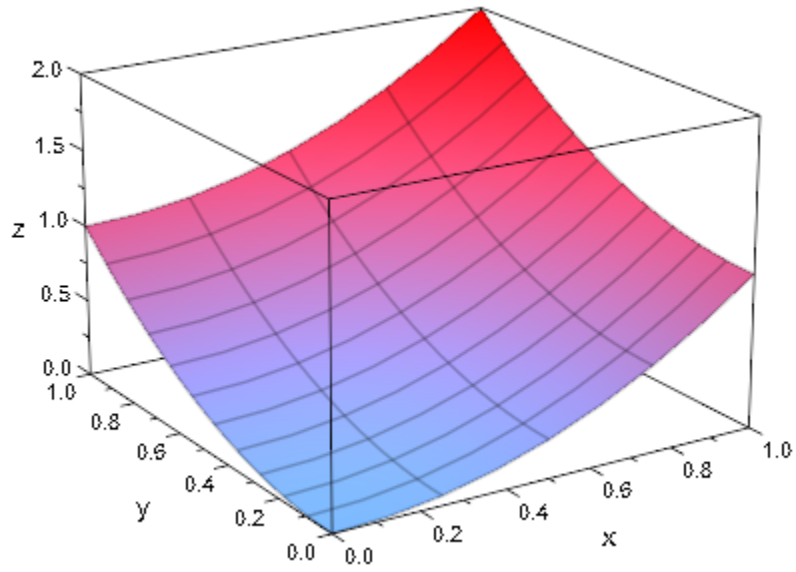
The number of parameter lines are determined by the `Mesh` attributes:

```
plot(plot::Function3d(x^2 + y^2, x = 0 .. 1, y = 0 .. 1,
                      Mesh = [5, 12]))
```



When the mesh is refined via the `Submesh` attributes, the numerical approximation of the surface becomes smoother. However, the number of parameter lines is not increased:

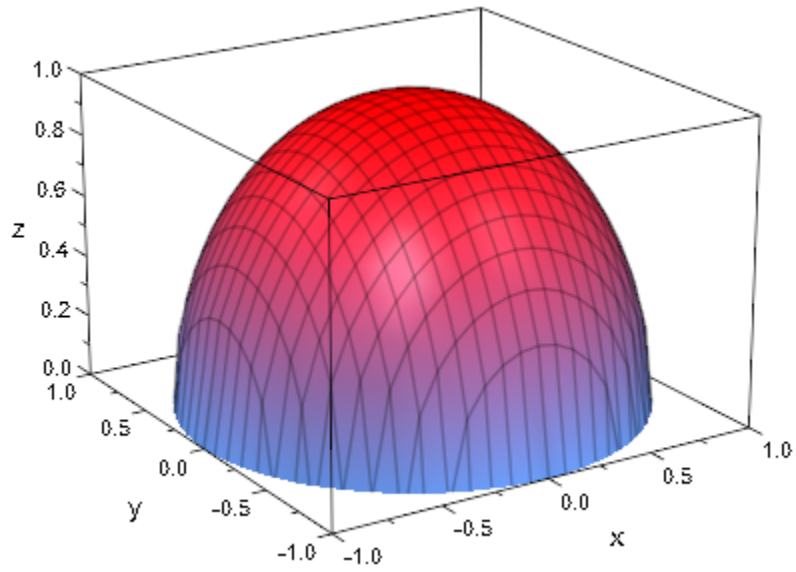
```
plot(plot::Function3d(x^2 + y^2, x = 0 .. 1, y = 0 .. 1,
                      Mesh = [5, 12],
                      XSubmesh = 1, YSubmesh = 2))
```



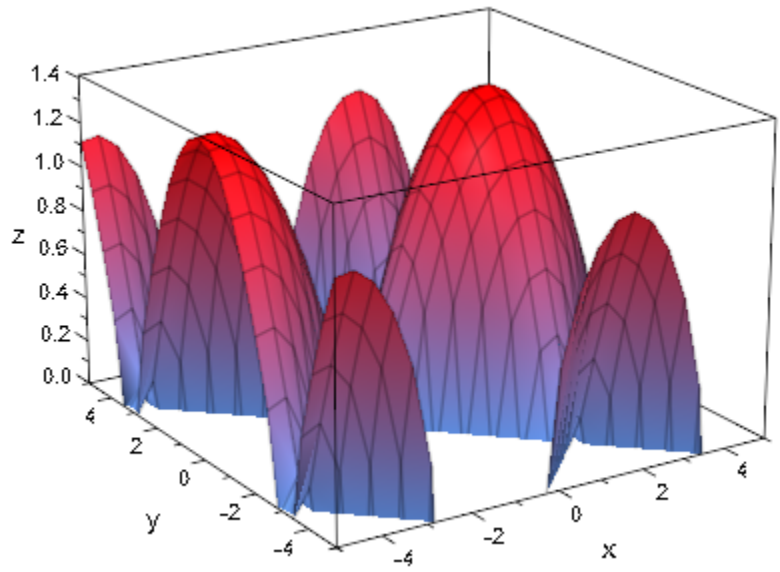
Example 6

Functions need not be defined over the whole parameter range:

```
plot(plot::Function3d(sqrt(1-x^2-y^2), x=-1..1, y=-1..1))
```




```
plot(plot::Function3d(sqrt(sin(x)+cos(y))))
```

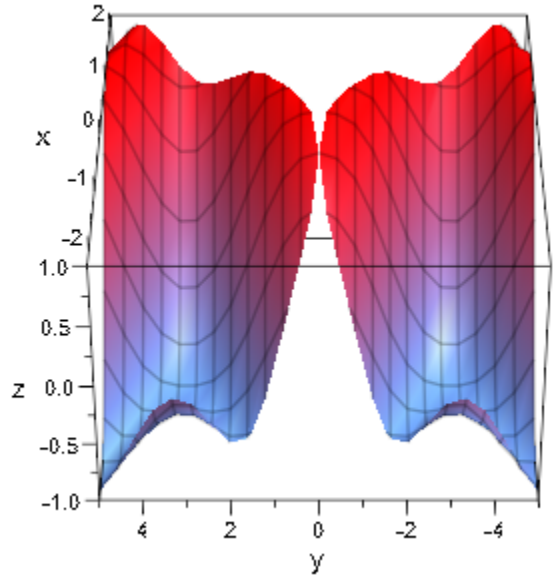


This makes for an easy way of plotting a function over a non-rectangular area:

```
chi := piecewise([x^2 < abs(y), 1])
```

$$\{1 \quad \text{if } x^2 < |y|$$

```
plot(plot::Function3d(chi*sin(x+cos(y))),
     CameraDirection=[-1,0,0.5])
```



Parameters

`f`

The function: an arithmetical expression or a `piecewise` object in the independent variables *x*, *y* and the animation parameter `a`. Alternatively, a procedure that accepts 2 input parameter *x*, *y* or 3 input parameters *x*, *y*, *a* and returns a numerical value when the input parameters are numerical.

f is equivalent to the attribute `Function`.

`x`

The first independent variable: an identifier or an indexed identifier.

x is equivalent to the attribute `XName`.

x_{min} `..` x_{max}

The plot range in *x* direction: x_{min}, x_{max} must be numerical real values or expressions of the animation parameter *a*. If not specified, the default range `x = -5 .. 5` is used.

x_{min} .. x_{max} is equivalent to the attributes `XRange`, `XMin`, `XMax`.

`y`

The second independent variable: an identifier or an indexed identifier.

y is equivalent to the attribute `YName`.

y_{min} .. y_{max}

The plot range in *y* direction: y_{min}, y_{max} must be numerical real values or expressions of the animation parameter *a*. If not specified, the default range `y = -5 .. 5` is used.

y_{min} .. y_{max} is equivalent to the attributes `YRange`, `YMin`, `YMax`.

`a`

Animation parameter, specified as $a = a_{min}..a_{max}$, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy` | `plotfunc2d` | `plotfunc3d`

MuPAD Graphical Primitives
`plot::Function2d` | `plot::Surface`

plot::Hatch

Hatched area

Syntax

```
plot::Hatch(f1, f2, <x1 .. x2>, <a = amin .. amax>, options)

plot::Hatch(f1, <base>, <x1 .. x2>, <a = amin .. amax>, options)

plot::Hatch(c, <a = amin .. amax>, options)
```

Description

`plot::Hatch(f)` hatches the area between the function `f` and the x-axis.

`plot::Hatch(f, base)` hatches the area between the function `f` and the horizontal line *y* = *base*.

`plot::Hatch(f, g)` hatches the area between the two functions `f` and `g`.

`plot::Hatch(c)` hatches the area enclosed by the curve `c`.

`plot::Hatch(f, base)` is the hatched area between a function `f` of type `plot::Function2d` and a line parallel to the x-axis with *y* = *base*. If `base` is omitted, the area between the function and the x-axis will be hatched (the baseline is assumed to be the x-axis). See “Example 1” on page 24-387.

`plot::Hatch( f1, f2)` is the hatched area between two functions f_1 and f_2. See “Example 2” on page 24-389.

`plot::Hatch(c)` is the hatched area enclosed by a `plot::Curve2d`. A curve is closed automatically by connecting the starting point and the end point. See “Example 3” on page 24-390.

The hatch may be restricted to the left and to the right by a range `x_..x_`. See “Example 4” on page 24-392.

The attributes FillColor and FillPattern can be used to change the color and pattern of the hatched area. See “Example 5” on page 24-397.

Note: A `plot::Hatch` is only the hatched area *without* outlining functions or curves! To see the border lines, you need to plot them separately as demonstrated in the examples.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`
`Baseline`	constant second function delimiting hatch	
`Color`	the main color	`RGB::Red`
`FillColor`	color of areas and surfaces	`RGB::Red`
`FillPattern`	type of area filling	`DiagonalLines`
`Frames`	the number of frames in an animation	`50`
`Function1`	first function/curve delimiting hatch	
`Function2`	second function delimiting hatch	
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	

Attribute	Purpose	Default Value
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XMax	final value of parameter “x”	infinity

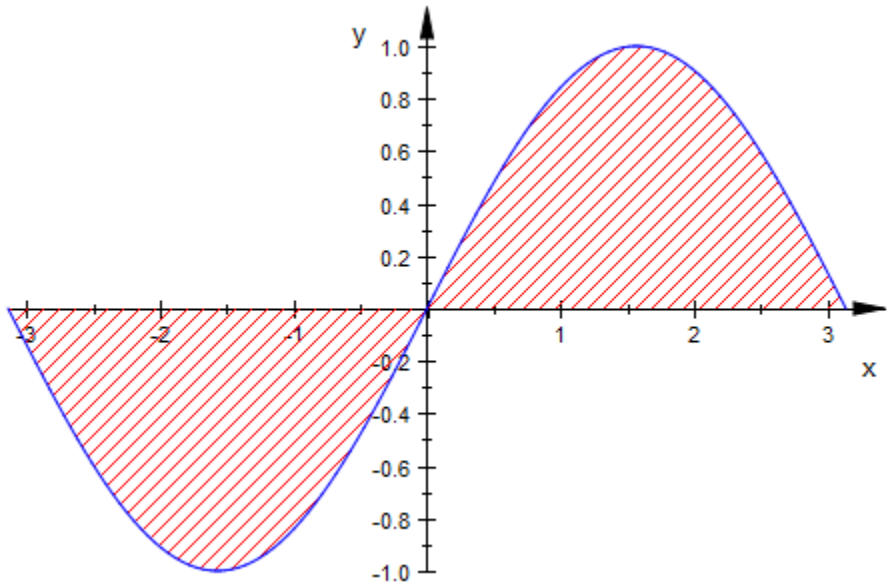
Attribute	Purpose	Default Value
`XMin`	initial value of parameter "x"	`-infinity`
`XRange`	range of parameter "x"	`-infinity .. infinity`

Examples

Example 1

If given a single `plot::Function2d` object, `plot::Hatch` hatches the area between the curve and the x-axis:

```
f := plot::Function2d(sin(x), x = -PI..PI):
plot(plot::Hatch(f), f)
```



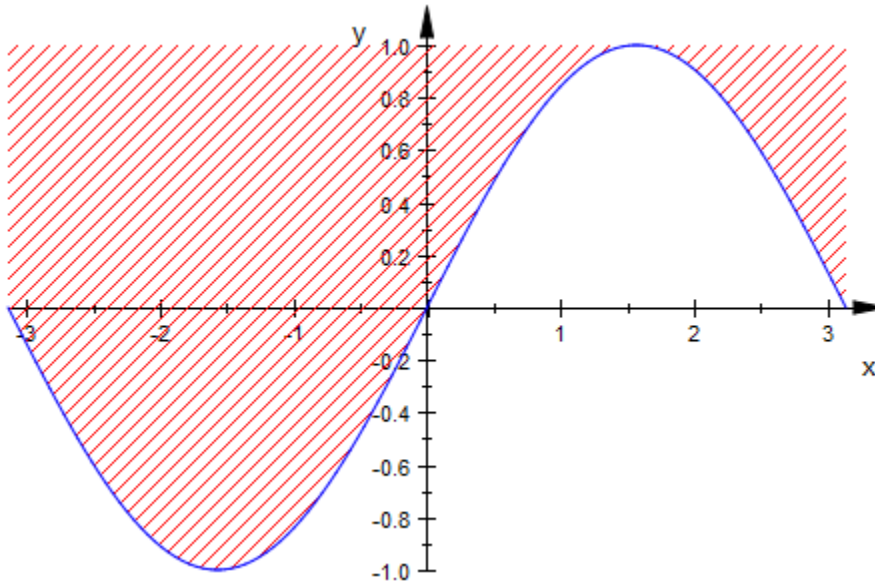
Note that `plot::Hatch` requires an *object* of type `plot::Function2d`, not just a function expression:

```
plot::Hatch(sin(x))
```

```
Error: No 'plot::Function2d' or 'plot::Curve2d' is given. [plot::Hatch::new]
```

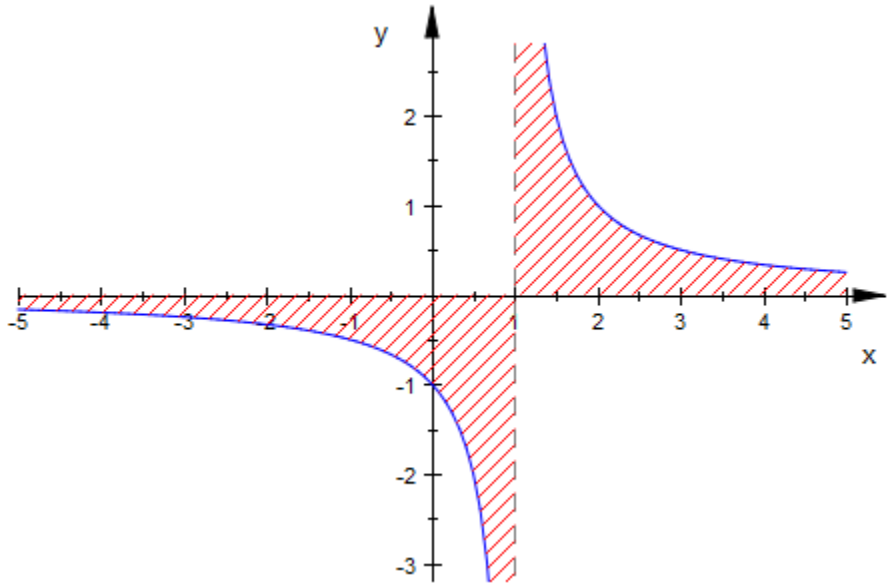
`plot::Hatch` can be asked to hatch the area between a function graph and some constant value (i.e., some line parallel to the x-axis):

```
plot(plot::Hatch(f, 1), f)
```



For functions with poles, keeping `VerticalAsymptotesVisible` set to `TRUE` is highly recommended:

```
f := plot::Function2d(1/(x - 1)):
h := plot::Hatch(f):
plot(f, h)
```

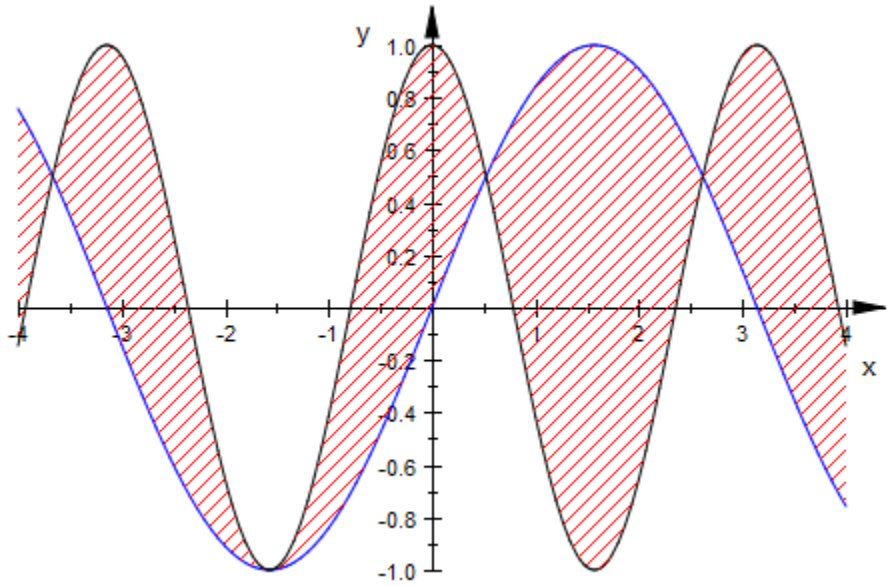



```
delete f, h:
```

Example 2

By passing two functions to `plot::Hatch`, we ask for a hatch of the area between the two:

```
f := plot::Function2d(sin(x), x = -4 .. 4,  Color = RGB::Blue):
g := plot::Function2d(cos(2*x), x = -4 .. 4, Color=RGB::Black):
h := plot::Hatch(f, g):
plot(f, g, h)
```

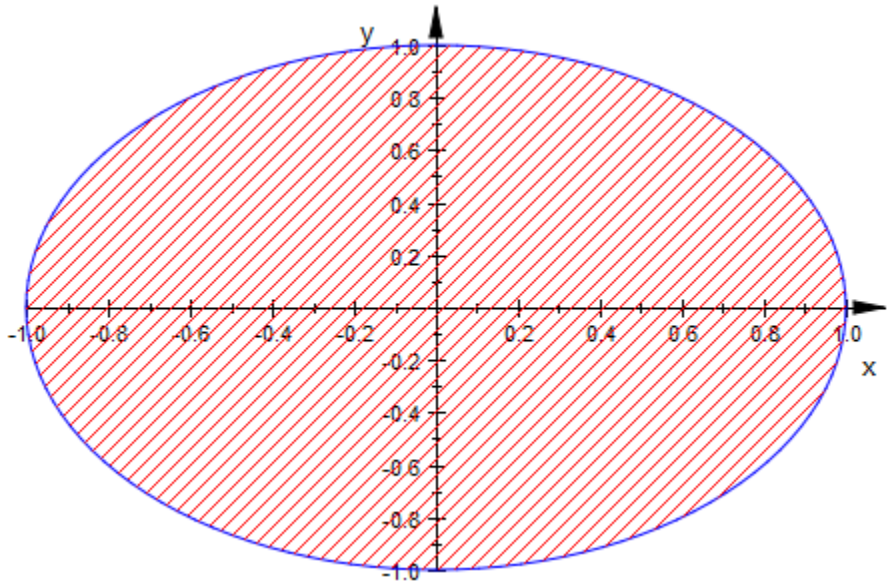



```
delete f, g, h:
```

Example 3

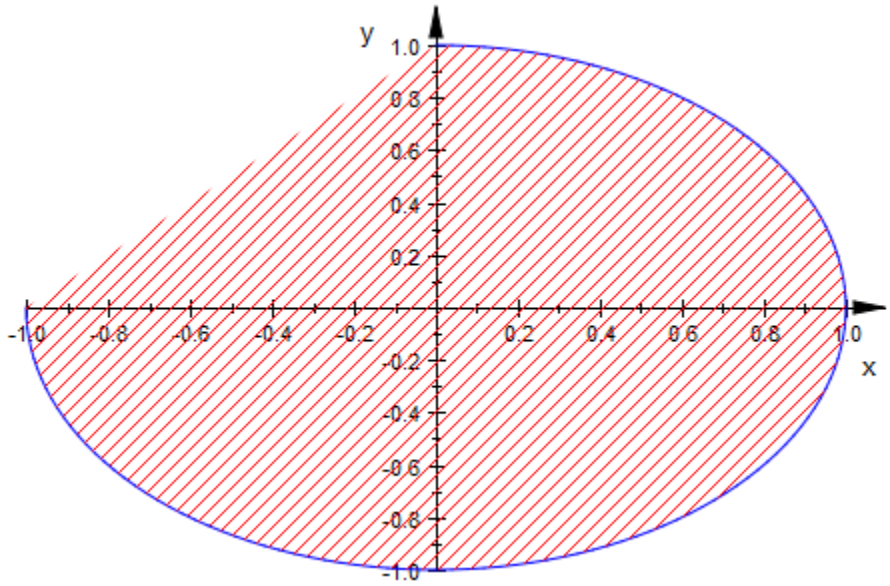
`plot::Hatch` can also hatch the inner part of a `plot::Curve2d` object:

```
circle := plot::Curve2d([sin(t), cos(t)], t=0..2*PI):
plot(circle, plot::Hatch(circle))
```



If the curve is not closed, `plot::Hatch` regards the first and last point to be connected:

```
circle::UMax := 3*PI/2:
plot(circle, plot::Hatch(circle))
```

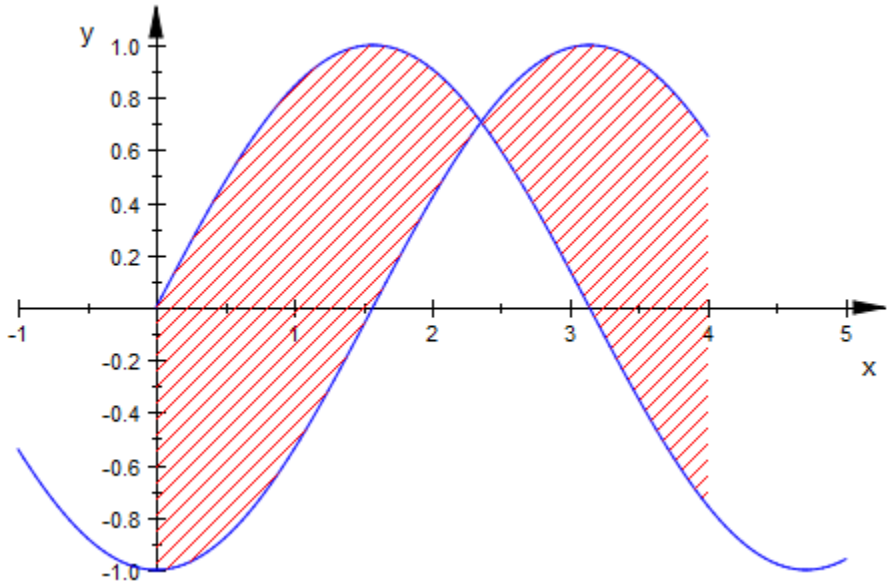



```
delete circle:
```

Example 4

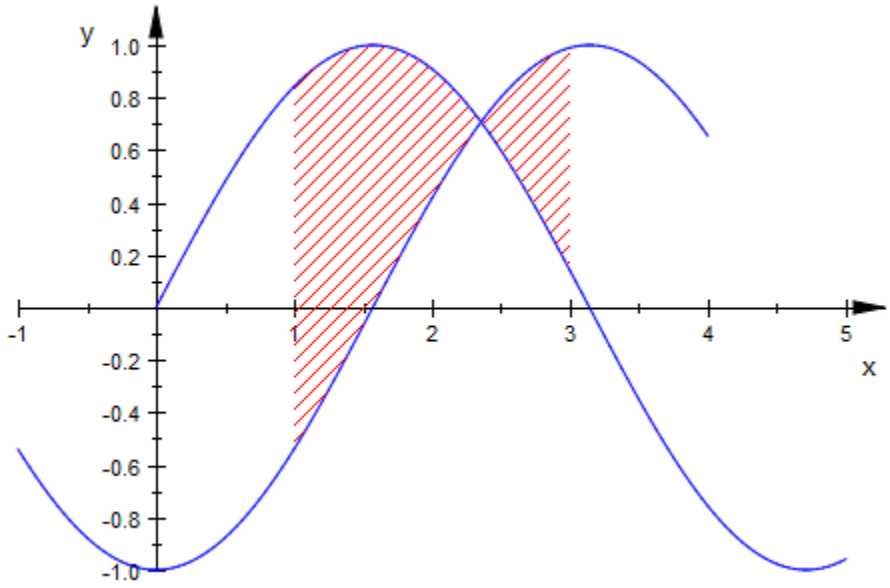
By default, `plot::Hatch` extends as far to the left and right as possible without leaving the common definition area of all given functions:

```
f := plot::Function2d(sin(x), x = 0 .. 5):
g := plot::Function2d(-cos(x), x = -1 .. 4):
h := plot::Hatch(f, g):
plot(f, g, h)
```



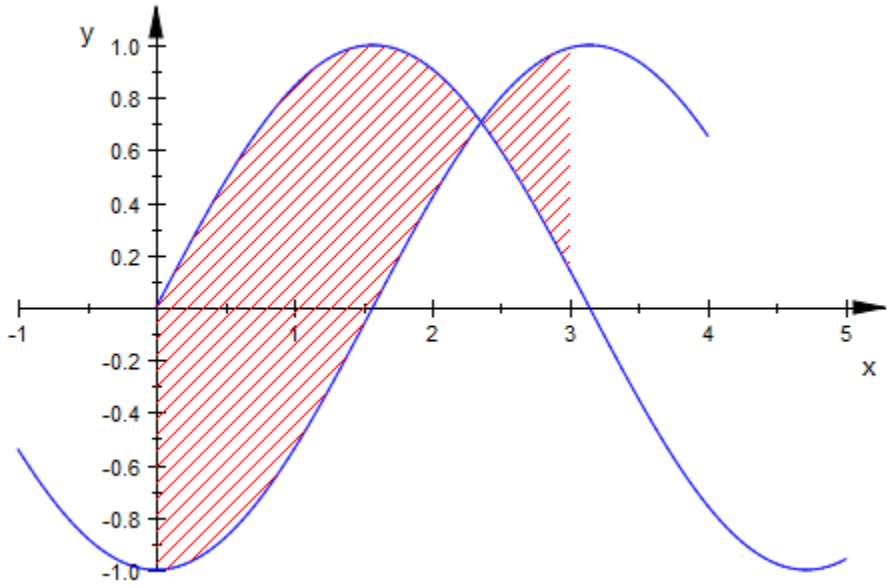
You can restrict this range by giving an explicit range of x values:

```
h := plot::Hatch(f, g, 1 .. 3):
plot(f, g, h)
```



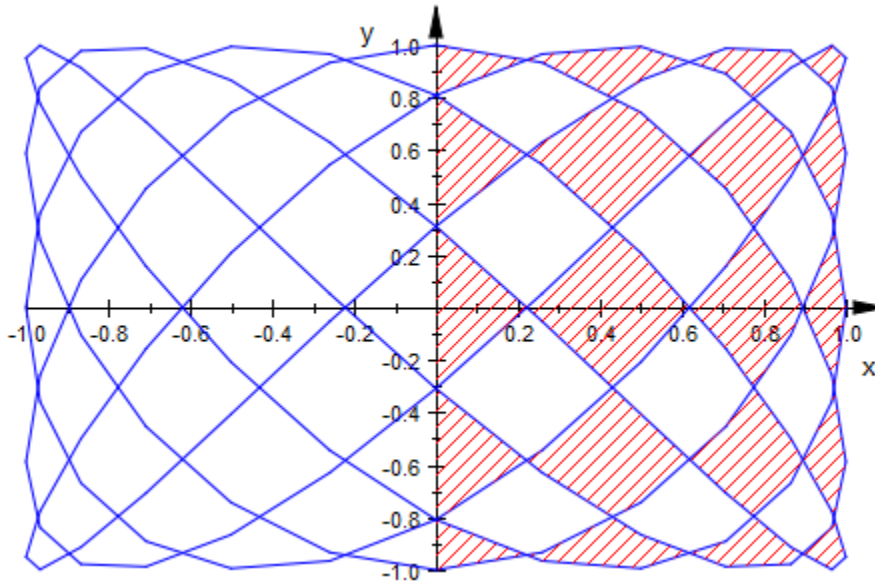
However, it is *not* possible to extend the range beyond the common definition range of both functions:

```
h := plot::Hatch(f, g, -1 .. 3):
plot(f, g, h)
```



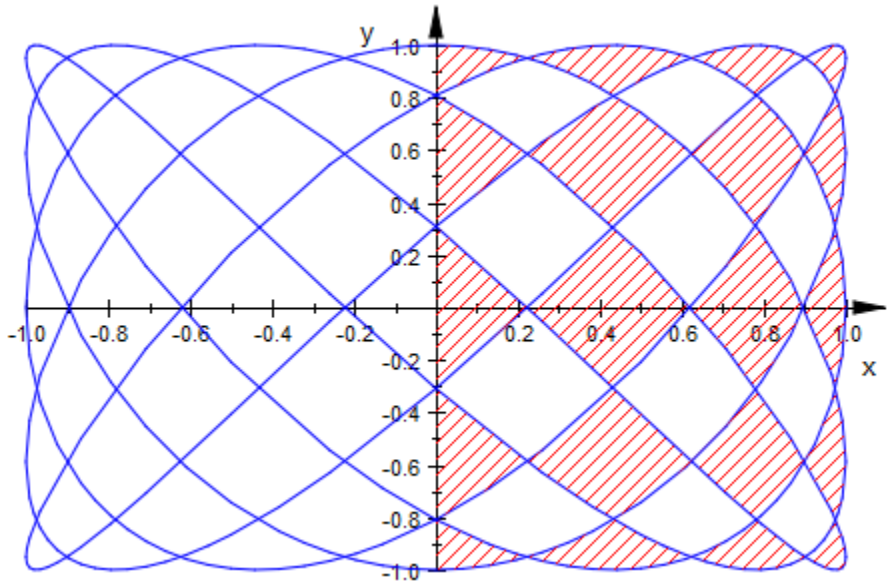
The restriction of the *x* range also works for hatching curve objects:

```
c := plot::Curve2d([sin(5*x), cos(7*x)], x = 0 .. 2*PI):
h := plot::Hatch(c, 0 .. 1):
plot(c, h)
```



Note that `plot::Hatch` reacts to the smoothness of the curve. This is one of the reasons why you have to provide a *objects* instead of expressions for the functions or curves:

```
c::AdaptiveMesh := 2:
plot(c, h)
```

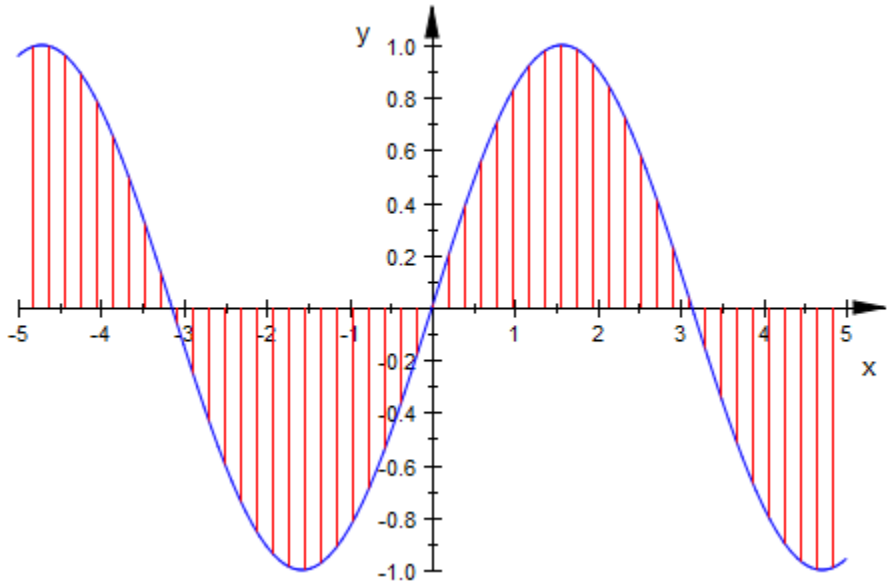



```
delete f, g, h, c:
```

Example 5

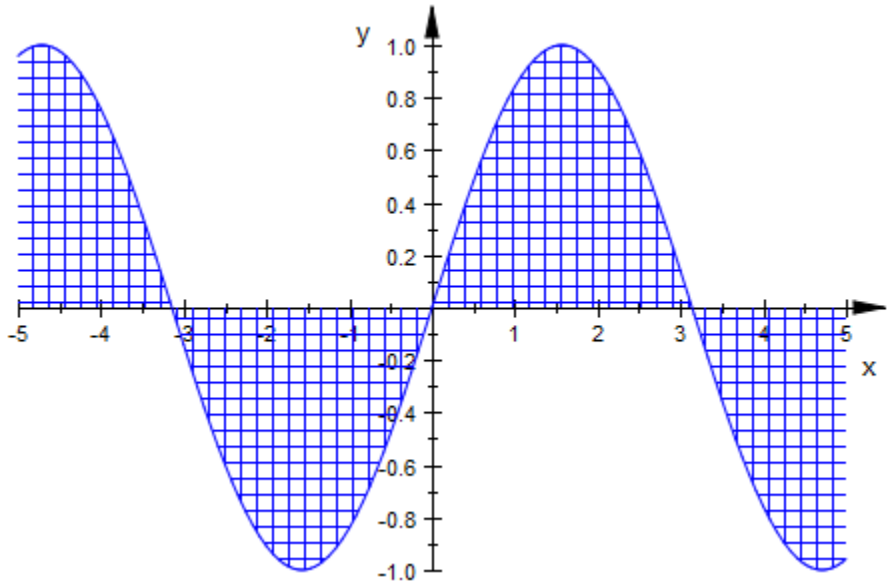
One of the most useful attributes of `plot::Hatch` is `FillPattern`, which can take one of the values `DiagonalLines` (the default), `FDiagonalLines`, `HorizontalLines`, `VerticalLines`, `CrossedLines`, `XCrossedLines`, or `Solid`:

```
f := plot::Function2d(sin(x)):
h := plot::Hatch(f, FillPattern = VerticalLines):
plot(f, h)
```



Another attribute that will often be useful is `FillColor`, to change the color of the hatch. We set the value right in our existing hatch object:

```
h::FillPattern := CrossedLines:
h::FillColor := RGB::Blue:
plot(f, h)
```

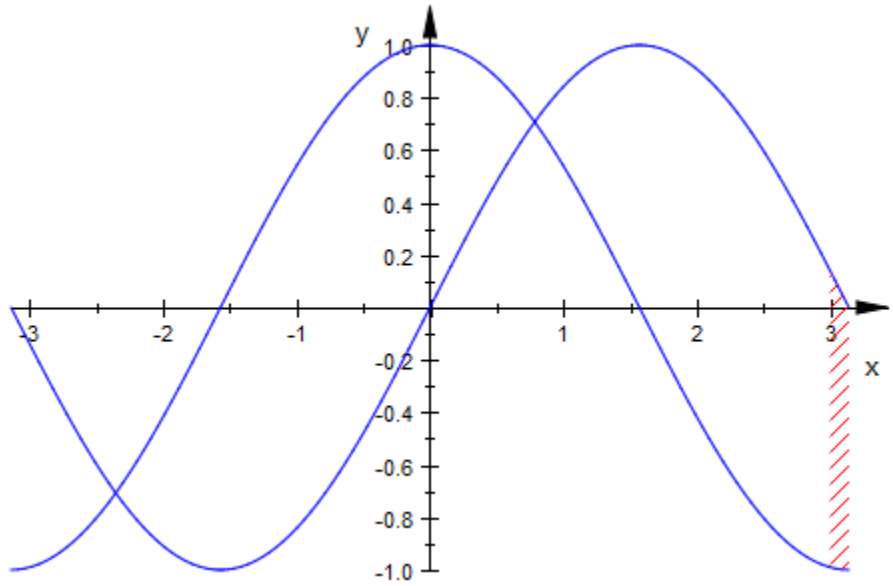



```
delete f, h:
```

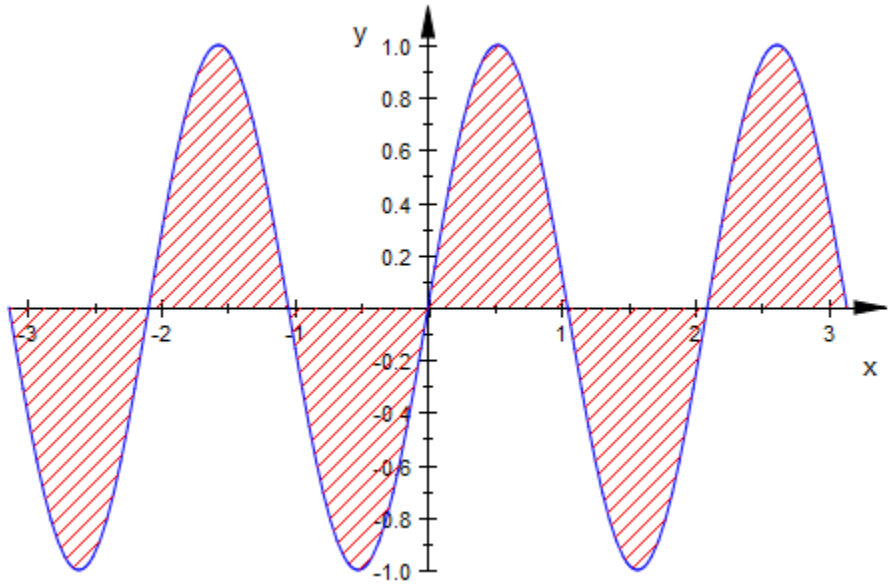
Example 6

The function(s) or curve, the baseline, and the restriction of the *x* range can be animated:

```
f := plot::Function2d(sin(x + a), x = -PI..PI, a = 0..2*PI):
g := plot::Function2d(cos(x - a), x = -PI..PI, a = 0..4*PI):
plot(f, g, plot::Hatch(f, g, x0 .. x0+1, x0 = -PI..3))
```




```
f := plot::Function2d(sin(a*x), x=-PI..PI, a=0.2..3):
plot(f, plot::Hatch(f))
```




```
delete f, g:
```

Example 7

A “hatch” may also be a solid area fill:

```
plot(plot::Hatch(
  plot::Curve2d([abs(r)*sin(r), abs(r)*cos(r)], r = -PI..PI),
  FillPattern = Solid, FillColor = RGB::Red, Axes = None))
```

Parameters

`f1, f2`

The outlining function(s) of the hatch: objects of type `plot::Function2d`.

f_1, f_2 are equivalent to the attributes `Function1`, `Function2`.

`c`

The outlining curve of the hatch: a parametrized curve of type `plot::Curve2d`.

c is equivalent to the attribute `Function1`.

`base`

The base line of the hatch: a numerical real value or an arithmetical expression of the animation parameter `a`.

base is equivalent to the attribute `Baseline`.

x_1 .. x_2

A range on the x-axis limiting the hatch to the left and the right hand side:numerical real values or arithmetical expressions of the animation paramater `a`.

x_1 .. x_2 is equivalent to the attributes `XMin`, `XMax`, `XRange`.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Curve2d` | `plot::Function2d` | `plot::Sweep`

plot::Histogram2d

Histogram plots of data

Syntax

`plot::Histogram2d(data, <a = a_min .. a_max>, options)`

Description

`plot::Histogram2d` creates a histogram plot of the given data, showing the frequency distribution in a user-definable cell array.

By default, data is grouped into 7 classes of equal width. To increase the number of cells, but still have them be of equal width, set `Cells = [n]`, as in "Example 1" on page 24-407. For full control over the classes, set `Cells` to a list specifying the cells, as in "Example 2" on page 24-410.

As long as the attribute `Area` is not changed from its default value of 0, `plot::Histogram2d` displays the absolute number of data in a class as the height of the corresponding bar. With `Area = a`, `a > 0`, the whole plot will take area `a`, with each rectangle area proportional to the number of data points in its cell. "Example 3" on page 24-412 shows the difference in detail.

By default, cells ("classes") given by the attribute `Cells` = `[a_1 .. b_1, a_2 .. b_2, dots]` are interpreted as a collection of semi-open intervals $(a_i, b_i]$ that are closed at the right boundary. A data item `x` is tallied into the i-th cell if it satisfies $a_i < x \le b_i$. Use the option `CellsClosed` = `Left` or the equivalent `ClassesClosed` = `Left` to interpret the classes as the semi-open intervals $[a_i, b_i)$ that are closed at the left boundary.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`

Attribute	Purpose	Default Value
AntiAliased	antialiased lines and points?	TRUE
Area	the area of a histogram plot	0
Cells	classes of histogram plots	[7]
CellsClosed	interpretation of the classes in histogram plots	Right
ClassesClosed	interpretation of the classes in histogram plots	[Right]
Color	the main color	RGB::GeraniumLake
Data	the (statistical) data to plot	
DrawMode	orientation of boxes and bars	Vertical
Filled	filled or transparent areas and surfaces	TRUE
FillColor	color of areas and surfaces	RGB::GeraniumLake
FillPattern	type of area filling	Solid
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Black
LineWidth	width of lines	0.35
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
Name	the name of a plot object (for browser and legend)	

Attribute	Purpose	Default Value
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	

Attribute	Purpose	Default Value
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

To plot a histogram of a normally distributed process, we first create a generator of random numbers with this distribution:

```
X := stats::normalRandom(0, 1)
```

```
proc X() ... end
```

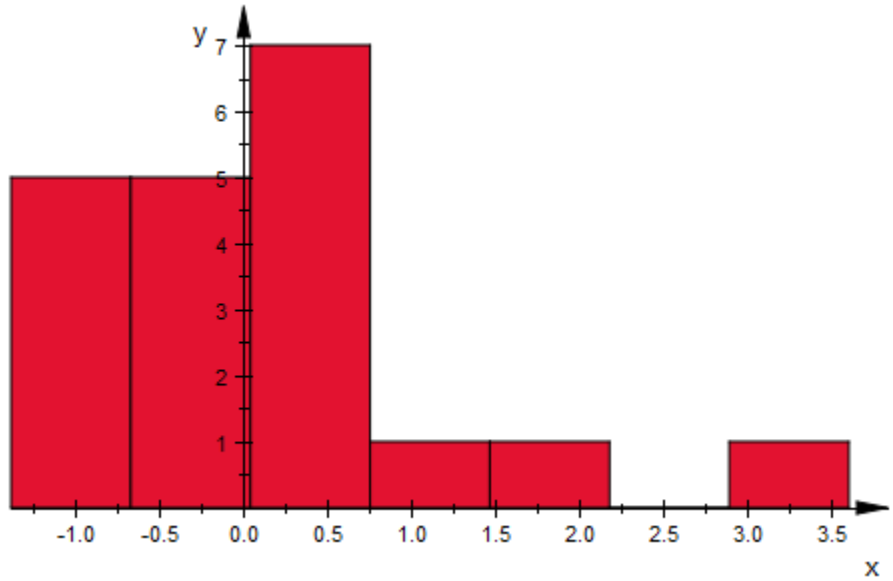
Next, we create a small number of "measurements":

```
data := [X() $ i = 1..20]
```

```
[-0.5297400457, -0.5694234147, -0.5161446272, -1.090814471, 1.782520584, 0.6370330472,
  0.6902341601, 0.3399758858, 1.177699186, -0.5970692982, -1.386247581, -0.9783222199,
  -0.7891413081, 0.2090732178, 0.2186783746, -0.7392138209, 0.6496128588, 0.6258699055,
  3.606896706, -0.3319378999]
```

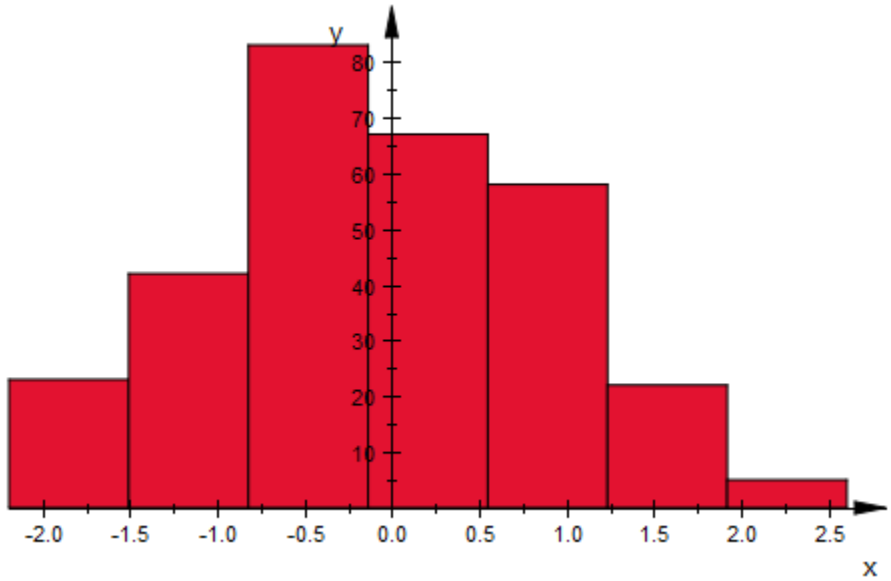
This data is ready to be put into `plot::Histogram2d`:

```
plot(plot::Histogram2d(data))
```



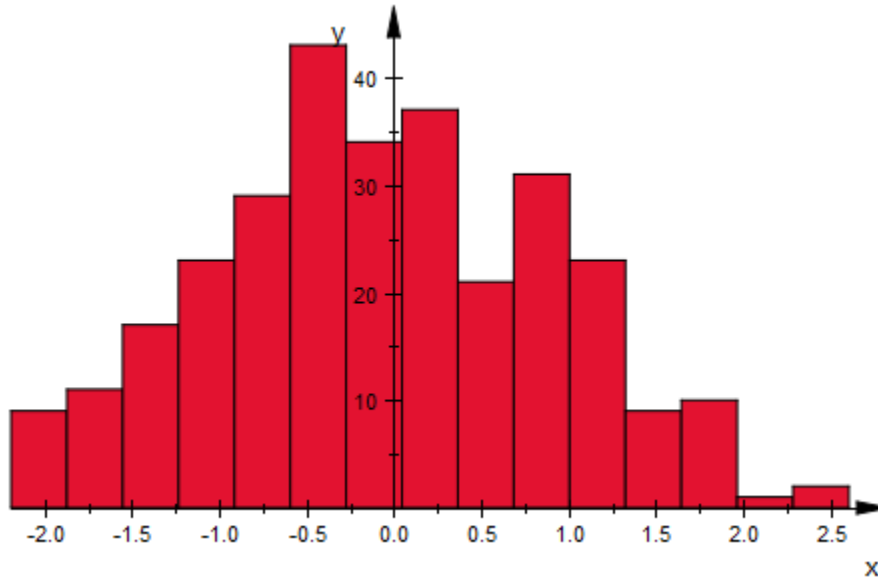
This plot, if nothing else, shows that 20 samples are very few. Let us repeat the process with more data:

```
data := [X() $ i = 1..300]:
plot(plot::Histogram2d(data))
```



On the other hand, this amount of data certainly justifies a finer classification:

```
plot(plot::Histogram2d(data, Cells = [15]))
```



Example 2

It is also possible to give the cells (classes) directly. To do so, you should give them as ranges or lists with two elements, as in the following example:

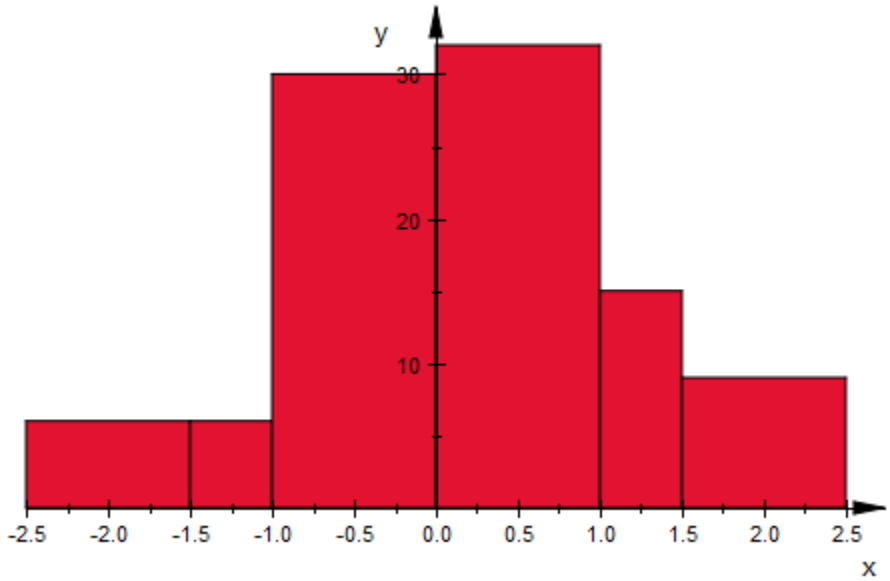
```
X := stats::normalRandom(0, 1):
data := [X() $ i = 1 .. 100]:
min(data), max(data)
```

$$-3.266420216,\ 2.409775834$$

```
h := plot::Histogram2d(data,
            Cells = [-2.5..-1.5, -1.5..-1, -1..0,
                     0..1, 1..1.5, 1.5..2.5])
```

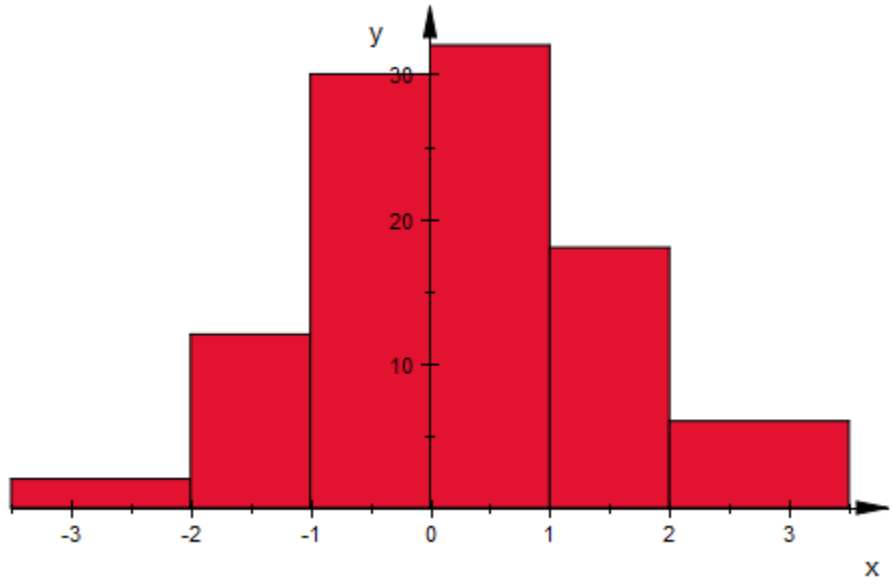
plot::Histogram2d(...)

```
plot(h)
```



It is even possible to use `-infinity` and `infinity` as border values in the cells:

```
h::Cells := [-infinity..-2, -2..-1, -1..0,
             0..1, 1..2, 2..infinity]:
plot(h)
```



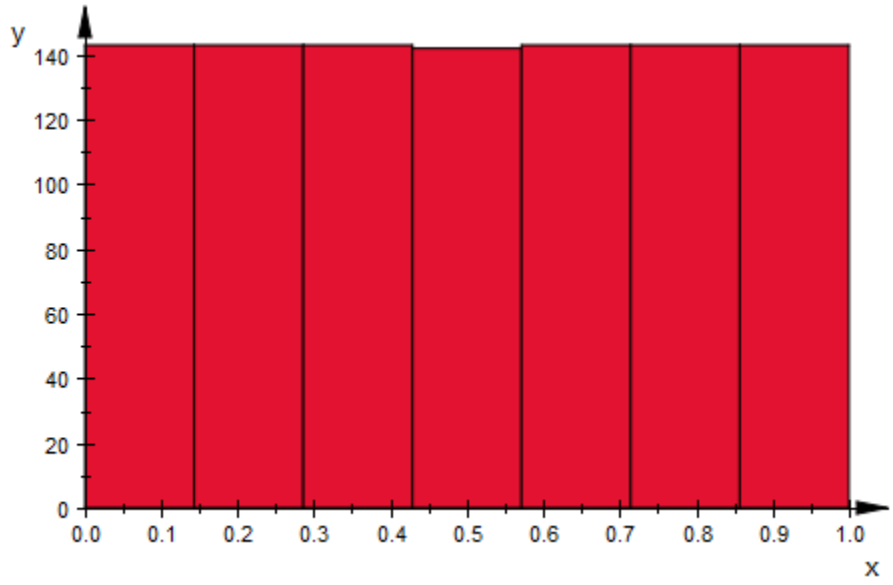
Example 3

One potential problem with classes of non-equal width is that still the *height* of the bars corresponds to the *number* of data points in a class. To see why this may be a problem, consider data perfectly uniformly distributed:

```
data := [i/1000 $ i = 1..1000]:
```

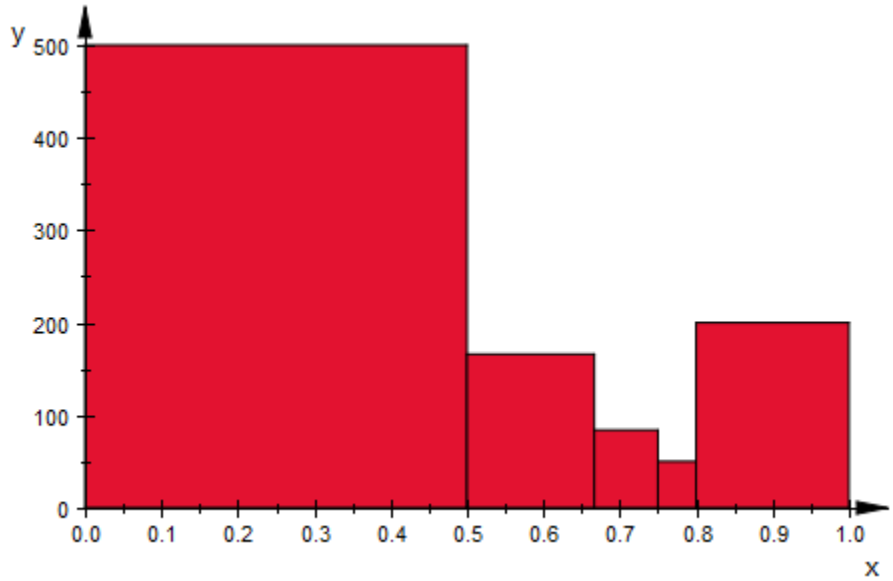
Plotting a histogram of this data, we see only very small deviations from a rectangle, caused by the fact that 1000 and 7 are coprime:

```
plot(plot::Histogram2d(data))
```



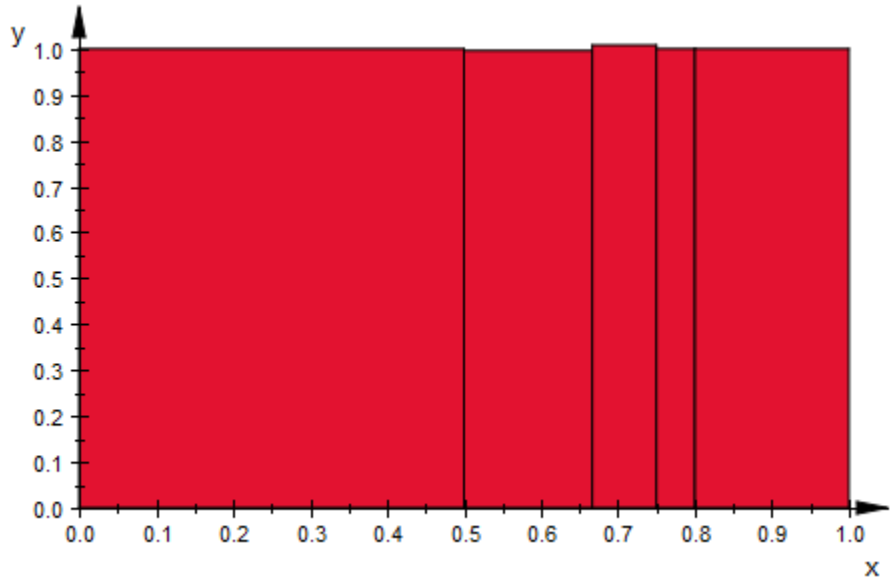
However, plotting a histogram with uneven classes, the image looks very much different:

```
plot(plot::Histogram2d(data,
       Cells = [0..1/2, 1/2..2/3, 2/3..3/4, 3/4..4/5, 4/5..1]))
```



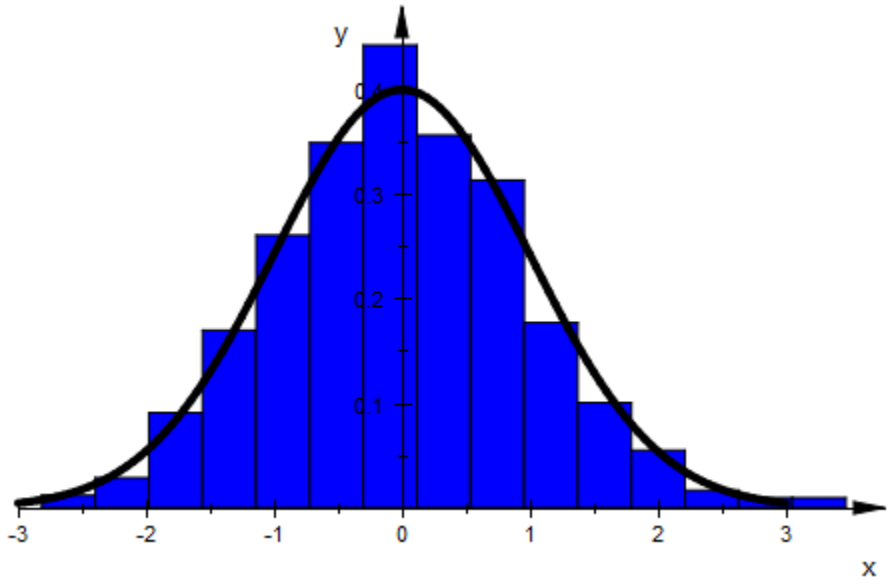
To make not the *height*, but rather the *area* of a bar depend on the number of samples in a class, set `Area` to a positive value:

```
plot(plot::Histogram2d(data,
       Cells = [0..1/2, 1/2..2/3, 2/3..3/4, 3/4..4/5, 4/5..1],
       Area = 1))
```



Note that with `Area = 1`, a histogram plot is scaled accordingly to the probability density function of the variable displayed:

```
X := stats::normalRandom(0, 1):
data := [X() $ i = 1..1000]:
h := plot::Histogram2d(data, Cells = [15],
                       Area = 1, Color = RGB::Blue):
f := plot::Function2d(stats::normalPDF(0, 1),
                      x = -3..3, LineWidth = 1*unit::mm,
                      Color = RGB::Black):
plot(h, f)
```




```
delete X, data, h, f:
```

Parameters

data

The data to plot: A list of real values or expressions in the animation parameter `a`.

data is equivalent to the attribute `Data`.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions

`plot` | `plot::copy` | `stats::frequency`

MuPAD Graphical Primitives

`plot::Bars2d` | `plot::Boxplot` | `plot::Scatterplot`

plot::Implicit2d

Contour lines of a function from R^2 to R

Syntax

`plot::Implicit2d(f, x = `x_{min}` .. `x_{max}`, y = `y_{min}` .. `y_{max}`, <a = `a_{min}` .. `a_{max}`>, options)`

Description

`plot::Implicit2d(f(x, y), x = `x_{min}`..`x_{max}` , y = `y_{min}`..`y_{max}` )` plots the curves where the smooth function `f` is zero.

`plot::Implicit2d(f, x = `x_{min}`..`x_{max}` , y = `y_{min}`..`y_{max}` )` plots the zeroes of *f* in the given range, i.e., the set $\{(x, y) \mid x_{min} \le x \le x_{max}, y_{min} \le y \le y_{max}, f(x, y) = 0\}$.

`plot::Implicit2d` assumes that *f* is *regular almost everywhere* on this curve, which means that *f* must be differentiable and at least one of its partial derivatives must be nonzero.

To plot other contours than zeroes, use the option `Contours`.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`
`Color`	the main color	`RGB::Blue`
`Contours`	the contours of an implicit function	[0]
`Frames`	the number of frames in an animation	50
`Function`	function expression or procedure	

Attribute	Purpose	Default Value
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`TRUE`
`LineColor`	color of lines	`RGB::Blue`
`LineWidth`	width of lines	`0.35`
`LineColor2`	color of lines	`RGB::DeepPink`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LinesVisible`	visibility of lines	`TRUE`
`LineColorType`	line coloring types	`Flat`
`LineColorFunction`	functional line coloring	
`LineColorDirection`	the direction of color transitions on lines	[0, 1]
`LineColorDirectionX`	x-component of the direction of color transitions on lines	0
`LineColorDirectionY`	y-component of the direction of color transitions on lines	1
`Mesh`	number of sample points	[11, 11]
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`TimeEnd`	end time of the animation	`10.0`

Attribute	Purpose	Default Value
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XMax	final value of parameter “x”	
XMesh	number of sample points for parameter “x”	11
XMin	initial value of parameter “x”	
XName	name of parameter “x”	
XRange	range of parameter “x”	
YMax	final value of parameter “y”	

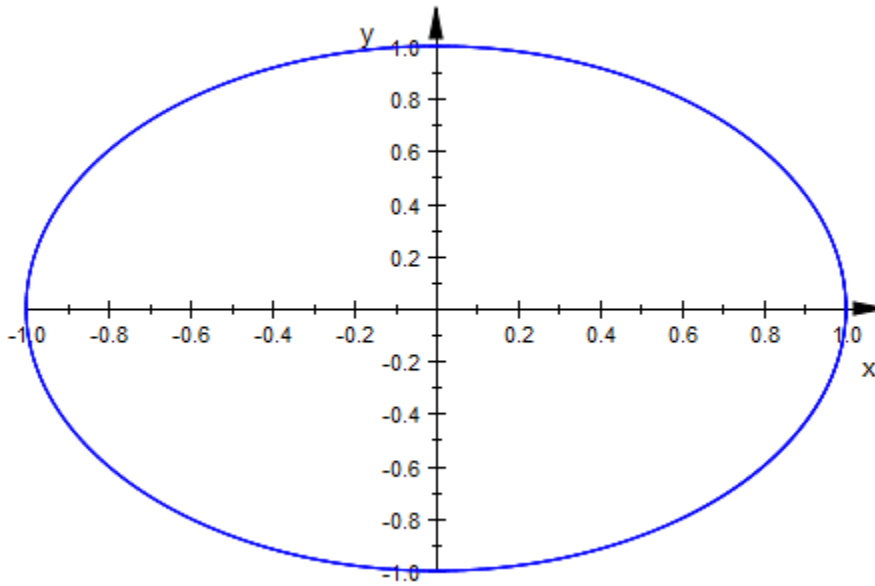
Attribute	Purpose	Default Value
YMesh	number of sample points for parameter “y”	11
YMin	initial value of parameter “y”	
YName	name of parameter “y”	
YRange	range of parameter “y”	

Examples

Example 1

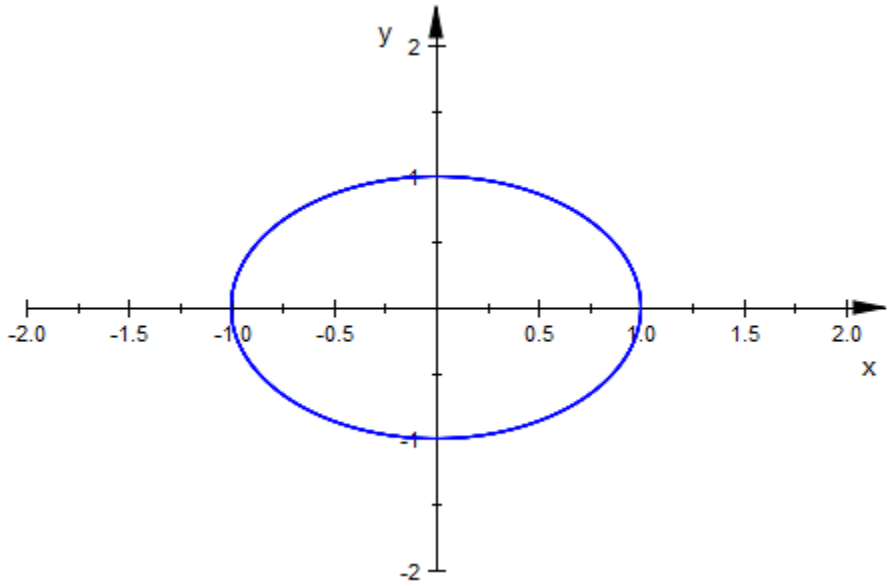
It is well-known that a circle can be described as $\{(x, y) \mid x^2 + y^2 = r^2\}$:

```
plot(plot::Implicit2d(x^2+y^2-1, x = -1..1, y = -1..1))
```



Note that `plot::Implicit2d` uses the given range completely, even if there is nothing to plot at a border:

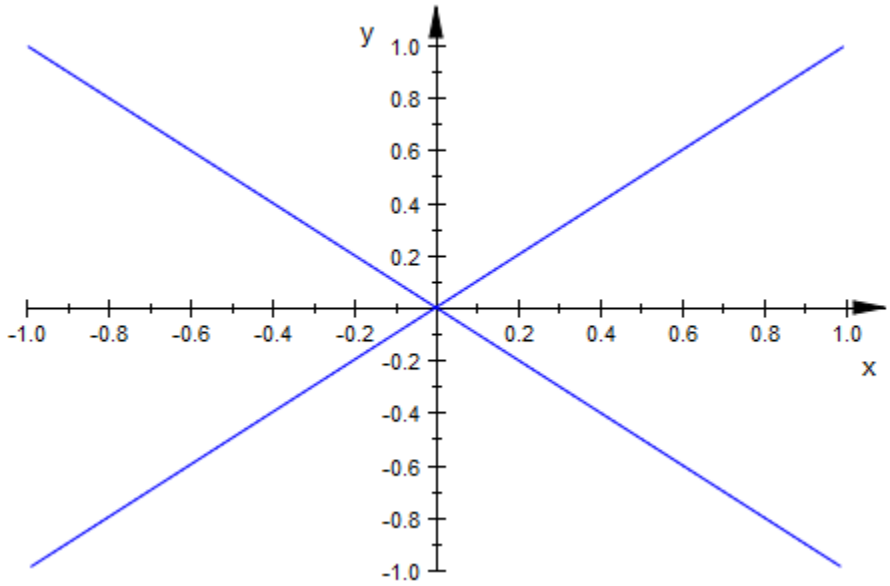
```
plot(plot::Implicit2d(x^2+y^2-1, x = -2..2, y = -2..2))
```



Example 2

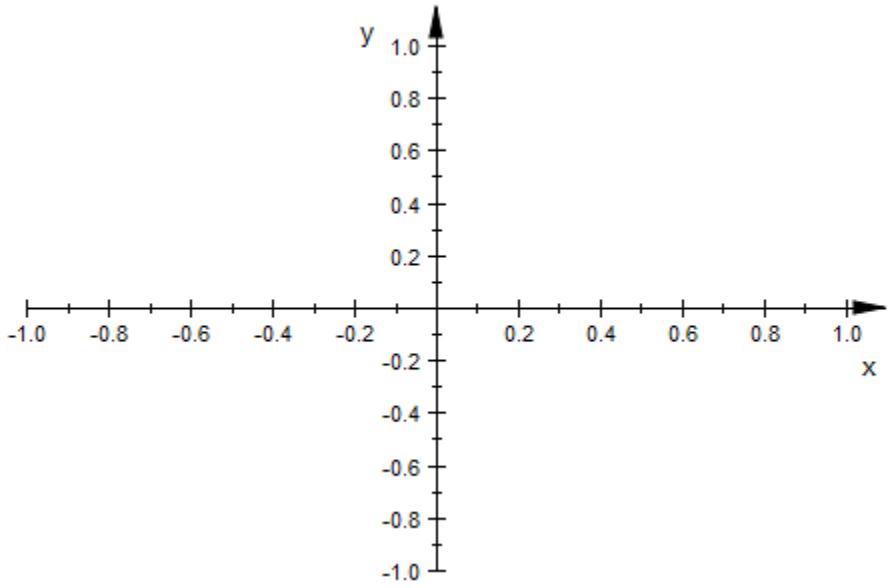
`plot::Implicit2d` handles functions which are not regular at isolated points on the contours:

```
plot(plot::Implicit2d((x-y)*(x+y), x = -1..1, y = -1..1))
```



However, it fails if the function is singular on more than isolated points:

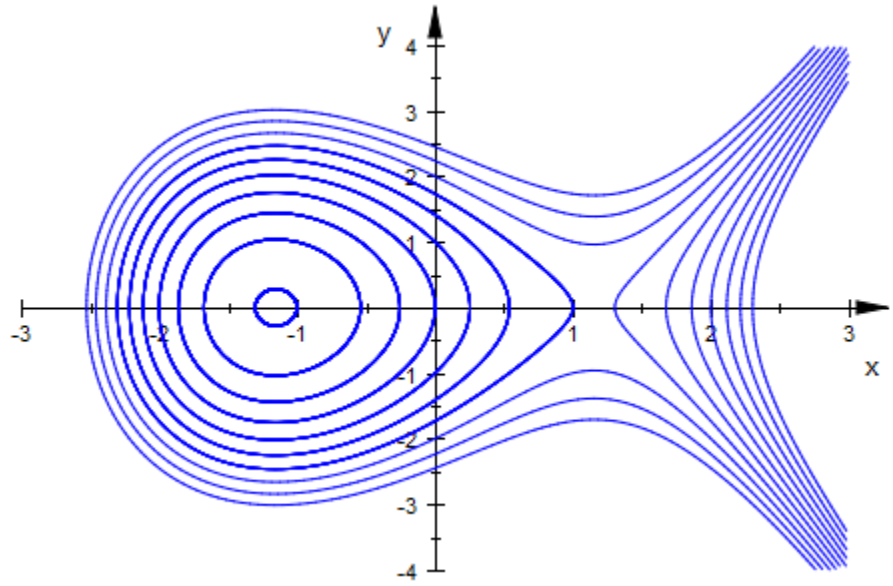
```
plot(plot::Implicit2d(0, x = -1..1, y = -1..1))
```



Example 3

We plot some of the elliptic curves $y^2 = x^3 + 4\,x + c$:

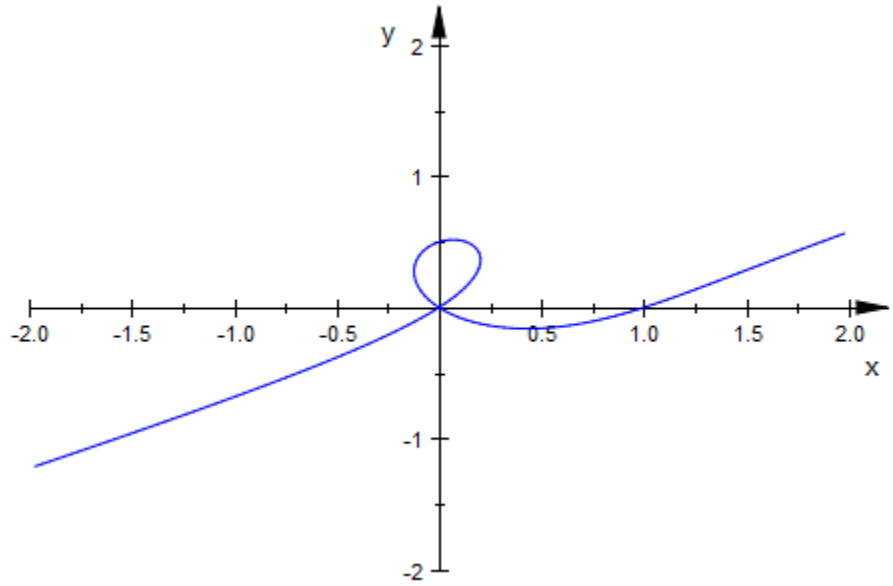
```
plot(plot::Implicit2d(y^2 - x^3 + 4*x, x = -3..3, y = -4..4,
                      Contours = [c $ c = -3..6]))
```



Example 4

Like most graphical objects, `plot::Implicit2d` can be animated easily:

```
plot(plot::Implicit2d(x^2 - y^2 = (x - a*y)*(x^2 + y^2),
                      x = -2..2, y = -2..2, a = -2..2))
```



Parameters

`f`

A real-valued expression or an equation in `x`, `y`, and possibly the animation parameter.

f is equivalent to the attribute `Function`.

`x, y`

identifiers.

x, y are equivalent to the attributes `XName`, `YName`.

x_{min} `..` x_{max}, y_{min} `..` y_{max}

Real-valued expressions, possibly in the animation parameter. The image is plotted with `x` in the range $x_{min} \le x \le x_{max}$ and $y_{min} \le y \le y_{max}$.

x_{min} .. x_{max}, y_{min} .. y_{max} are equivalent to the attributes `XRange`, `XMin`, `XMax`, `YRange`, `YMin`, `YMax`.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

Algorithms

`plot::Implicit2d` uses a curve tracking method: It first generates starting points on the curve and then uses a predictor-corrector method to follow the curve thus found in both directions, using the implicit function theorem.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Curve2d` | `plot::Implicit3d`

plot::Implicit3d

Contour surfaces of a function from R^3 to R

Syntax

```
plot::Implicit3d(f, x = xmin .. xmax, y = ymin .. ymax, z = zmin .. zmax, <a = amin .. amax>, c
```

Description

`plot::Implicit3d(f(x, y, z), x = xmin..xmax , y = ymin..ymax , z = zmin..zmax )` plots the surfaces where the smooth function `f` is zero.

`plot::Implicit3d(f, x = xmin..xmax , y = ymin..ymax , z = zmin..zmax )` plots the (two-dimensional part of the) zeroes of *f* in the given range, i.e., the set $\{(x, y, z) \mid x_{min} \le x \le x_{max}, y_{min} \le y \le y_{max}, z_{min} \le z \le z_{max}, f(x, y, z) = 0\}$.

`plot::Implicit3d` assumes that *f* is *regular almost everywhere* on this surface, which means that *f* must be differentiable and at least two of its partial derivatives must be nonzero.

`plot::Implicit3d` evaluates the given function on an equidistant, three-dimensional mesh, the coarsity of which can be set with the attributes `XMesh`, `YMesh`, and `ZMesh` for each of the three directions, or with the combining attribute `Mesh` that sets all three of these simultaneously.

After finding an initial triangulation of the surface from the numerical data on the initial grid, `plot::Implicit3d` optionally performs adaptive subdivision of the triangles. To make a long story short: If the initial calculation misses details altogether, adaptive refinement will not find them either. On the other hand, if the initial calculation shows spurious spikes, adaptive refinement will result in a much more realistic image, at the expense of time; the higher the value of `AdaptiveMesh`, the more. Increasing `AdaptiveMesh` by one may in extreme cases increase calculation time by a factor of eight or more!

The details of the algorithm are as follows: On top level, the “effective adaptive level” is set to the value of the attribute `AdaptiveMesh`. If, for a given edge, the effective

adaptive level is positive and the edge is not very short already, compared with the size of the complete image, and inserting a new point on the implicit surface near the middle of this edge would cause the two new edges to have an angle of less than 170 degrees, then the edge is split, the adjoining triangles are split accordingly (taking into account all their edges) and all the new edges caused by this operation are examined with an effective adaptive level reduced by one.

To plot other contours than zeroes, use the option `Contours`.

Attributes

Attribute	Purpose	Default Value
`AdaptiveMesh`	adaptive sampling	`0`
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Color`	the main color	`RGB::Red`
`Contours`	the contours of an implicit function	[0]
`Filled`	filled or transparent areas and surfaces	`TRUE`
`FillColor`	color of areas and surfaces	`RGB::Red`
`FillColor2`	second color of areas and surfaces for color blends	`RGB::CornflowerBlue`
`FillColorType`	surface filling types	`Dichromatic`
`FillColorFunction`	functional area/surface coloring	
`FillColorDirection`	the direction of color transitions on surfaces	[0, 0, 1]
`FillColorDirectionX`	x-component of the direction of color transitions on surfaces	`0`
`FillColorDirectionY`	y-component of the direction of color transitions on surfaces	`0`

Attribute	Purpose	Default Value
FillColorDirectionZ	z-component of the direction of color transitions on surfaces	1
Frames	the number of frames in an animation	50
Function	function expression or procedure	
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	TRUE
LineColor	color of lines	RGB::Black.[0.15]
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	0
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Mesh	number of sample points	[11, 11, 11]
MeshVisible	visibility of irregular mesh lines in 3D	FALSE

Attribute	Purpose	Default Value
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`PointStyle`	the presentation style of points	`FilledCircles`
`PointsVisible`	visibility of mesh points	`FALSE`
`Shading`	smooth color blend of surfaces	`Smooth`
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`
`Title`	object title	
`TitleFont`	font of object titles	`[" sans-serif ", 11]`
`TitlePosition`	position of object titles	
`TitleAlignment`	horizontal alignment of titles w.r.t. their coordinates	`Center`
`TitlePositionX`	position of object titles, x component	
`TitlePositionY`	position of object titles, y component	
`TitlePositionZ`	position of object titles, z component	
`Visible`	visibility	`TRUE`

Attribute	Purpose	Default Value
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XContours	contour lines at constant x values	[Automatic, 15]
XMax	final value of parameter “x”	
XMesh	number of sample points for parameter “x”	11
XMin	initial value of parameter “x”	
XName	name of parameter “x”	
XRange	range of parameter “x”	
YContours	contour lines at constant y values	[Automatic, 15]
YMax	final value of parameter “y”	
YMesh	number of sample points for parameter “y”	11
YMin	initial value of parameter “y”	
YName	name of parameter “y”	
YRange	range of parameter “y”	
ZContours	contour lines at constant z values	[Automatic, 15]
ZMax	final value of parameter “z”	

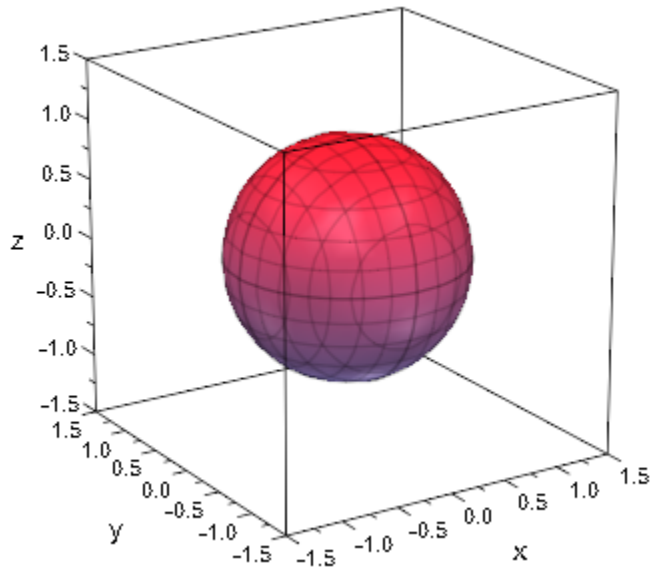
Attribute	Purpose	Default Value
`ZMesh`	number of sample points for parameter “z”	`11`
`ZMin`	initial value of parameter “z”	
`ZName`	name of parameter “z”	
`ZRange`	range of parameter “z”	

Examples

Example 1

The set of x, y, z where $x^2 + y^2 + z^2 = 1$ form a sphere:

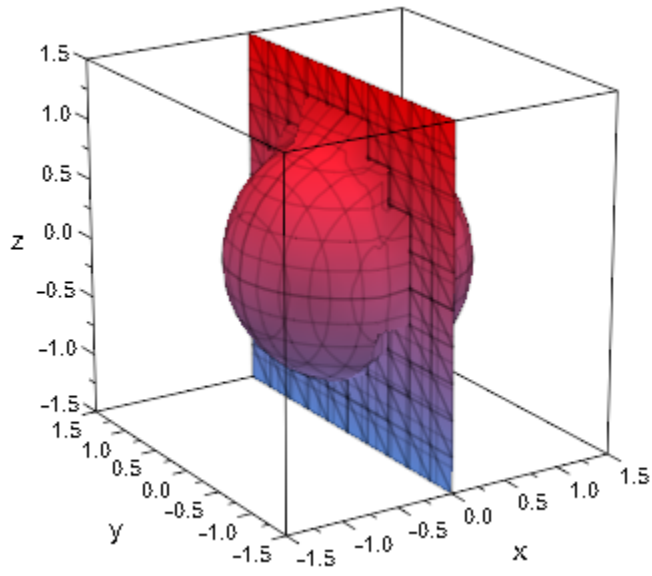
```
plot(plot::Implicit3d(x^2 + y^2 + z^2 - 1,
                      x = -1.5..1.5,
                      y = -1.5..1.5,
                      z = -1.5..1.5),
     Scaling = Constrained)
```



Example 2

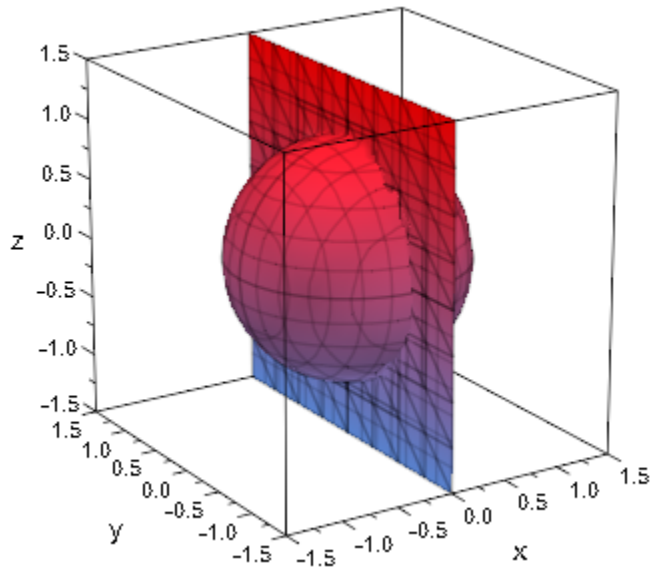
The set of zeroes of a product is the union of the zeroes of the individual functions:

```
plot(plot::Implicit3d((x^2 + y^2 + z^2 - 1) * x,
                      x = -1.5..1.5,
                      y = -1.5..1.5,
                      z = -1.5..1.5),
     Scaling = Constrained)
```



Note that this image is largely dominated by artifacts caused by the coarse evaluation mesh. Increasing this mesh improves the graphics, but increases computation time:

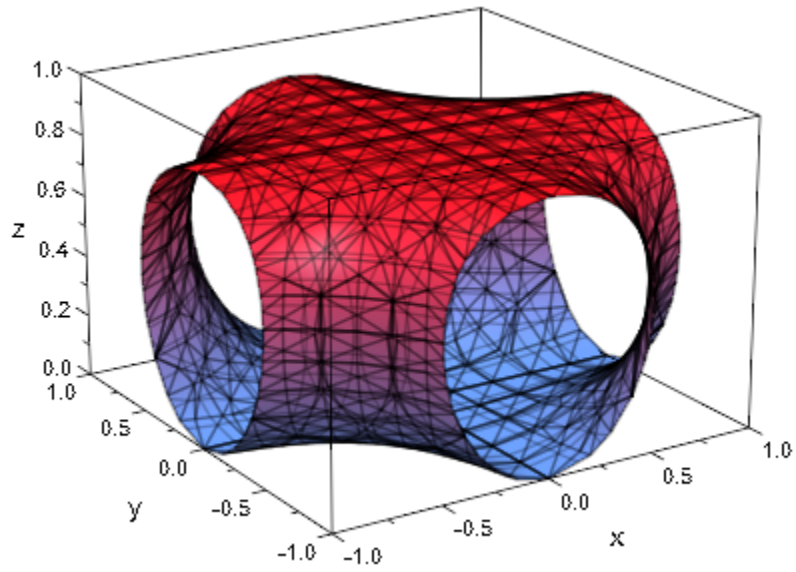
```
plot(plot::Implicit3d((x^2 + y^2 + z^2 - 1) * x,
                      x = -1.5..1.5,
                      y = -1.5..1.5,
                      z = -1.5..1.5,
                      Mesh = [21, 9, 9], AdaptiveMesh = 2),
     Scaling = Constrained)
```



Example 3

With `MeshVisible = TRUE`, the internal triangulation becomes visible:

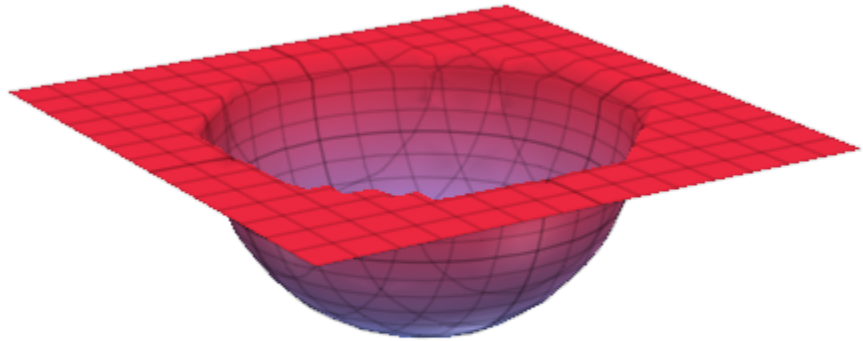
```
plot(plot::Implicit3d(z^2 - sin(z - x^2*y^2) = 0,
                      x = -1 .. 1, y = -1 .. 1, z = 0 .. 1,
                      AdaptiveMesh = 2, MeshVisible = TRUE,
                      LineColor = RGB::Black.[0.25])):
```



Example 4

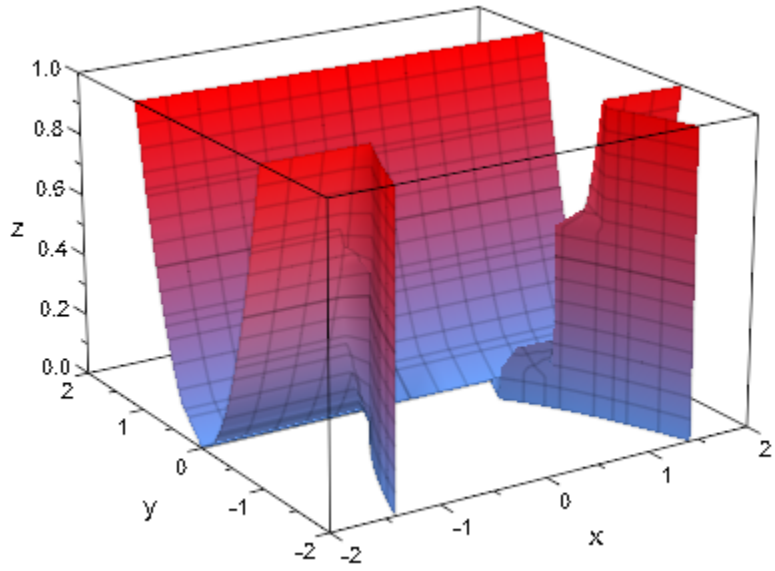
Using functions that are not continuously differentiable, it is possible to generate sharp edges in the images:

```
plot(plot::Implicit3d(min(x^2 + y^2 + z^2 - 2, -z),
                      x = -2..2, y = -2..2, z = -1.5..0.5),
     Axes = None, Scaling = Constrained)
```

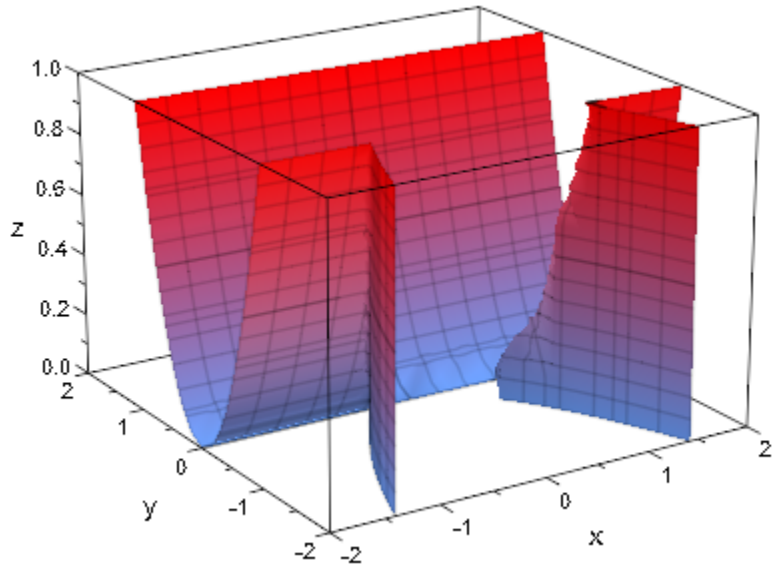


Just like in the preceding example, these sharp corners are prime sources of artifacts, which may require a finer initial mesh and/or adaptive mesh refinement:

```
im := plot::Implicit3d(min(x^2 + y, y^2 - z),
                       x = -2..2, y = -2..2, z = 0..1):
plot(im)
```



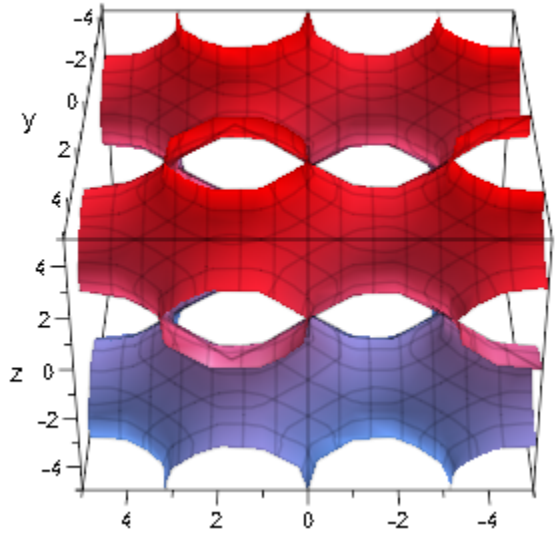

```
plot(im, AdaptiveMesh = 3)
```



Example 5

Animating `plot::Implicit3d` objects takes a lot of time. It is easy and fast, though, to add an animated camera object:

```
plot(plot::Implicit3d(sin(x)+sin(y)+sin(z), x=-5..5, y=-5..5, z=-5..5),
     plot::Camera([42*sin(t),42*cos(t),42*cos(t-sin(t))], [0,0,0],
                  PI/12, t=0..2*PI),
     AnimationStyle=Loop)
```



Parameters

`f`

A real-valued expression or an equation in `x`, `y`, `z`, and possibly the animation parameter.

f is equivalent to the attribute `Function`.

`x, y, z`

identifiers.

x, y, z are equivalent to the attributes `XName`, `YName`, `ZName`.

x_{min} .. x_{max}, y_{min} .. y_{max}, z_{min} .. z_{max}

Real-valued expressions, possibly in the animation parameter. The image is plotted with x in the range $x_{min} \le x \le x_{max}$, $y_{min} \le y \le y_{max}$ and $z_{min} \le z \le z_{max}$.

x_{min} .. x_{max}, y_{min} .. y_{max}, z_{min} .. z_{max} are equivalent to the attributes `XRange`, `XMin`, `XMax`, `YRange`, `YMin`, `YMax`, `ZRange`, `ZMin`, `ZMax`.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Implicit2d` | `plot::Surface`

plot::Inequality

Display areas where inequalities are fulfilled

Syntax

```
plot::Inequality(ineq, x = x_min .. x_max, y = y_min .. y_max, <a = a_min .. a_max>, options)
```

```
plot::Inequality([ineq_1, …], x = x_min .. x_max, y = y_min .. y_max, <a = a_min .. a_max>, options)
```

Description

`plot::Inequality(f(x, y) < g(x, y), x = `x_{min}`..`x_{max}` , y = `y_{min}`..`y_{max}` )` fills the rectangle $x_{min} \le x \le x_{max}$, $y_{min} \le y \le y_{max}$ with several colors, indicating which points satisfy the inequality.

`plot::Inequality` computes a (more or less coarse) rasterization of the area specified by `` `x_{min}`..`x_{max}` `` and `` `y_{min}`..`y_{max}` `` and colors subareas according to whether all of the given inequalities are fulfilled (these are colored in `FillColorTrue`), at least one inequality is nowhere fulfilled in the subarea (`FillColorFalse`) or the granularity is insufficient to decide for either of these cases (`FillColorUnknown`).

You can control the density of the rasterization with the attribute `Mesh`. Cf. “Example 2” on page 24-448.

`plot::Inequality` uses interval numerics for evaluation, so the results are reliable, but certain special functions (such as `hypergeom`) cannot be used because they are not supported for intervals.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`FALSE`
`FillPattern`	type of area filling	`Solid`

Attribute	Purpose	Default Value
FillColorTrue	the color for “true” areas (inequality plot)	RGB::Green
FillColorFalse	the color for “false” areas (inequality plot)	RGB::Red
FillColorUnknown	the color for “unknown” areas (inequality plot)	RGB::Black
Frames	the number of frames in an animation	50
Inequalities	inequalities displayed in inequality plots	
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Blue
LineWidth	width of lines	0.35
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	FALSE
Mesh	number of sample points	[256, 256]
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	

Attribute	Purpose	Default Value
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XMax	final value of parameter “x”	
XMesh	number of sample points for parameter “x”	256
XMin	initial value of parameter “x”	
XName	name of parameter “x”	
XRange	range of parameter “x”	

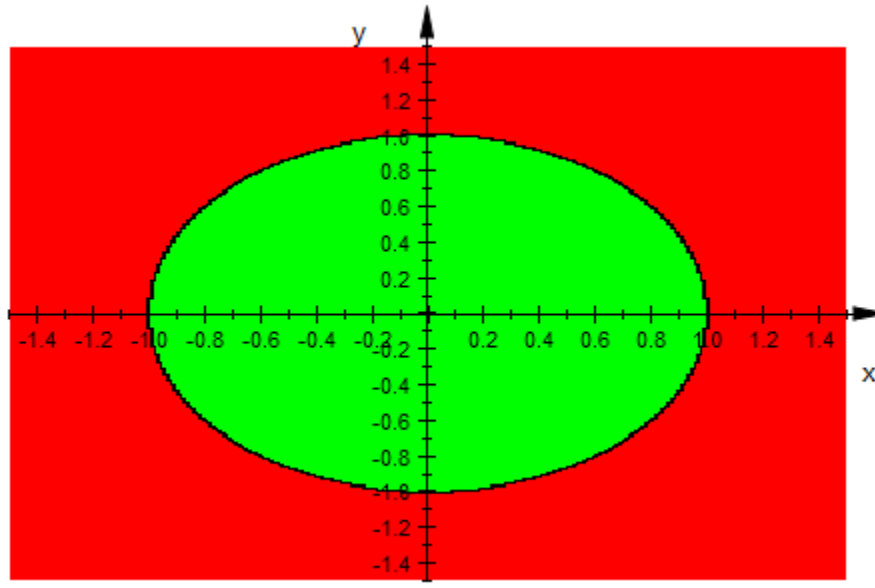
Attribute	Purpose	Default Value
YMax	final value of parameter “y”	
YMesh	number of sample points for parameter “y”	256
YMin	initial value of parameter “y”	
YName	name of parameter “y”	
YRange	range of parameter “y”	

Examples

Example 1

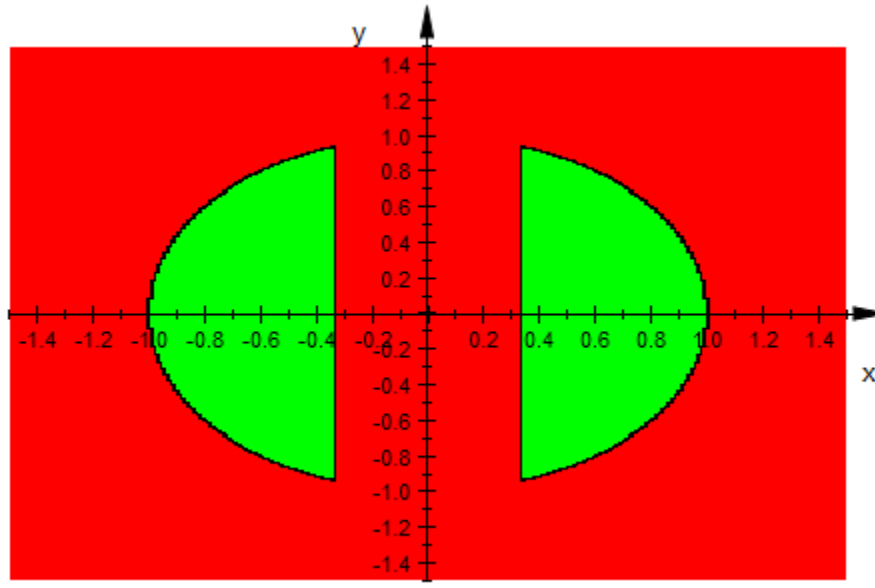
With a single inequality, `plot::Inequality` colors the area where it is fulfilled or violated, with areas at the border line, where the inequality is fulfilled in some parts of the rectangle and violated in other parts:

```
plot(plot::Inequality(x^2 + y^2 < 1,
                      x = -1.5..1.5, y = -1.5..1.5))
```



When giving more than one inequality, only those areas where *all* inequalities are fulfilled are painted in blue (or whatever you set `FillColorTrue` to), while all rectangles where *any* inequality is violated (over the whole rectangle) are colored red:

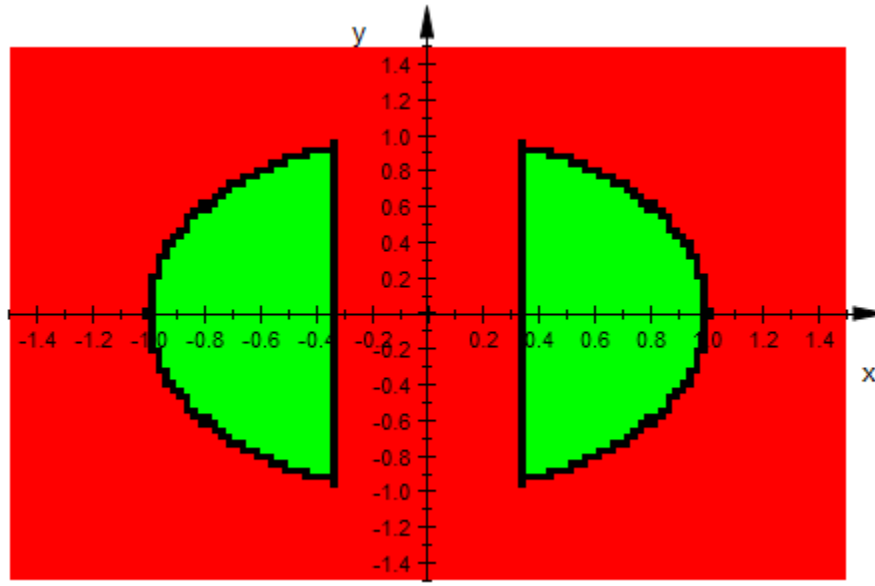
```
plot(plot::Inequality([x^2 + y^2 < 1, abs(x) > 1/3],
                      x = -1.5..1.5, y = -1.5..1.5))
```



Example 2

To get a more detailed image from `plot::Inequality`, increase the mesh density:

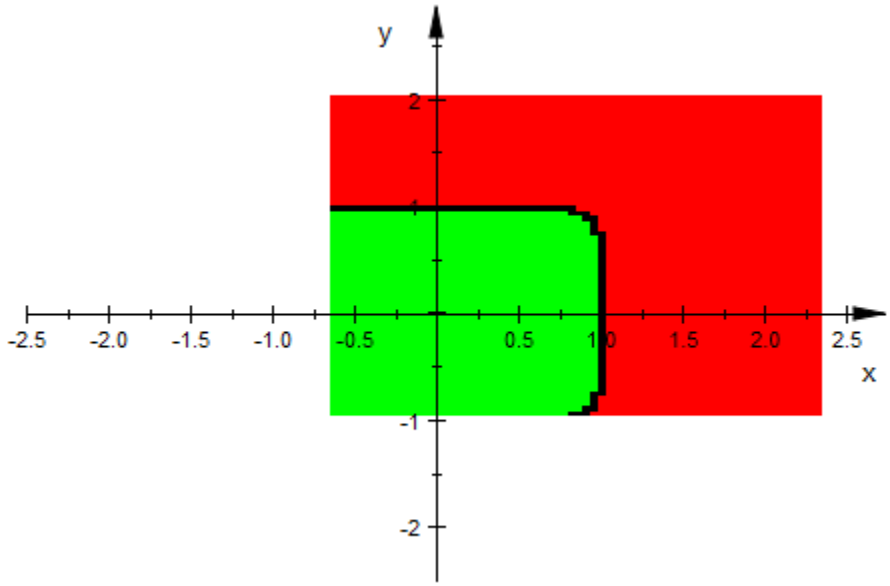
```
plot(plot::Inequality([x^2 + y^2 < 1, abs(x) > 1/3],
                      x = -1.5..1.5, y = -1.5..1.5,
                      Mesh = [120, 80]))
```



Example 3

Almost all parameters of `plot::Inequality` can be animated (the mesh is one exception though):

```
plot(plot::Inequality([abs(x)^a + abs(y)^a < 1],
                      x = -1.5+sin(a)..1.5+sin(a),
                      y = -1.5+cos(a)..1.5+cos(a),
                      Mesh = [64, 64],
                      a = 1..2*PI+1))
```



Parameters

`ineq, ineq`$_1$`, …`

Inequalities to plot: Expressions of the form `f(x, y) < g(x, y)`, `f(x, y) <= g(x, y)`, `f(x, y) = g(x, y)`, `f(x, y) >= g(x, y)`, or `f(x, y) > g(x, y)`.

ineq, ineq$_1$, … is equivalent to the attribute `Inequalities`.

`x, y`

Identifiers or indexed identifiers. These denote the free variables spanning the plane.

x, y are equivalent to the attributes `XName`, `YName`.

`x`$_{min}$ `..` `x`$_{max}$`, y`$_{min}$ `..` `y`$_{max}$

The ranges for `x` and `y`. `x`$_{min}$, `x`$_{max}$, `y`$_{min}$, and `y`$_{max}$ must be real numerical values, or expressions of the animation parameter `a`.

x_{min} .. x_{max}, y_{min} .. y_{max} are equivalent to the attributes `XRange`, `YRange`.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Density` | `plot::Implicit2d` | `plot::Raster`

plot::Integral

Numerical approximation of an integral

Syntax

```
plot::Integral(f, <n>, <IntMethod = m>, <a = a_min .. a_max>, options)
```

Description

`plot::Integral(f, IntMethod = m)` visualizes the approximation of the integral of the function `f` using the numerical quadrature method `m`. Riemann sums, the trapezoidal rule, and the Simpson rule are available.

`plot::Integral(f, n, IntMethod = m)` uses `n` subintervals to approximate the integral.

The attribute `IntMethod` determines the numerical method. Riemann sums, the trapezoidal rule, or the Simpson rule are available. See the help page of `IntMethod` for further details. Cf. “Example 1” on page 24-455.

`plot::Integral` does not plot the function graph of the integrand. If the integrand is to be plotted, too, `f` has to be passed to the `plot` command together with the approximation object of type `plot::Integral`.

If no quadrature method is specified by `IntMethod = m`, `plot::Integral` just hatches the area between the function `f` and the x-axis.

Several `plot::Integral` objects can be plotted together to illustrate the difference between various quadrature methods. The order of the objects in the `plot` command determines the object in front.

The plot contains a text object providing information about the quadrature method, the value of the approximation, the exact value of the integral, the quadrature error, and the number of nodes. See the help page of the attribute `ShowInfo` for further details.

Attributes

Attribute	Purpose	Default Value
AffectViewingBox	influence of objects on the ViewingBox of a scene	TRUE
AntiAliased	antialiased lines and points?	TRUE
Color	the main color	RGB::PaleBlue
Filled	filled or transparent areas and surfaces	TRUE
FillColor	color of areas and surfaces	RGB::PaleBlue
FillPattern	type of area filling	Solid
Frames	the number of frames in an animation	50
Function1	first function/curve delimiting hatch	
HorizontalAlignment	horizontal alignment of text objects w.r.t. their coordinates	Left
IntMethod	method for integral approximation	Exact
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Black
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::Grey
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
LineColorType	line coloring types	Flat

Attribute	Purpose	Default Value
LineColorFunction	functional line coloring	
Name	the name of a plot object (for browser and legend)	
Nodes	number of subintervals or list of x-values for subintervals	[10]
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE
ShowInfo	Information about integral approximation	[2, IntMethod, Integral]
TextFont	font of text objects	[" sans-serif ", 11]
TextRotation	rotation of a 2D text	0
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center

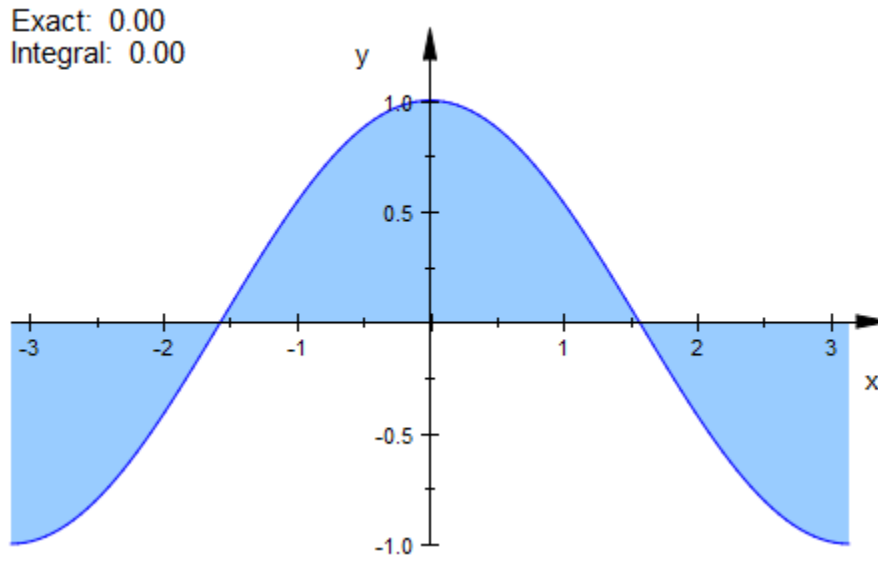
Attribute	Purpose	Default Value
`TitlePositionX`	position of object titles, x component	
`TitlePositionY`	position of object titles, y component	
`VerticalAlignment`	vertical alignment of text objects w.r.t. their coordinates	`Bottom`
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

If a single `plot::Function2d` object is given without specifying an approximation method, `plot::Integral` just hatches the area between the function graph and the x-axis:

```
f := plot::Function2d(cos(x), x = -PI..PI):
plot(plot::Integral(f), f)
```



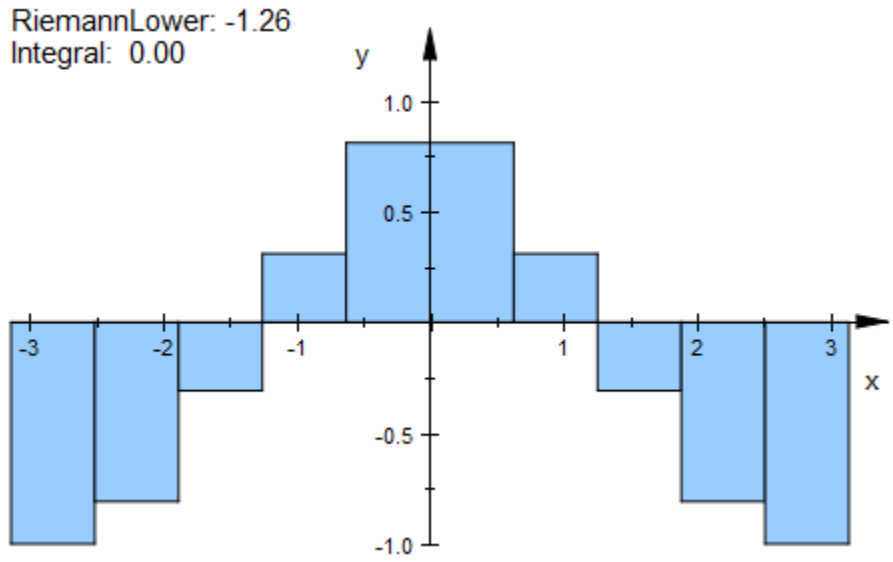
Note that `plot::Integral` requires an *object* of type `plot::Function2d`, not just a function expression:

```
plot::Integral(sin(x))
```

```
Error: The first argument must be a 'plot::Function2d' object. [plot::Integral::new]
```

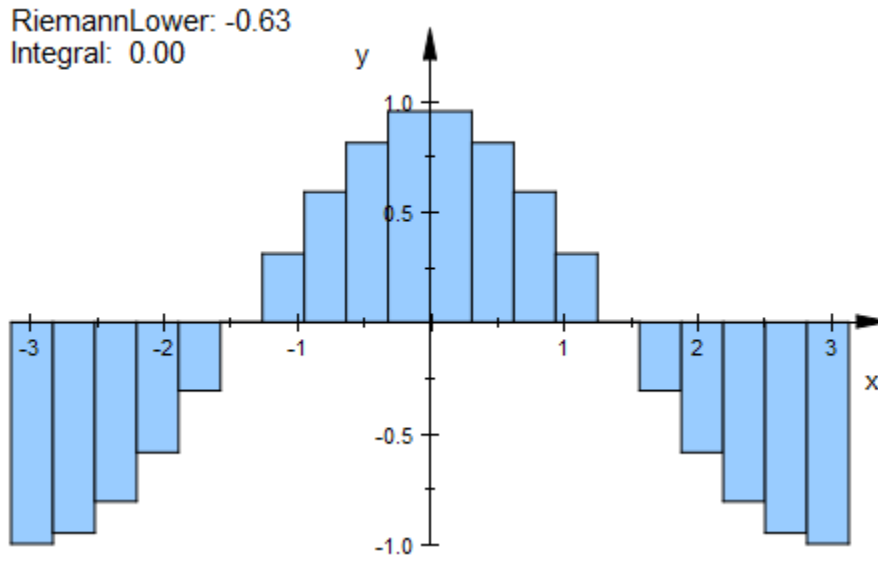
If an approximation method is specified, the numerical quadrature value computed by this method is displayed:

```
plot(plot::Integral(f, IntMethod = RiemannLower))
```



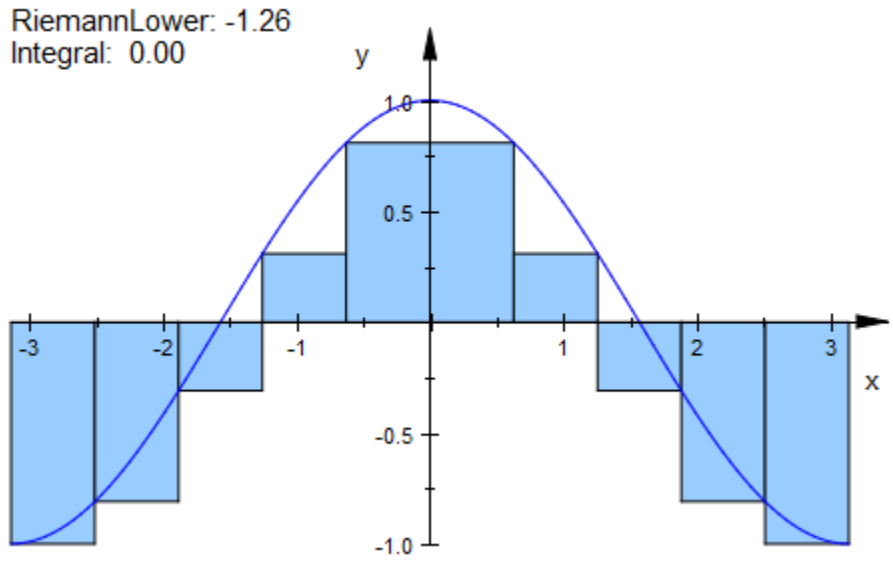
The number of quadrature intervals can be set by passing a second argument `n` or by specifying `Nodes = n`:

```
plot(plot::Integral(f, 20, IntMethod = RiemannLower))
```



To see the integrand in the plot, the function object must be passed together with the approximation object. The order determines which object is in front:

```
plot(plot::Integral(f, IntMethod = RiemannLower), f)
```

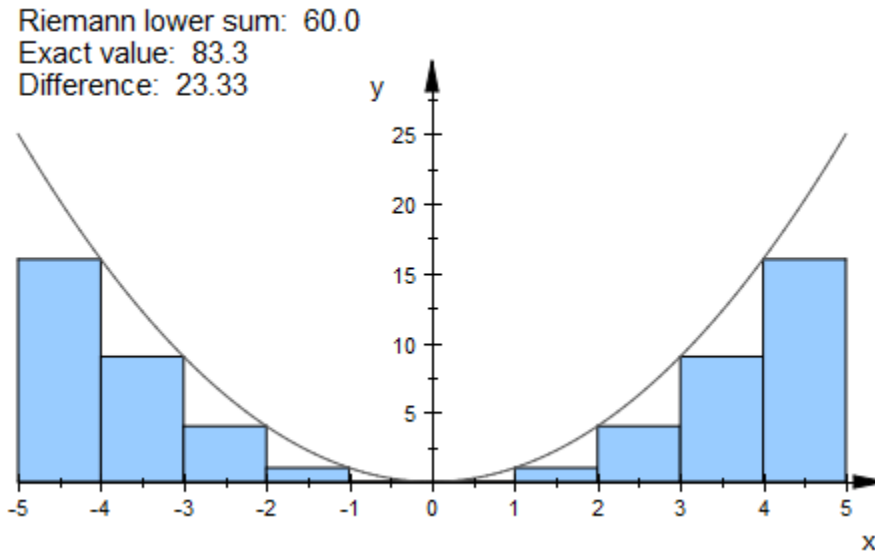



```
delete f:
```

Example 2

The displayed information can be configured by the user:

```
f := plot::Function2d(x^2, x = -5..5, Color = RGB::DarkGrey):
plot(plot::Integral(f, IntMethod = RiemannLower,
       ShowInfo = [1, IntMethod = "Riemann lower sum",
                   Integral = "Exact value",
                   2, Error = "Difference"]), f)
```

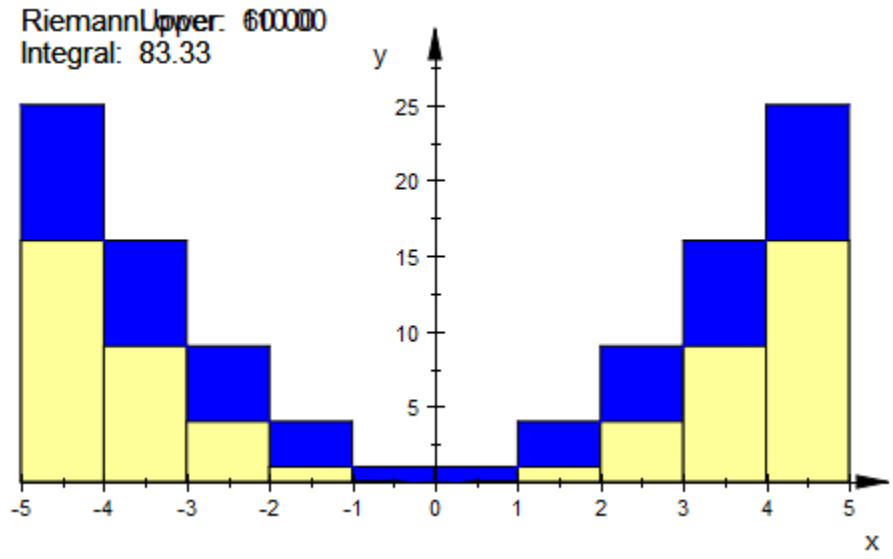



```
delete f:
```

Example 3

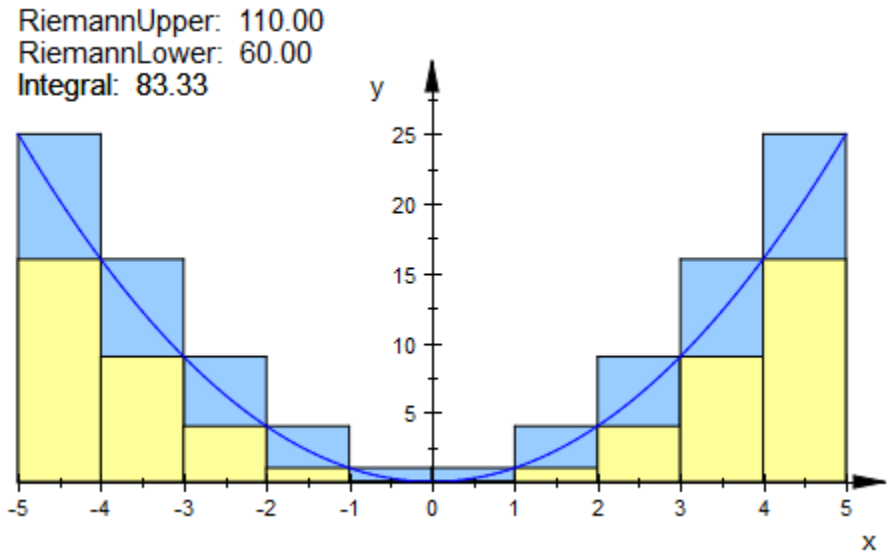
One may combine several approximation objects, e.g., lower and upper sum:

```
f := plot::Function2d(x^2, x = -5..5):
plot(plot::Integral(f, IntMethod = RiemannUpper,
                       Color = RGB::Blue),
     plot::Integral(f, IntMethod = RiemannLower,
                       Color = RGB::LightYellow),
     f)
```



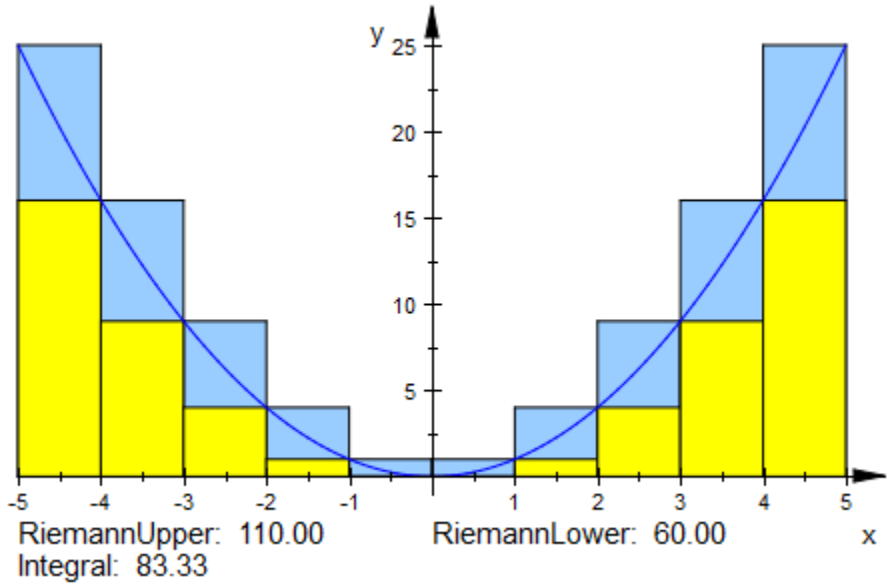
The automatically placed information texts overlap. To correct this, the option `ShowInfo` must be used. In the text of the upper sum, one additional empty line is inserted. Apart from this, both objects use the default value, therefore there is not need to specify `ShowInfo` in the second object:

```
plot(plot::Integral(f, IntMethod = RiemannUpper,
                       ShowInfo = [IntMethod, "", Integral]),
     plot::Integral(f, IntMethod = RiemannLower,
                       Color = RGB::LightYellow),
     f)
```



The info text can be positioned explicitly:

```
plot(plot::Integral(f, IntMethod = RiemannUpper,
                       ShowInfo = [IntMethod, Integral,
                                   Position = [-5, -1]],
                       VerticalAlignment = Top),
     plot::Integral(f, IntMethod = RiemannLower, Color = RGB::Yellow,
                       ShowInfo = [IntMethod,
                                   Position = [0, -1]],
                       VerticalAlignment = Top),
     f)
```

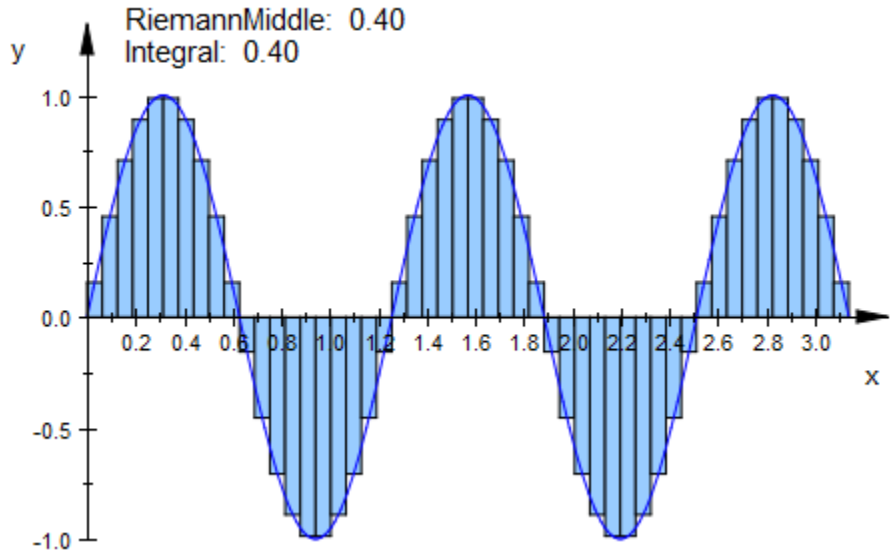



```
delete f:
```

Example 4

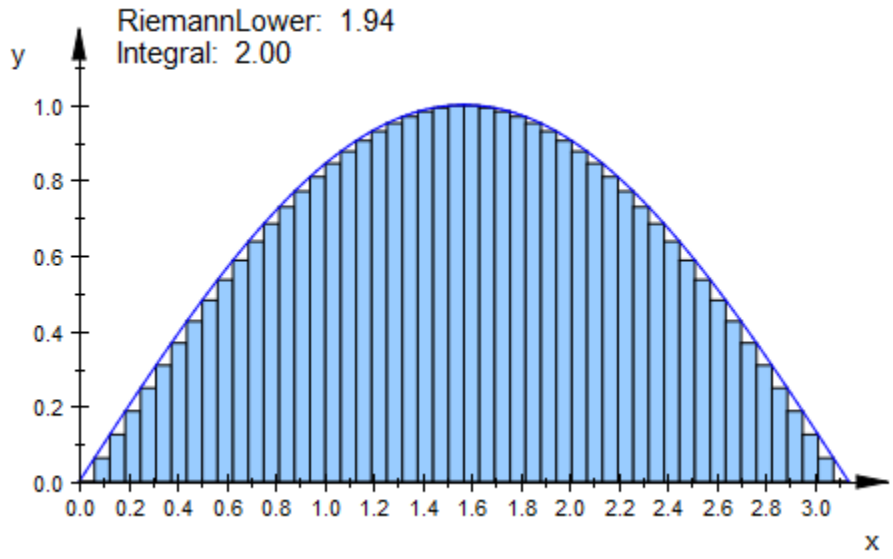
`plot::Integral` can be animated:

```
f := plot::Function2d(sin(a*x), x = 0..PI, a = 1..5):
plot(plot::Integral(f, 50, IntMethod = RiemannMiddle), f)
```



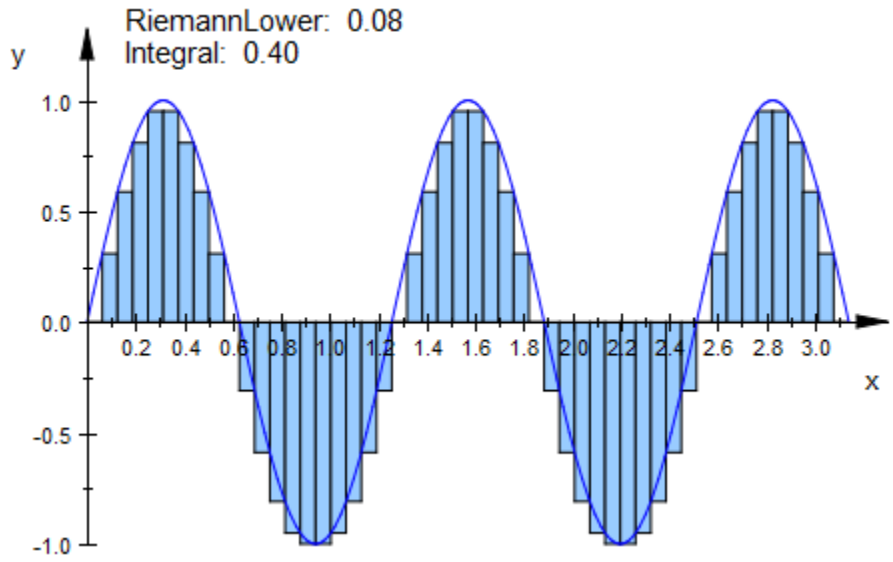
Increasing the number of nodes decreases the quadrature error:

```
f := plot::Function2d(sin(x), x = 0..PI):
plot(plot::Integral(f, N, N = 10..50, IntMethod = RiemannLower), f)
```



The function and the number of nodes can be animated simultaneously:

```
f := plot::Function2d(sin(a*x), x = 0..PI, a = 1..5):
plot(plot::Integral(f, N, N = 10..50, IntMethod = RiemannLower), f)
```




```
delete f:
```

Parameters

f

The integrand: an object of type `plot::Function2d`.

f is equivalent to the attribute `Function1`.

n

The number of subintervals (a positive integer) or a list of real numbers representing nodes of the integration variable.

n is equivalent to the attribute `Nodes`.

a

Animation parameter, specified as a = $a_{min}..a_{max}$, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

Options

`IntMethod`

Option, specified as `IntMethod = m`

The quadrature method; see `IntMethod`

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Function2d` | `plot::Hatch` | `plot::Text2d`

plot::Iteration

Plotting iterated functions

Syntax

```
plot::Iteration(f, x0, <n>, x = xmin .. xmax, <a = amin .. amax>, options)
```

Description

`plot::Iteration`(f, x_0, n, x = `` `x_{min}` .. `x_{max}` ``) is a graphical object visualizing the iteration $x_i = f(x_{i-1})$ ($i = 1, \ldots, n$) of the given starting point x_0.

The iteration is visualized by connecting the points $(x_0, 0)$ and (x_0, x_1) by a vertical line. For any step of the iteration, a horizontal line is drawn from the point (x_{i-1}, x_i) (on the graph of f) to the point (x_i, x_i) on the main diagonal. From there, a vertical line is drawn to the next pair (x_i, x_{i+1}) of the iteration.

The iteration object neither includes the graph of the function $y = f(x)$ nor the main diagonal $y = x$. You need to plot them separately if you wish the function and/or the diagonal to be in your picture! See the examples.

The iteration is stopped prematurely when the iterated point leaves the plot range `` `x_{min}`..`x_{max}` ``. Cf. “Example 3” on page 24-475.

Despite the fact that the number of iterations `n` represents an integer, it can be animated! Cf. “Example 4” on page 24-477

The default color used for the iteration plot is `RGB::Grey50`. It can be modified by setting the attribute `Color` or `LineColor`. Cf. “Example 1” on page 24-471.

The default line style is solid. It can be modified by setting the attribute `LineStyle`.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`

Attribute	Purpose	Default Value
AntiAliased	antialiased lines and points?	FALSE
Color	the main color	RGB::Grey50
Frames	the number of frames in an animation	50
Function	function expression or procedure	
Iterations	number of iterations in plot::Iteration	10
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Grey50
LineWidth	width of lines	0.35
LineStyle	solid, dashed or dotted lines?	Solid
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
StartingPoint	starting point of the iteration	
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0

Attribute	Purpose	Default Value
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XMax	final value of parameter “x”	
XMin	initial value of parameter “x”	
XName	name of parameter “x”	
XRange	range of parameter “x”	

Examples

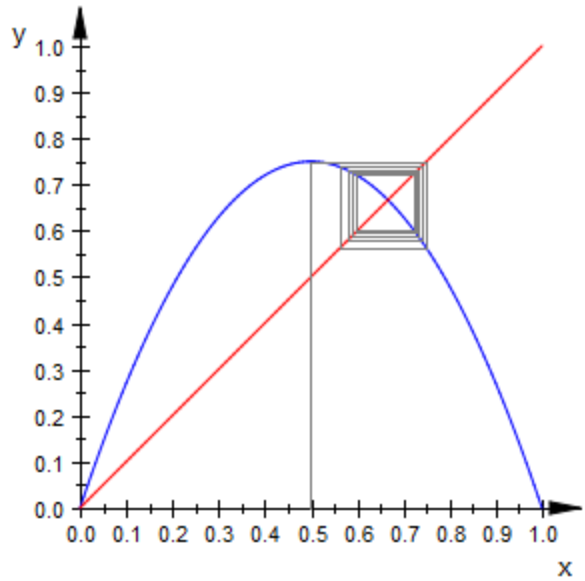
Example 1

We consider the logistic map for the parameter value 3, i.e., the parabola $f(x) = 3\,x\,(1 - x)$ for $x \in [0, 1]$. We iterate the starting point $x_0 = 0.5$:

```
f :=  plot::Function2d(3*x*(1 - x), x = 0..1,
                       Color = RGB::Blue):
x0 := 0.5:
```

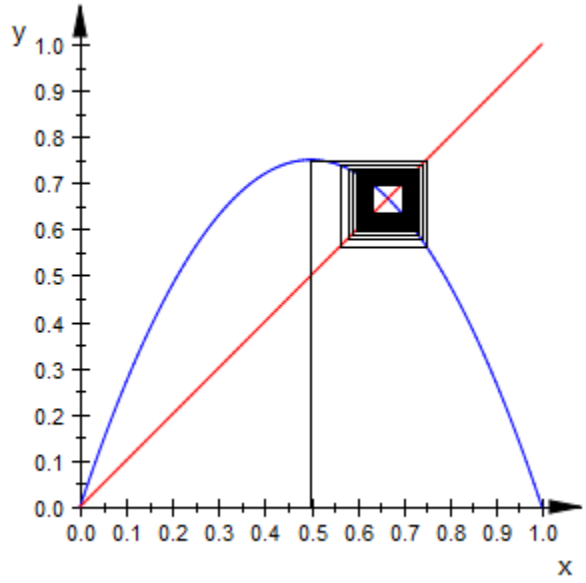
We plot the iteration (without specifying the number of iterations), the parabola f and the diagonal line $g(x) = x$:

```
g := plot::Function2d(x, x = 0..1, Color = RGB::Red):
it := plot::Iteration(3*x*(1 - x), x0, x = 0..1):
plot(f, g, it)
```



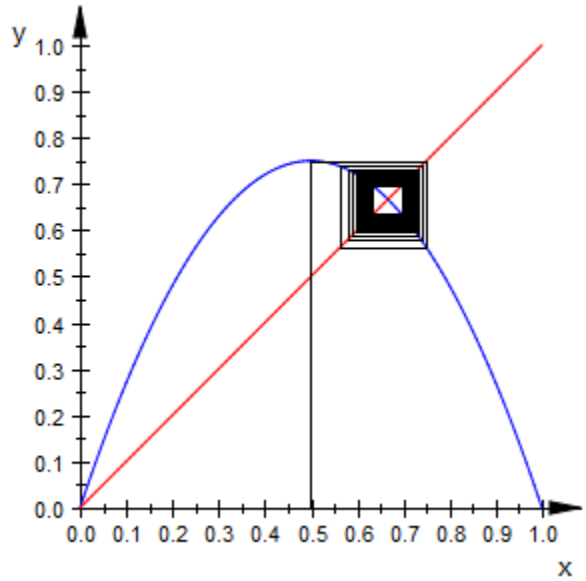
We increase the number of iterations to 50 and change the color of the lines to `RGB::Black`:

```
it::Iterations := 50:
it::Color := RGB::Black:
plot(f, g, it)
```



Finally, we animate the number of steps, allowing to follow the course of the iteration:

```
it := plot::Iteration(3*x*(1 - x), x0, n, x = 0..1,
                      n = 1..50, Color = RGB::Black):
plot(f, g, it)
```




```
delete f, g, it:
```

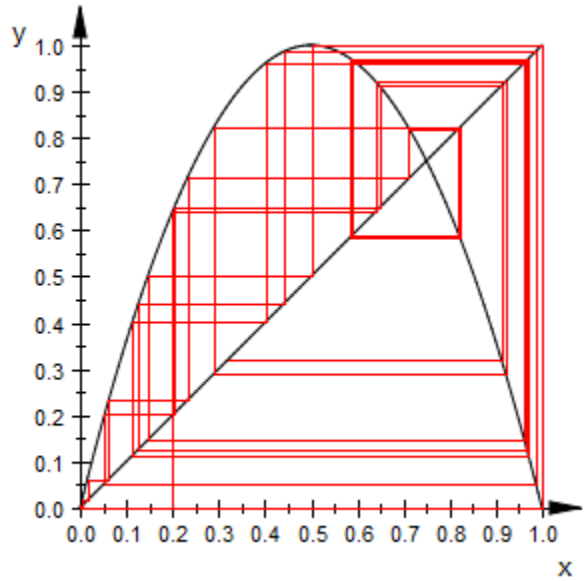
Example 2

We consider the logistic map $f(x) = a\,x\,(1 - x)$ for $x \in [0, 1]$ and the animation parameter a running from $a = 2$ to $a = 4$:

```
f := plot::Function2d(a*x*(1 - x), x = 0..1, a = 2..4,
                      Color = RGB::Black):
```

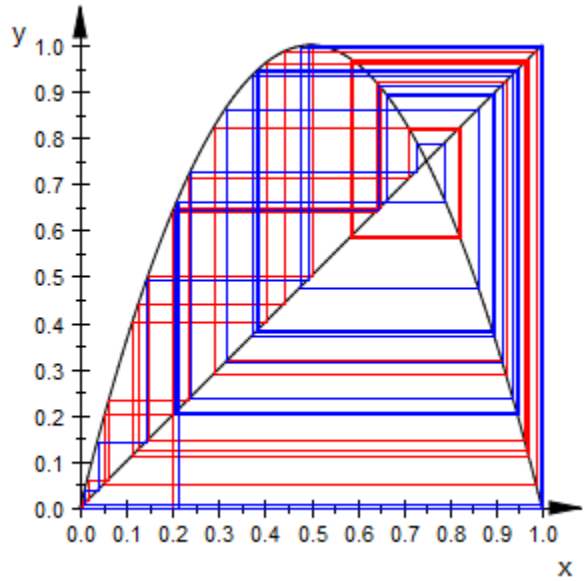
We define the iteration of the starting point $x_0 = 0.2$ by f and plot it together with the function graph of $f(x)$ and the diagonal line $g(x) = x$:

```
g := plot::Function2d(x, x = 0..1, Color = RGB::Black):
it1 := plot::Iteration(a*x*(1 - x), 0.2, 30, x = 0..1,
                       a = 2..4, Color = RGB::Red):
plot(f, g, it1)
```



We define an additional iteration starting at $x_0 = 0.21$ and add it to the plot:

```
it2 := plot::Iteration(a*x*(1 - x), 0.21, 30, x = 0..1,
                       a = 2..4, Color = RGB::Blue):
plot(f, g, it1, it2)
```



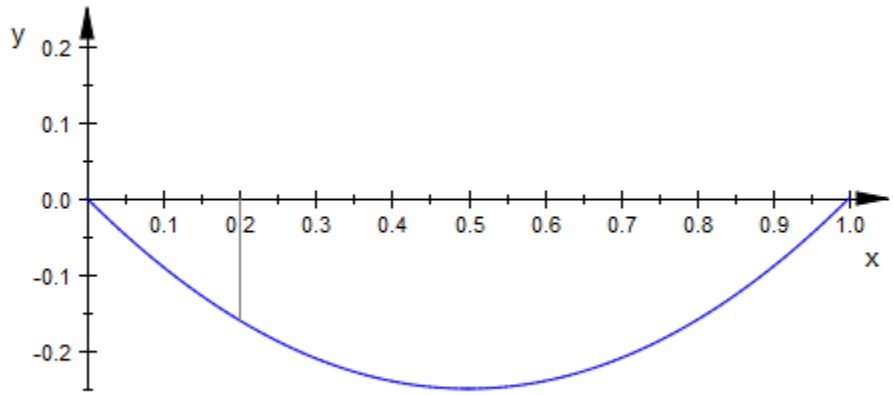
For small values of a, the two iterations converge to the same fixed point. When a approaches the value 4, the iterations drift into chaos.

```
delete f, g, it1, it2:
```

Example 3

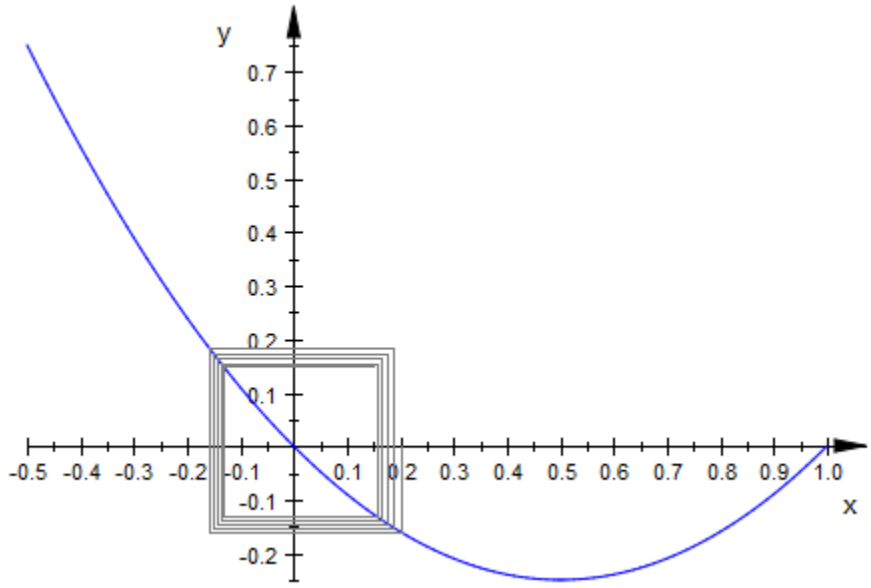
Consider the iteration of the starting point $x_0 = 0.2$ by the logistic map $f(x) = x\,(x - 1)$ with the plot range $x \in [0, 1]$:

```
f := plot::Function2d(x*(x - 1), x = 0..1):
it := plot::Iteration(x*(x - 1), 0.2, x = 0..1):
plot(f, it)
```



We see that only one step of the iteration is plotted. The reason is that the point $x_1 = f(x_0)$ is negative and, hence, not contained in the requested plot range `x = 0..1`. We modifiy the plot range:

```
f::XRange:= -0.5..1:
it::XRange:= -0.5..1:
plot(f, it)
```

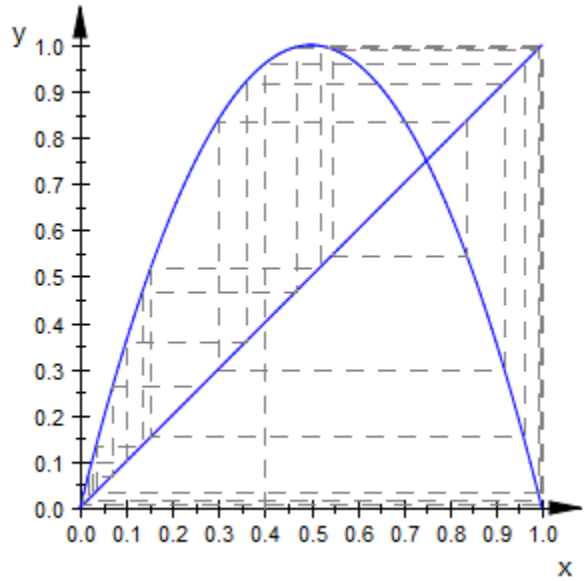



```
delete f, it:
```

Example 4

We animate the parameter n that sets the number of iterations. We set the time range for the animation to 40 (seconds). Using `Frames`, the total number of frames is chosen such that approximately 10 frames are used to visualize the step from n to $n + 1$:

```
f := plot::Function2d(4*x*(1 - x), x = 0..1):
g := plot::Function2d(x, x = 0..1):
it := plot::Iteration(4*x*(1 - x), 0.4, n, x = 0..1,
                      LineStyle = Dashed,
                      n = 0..40, Frames = 411,
                      TimeRange = 0..40):
plot(f, g, it)
```




```
delete f, g, it:
```

Parameters

f

The iteration function: an arithmetical expression in the independent variable `x` and the animation parameter `a`. Alternatively, a procedure that accepts 1 input parameter `x` or 2 input parameters `x`, `a` and returns a real numerical value when the input parameters are numerical.

f is equivalent to the attribute `Function`.

x_0

The starting point for the iteration: x_0 must be a numerical real value or an expression in the animation parameter `a`.

x_0 is equivalent to the attribute `StartingPoint`.

n

The number of iterations: `n` must be a positive integer or an expression in the animation parameter `a`.

n is equivalent to the attribute `Iterations`.

x

The independent variable: an identifier or an indexed identifier.

x is equivalent to the attribute `XName`.

x_{min} .. x_{max}

The plot range: x_{min}, x_{max} must be numerical real values or expressions in the animation parameter a.

x_{min} .. x_{max} is equivalent to the attributes `XRange`, `XMin`, `XMax`.

a

Animation parameter, specified as a = a_{min}..a_{max}, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Lsys`

plot::Line2d

2D line segments

Syntax

```
plot::Line2d([x1, y1], [x2, y2], <a = amin .. amax>, options)
```

Description

`plot::Line2d([x1, y1], [x2, y2])` creates a 2D line segment between the points (x_1, y_1) and (x_2, y_2).

The end points may be passed as lists or vectors.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`
`Color`	the main color	`RGB::Blue`
`Extension`	line extensions	`Finite`
`Frames`	the number of frames in an animation	`50`
`From`	starting point of arrows and lines	`[0, 0]`
`FromX`	starting point of arrows and lines, x-coordinate	`0`
`FromY`	starting point of arrows and lines, y-coordinate	`0`
`Legend`	makes a legend entry	

Attribute	Purpose	Default Value
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`LineColor`	color of lines	`RGB::Blue`
`LineWidth`	width of lines	`0.35`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`
`Title`	object title	
`TitleFont`	font of object titles	`[" sans-serif ", 11]`
`TitlePosition`	position of object titles	
`TitleAlignment`	horizontal alignment of titles w.r.t. their coordinates	`Center`
`TitlePositionX`	position of object titles, x component	
`TitlePositionY`	position of object titles, y component	

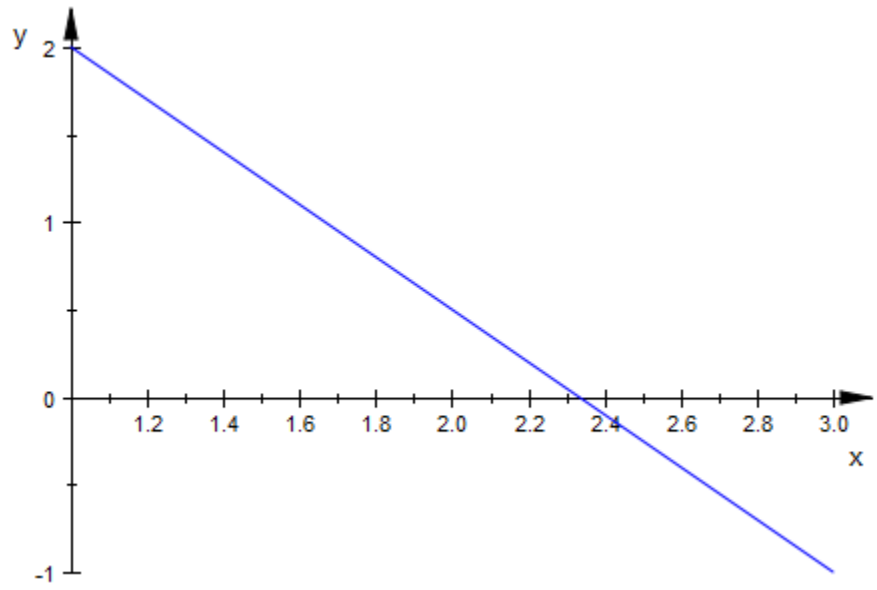
Attribute	Purpose	Default Value
`To`	end point of arrows and lines	[1, 0]
`ToX`	end point of arrows and lines, x-coordinate	1
`ToY`	end point of arrows and lines, y-coordinate	0
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

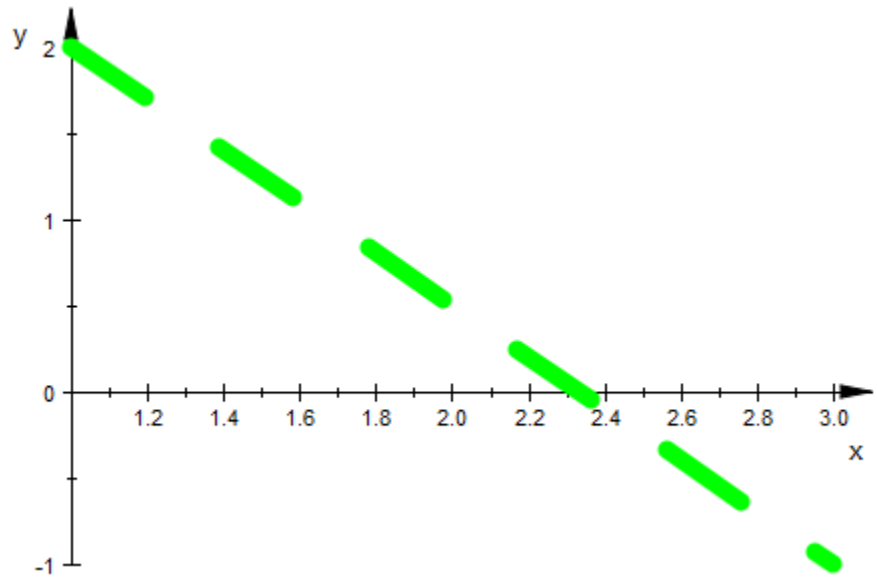
We create a 2D line segement:

```
plot(plot::Line2d([1, 2], [3,-1]))
```



The `LineStyle` can be changed from `Solid`, as is the default, to `Dashed` or `Dotted`. Likewise `LineColor` and `LineWidth` can be set explicitly:

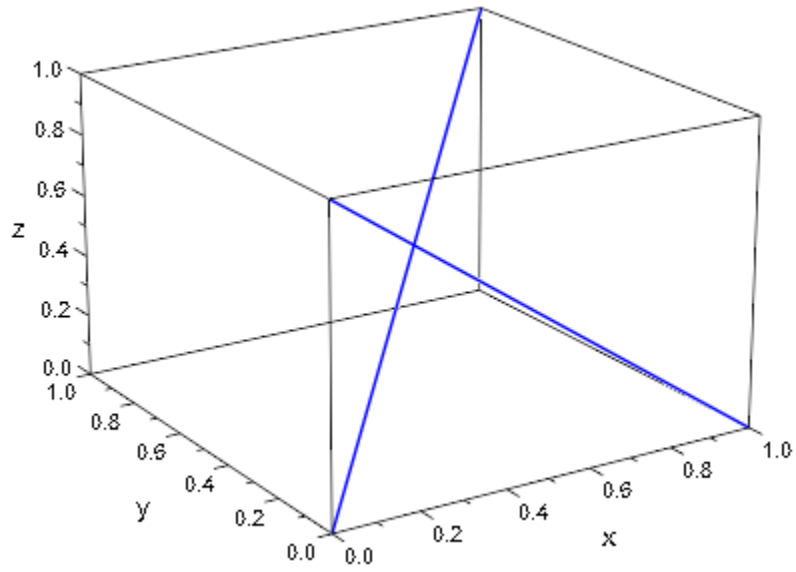
```
plot(plot::Line2d([1, 2], [3, -1],
                  LineStyle = Dashed,
                  LineWidth = 2.5*unit::mm,
                  LineColor = RGB::Green))
```



Example 2

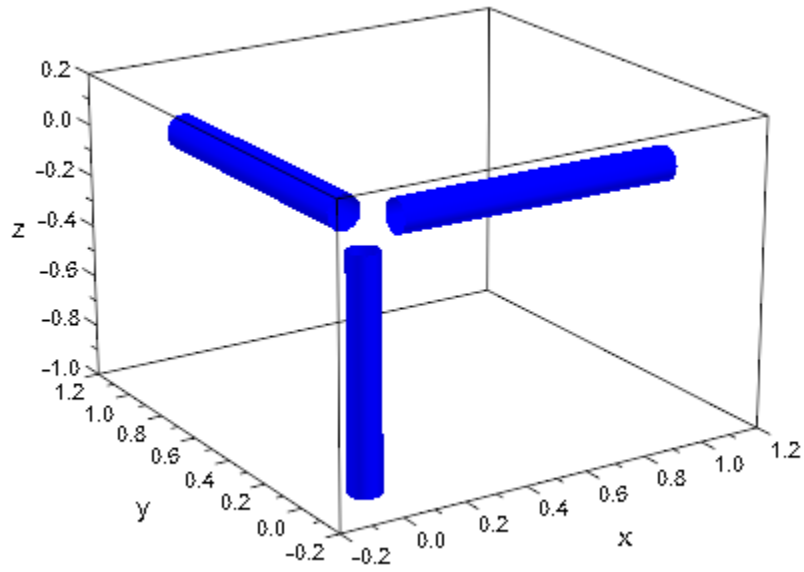
We plot two animated 3D line segements starting off parallel, ending up skew:

```
plot(plot::Line3d([0, 0, 0], [a, a, 1], a = 0..1),
     plot::Line3d([1, 0, 0], [a, 0, 1], a = 1..0))
```



In addition to `LineStyle`, `LineColor` and `LineWidth`, 3D line segments support the style option `Tubular`. If this is set to `TRUE`, the `TubeDiameter` can be set explicitly:

```
plot(plot::Line3d([0.1, 0, 0], [1, 0, 0]),
     plot::Line3d([0, 0.1, 0], [0, 1, 0]),
     plot::Line3d([0, 0, -0.1], [0, 0,-1]),
     ViewingBox = [-0.2..1.2,  -0.2..1.2, -1..0.2],
     Tubular = TRUE, TubeDiameter = 5.0*unit::mm)
```



Parameters

x_1, y_1

The coordinates of one end point: real numerical values or arithmetical expressions of the animation parameter `a`.

x_1, y_1 are equivalent to the attributes `FromX`, `FromY`.

x_2, y_2

The coordinates of the other end point: real numerical values or arithmetical expressions of the animation parameter `a`.

x_2, y_2, z_2 are equivalent to the attributes `ToX`, `ToY`, `ToZ`.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Line3d` | `plot::Polygon2d` | `plot::Polygon3d` | `plot::Rectangle`

plot::Line3d

3D line segments

Syntax

```
plot::Line3d([x1, y1, z1], [x2, y2, z2], <a = amin .. amax>, options)
```

Description

`plot::Line3d([x1, y1, z1], [x2, y2, z2])` creates a 3D line segment from (x_1, y_1, z_1) to (x_2, y_2, z_2).

The end points may be passed as lists or vectors.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Color`	the main color	`RGB::Blue`
`Extension`	line extensions	`Finite`
`Frames`	the number of frames in an animation	`50`
`From`	starting point of arrows and lines	`[0, 0, 0]`
`FromX`	starting point of arrows and lines, x-coordinate	`0`
`FromY`	starting point of arrows and lines, y-coordinate	`0`
`FromZ`	starting point of arrows and lines, z-coordinate	`0`
`Legend`	makes a legend entry	

Attribute	Purpose	Default Value
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Blue
LineWidth	width of lines	0.35
LineStyle	solid, dashed or dotted lines?	Solid
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	

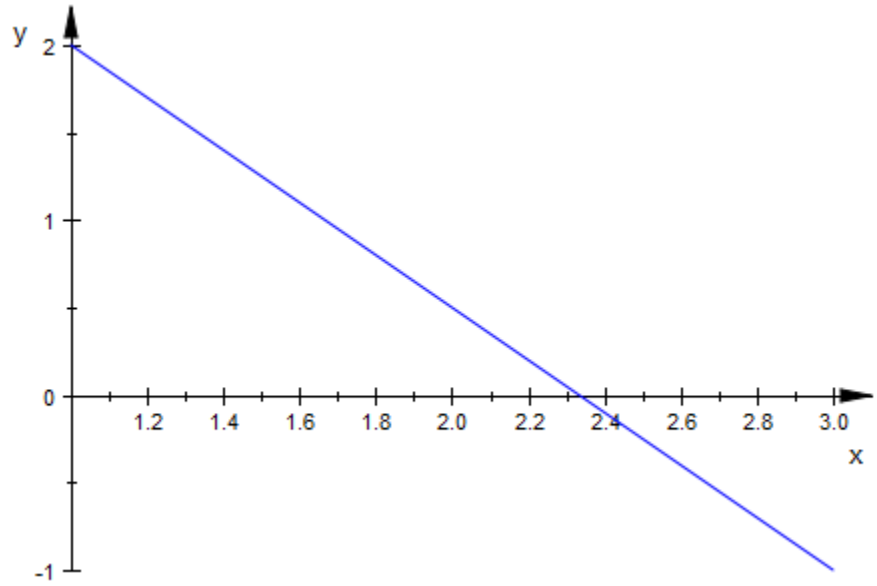
Attribute	Purpose	Default Value
TitlePositionZ	position of object titles, z component	
To	end point of arrows and lines	[1, 0, 0]
ToX	end point of arrows and lines, x-coordinate	1
ToY	end point of arrows and lines, y-coordinate	0
ToZ	end point of arrows and lines, z-coordinate	0
Tubular	display 3D arrows and lines as tubes?	FALSE
TubeDiameter	diameter of tubular arrows and lines.	1.0
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE

Examples

Example 1

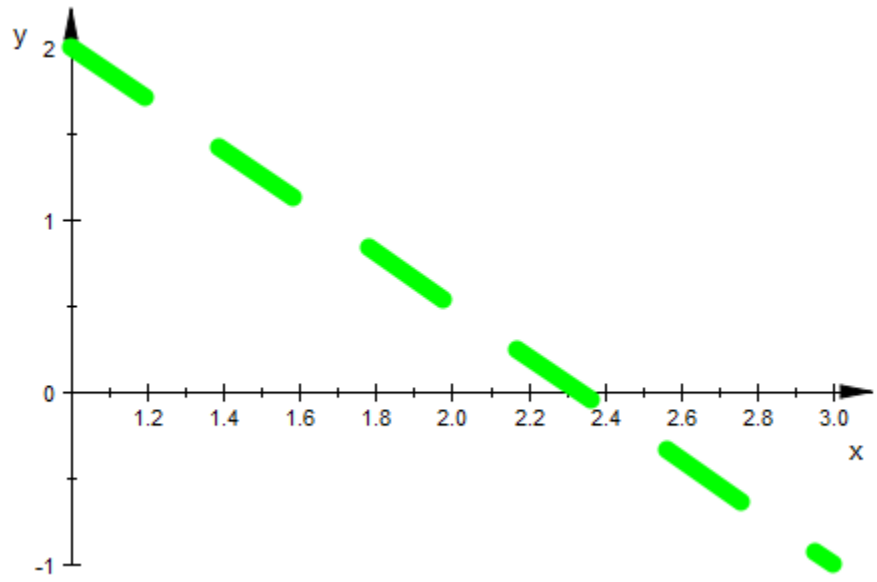
We create a 2D line segement:

```
plot(plot::Line2d([1, 2], [3,-1]))
```



The `LineStyle` can be changed from `Solid`, as is the default, to `Dashed` or `Dotted`. Likewise `LineColor` and `LineWidth` can be set explicitly:

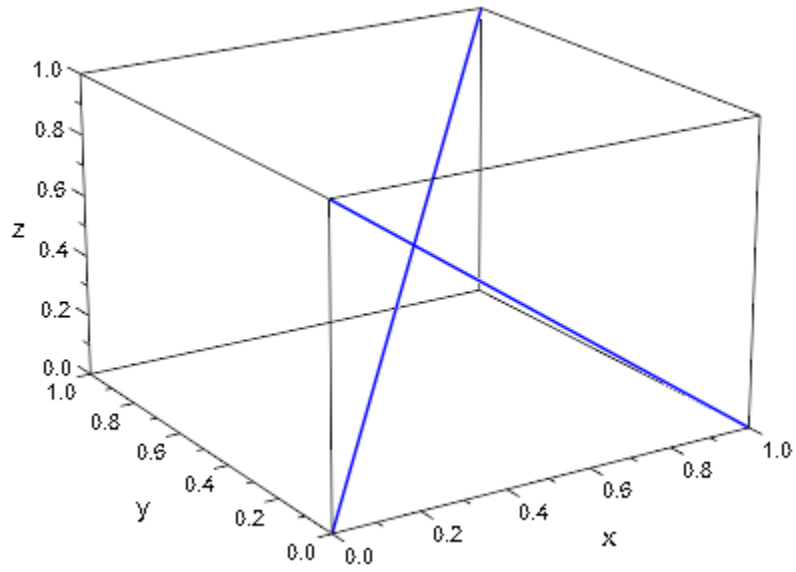
```
plot(plot::Line2d([1, 2], [3, -1],
                  LineStyle = Dashed,
                  LineWidth = 2.5*unit::mm,
                  LineColor = RGB::Green))
```



Example 2

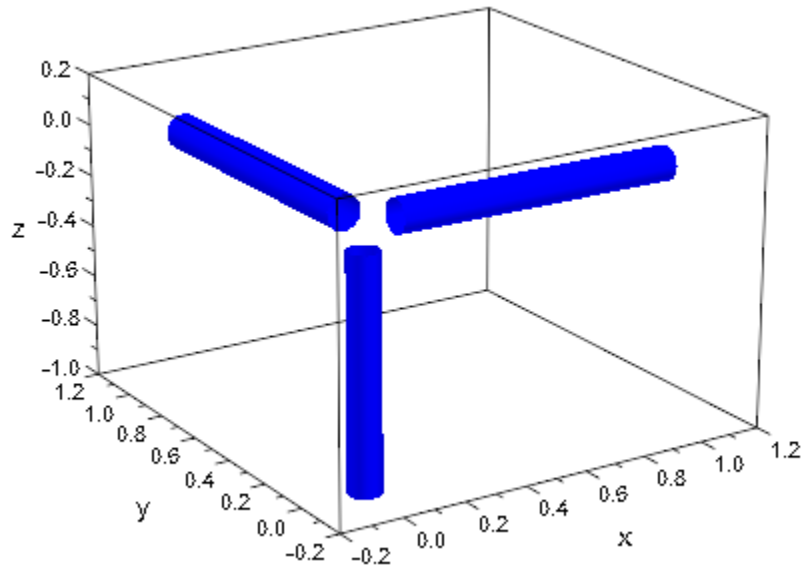
We plot two animated 3D line segements starting off parallel, ending up skew:

```
plot(plot::Line3d([0, 0, 0], [a, a, 1], a = 0..1),
     plot::Line3d([1, 0, 0], [a, 0, 1], a = 1..0))
```



In addition to `LineStyle`, `LineColor` and `LineWidth`, 3D line segments support the style option `Tubular`. If this is set to `TRUE`, the `TubeDiameter` can be set explicitly:

```
plot(plot::Line3d([0.1, 0, 0], [1, 0, 0]),
     plot::Line3d([0, 0.1, 0], [0, 1, 0]),
     plot::Line3d([0, 0, -0.1], [0, 0,-1]),
     ViewingBox = [-0.2..1.2,  -0.2..1.2, -1..0.2],
     Tubular = TRUE, TubeDiameter = 5.0*unit::mm)
```



Parameters

x_1, y_1, z_1

The coordinates of one end point: real numerical values or arithmetical expressions of the animation parameter `a`.

x_1, y_1, z_1 are equivalent to the attributes `FromX`, `FromY`, `FromZ`.

x_2, y_2, z_2

The coordinates of the other end point: real numerical values or arithmetical expressions of the animation parameter `a`.

x_2, y_2, z_2 are equivalent to the attributes `ToX`, `ToY`, `ToZ`.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Line2d` | `plot::Polygon2d` | `plot::Polygon3d` | `plot::Rectangle`

plot::Listplot

Finite lists of 2D points

Syntax

`plot::Listplot([`y_1`, `y_2`, …], <x = `x_{min}` .. `x_{max}`>, <a = `a_{min}` .. `a_{max}`>, options)`

`plot::Listplot(`A_1`, <x = `x_{min}` .. `x_{max}`>, <a = `a_{min}` .. `a_{max}`>, options)`

`plot::Listplot([[`x_1`, `y_1`], [`x_2`, `y_2`], …], <a = `a_{min}` .. `a_{max}`>, options)`

`plot::Listplot(`A_2`, <a = `a_{min}` .. `a_{max}`>, options)`

Description

`plot::Listplot` serves for visualizing discrete data values `[`y_1`, `y_2`, …]`. If no range `x = `x_{min}` .. `x_{max}`` is specified, the data are plotted as the points `[`$x_1$`, `$y_1$`]`, `[`$x_2$`, `$y_2$`]` etc. with equidistant *x*-values `x1 = 1`, `x2 = 2` etc. If a range `x = `x_{min}` .. `x_{max}`` is specified, equidistant *x*-values between `x_min` and `x_max` are used.

If the data are specified as a list of coordinate pairs `[[`x_1`, `y_1`], [`x_2`, `y_2`], …]`, `plot::Listplot` generates plot points with these coordinates.

With the attribute `LinesVisible` = `TRUE`, each pair of consecutive data points is connected by a curve.

With `InterpolationStyle` = `Linear` (default), the points are connected by straight line segments.

With `InterpolationStyle` = `Cubic`, a cubic spline curve is used to connect the points. The spline curve between two data points is rendered as a collection of $m + 1$ straight line segments, where m is specified by the attribute `Submesh` = `m`.

Use `LinesVisible` = `FALSE`, if only the data points without connecting lines are to be rendered.

Attributes

Attribute	Purpose	Default Value
AffectViewingBox	influence of objects on the ViewingBox of a scene	TRUE
AntiAliased	antialiased lines and points?	TRUE
Data	the (statistical) data to plot	
FillColorDirection	the direction of color transitions on surfaces	[0, 0]
Frames	the number of frames in an animation	50
InterpolationStyle	interpolation via linear or cubic splines	Linear
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Blue
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	1

Attribute	Purpose	Default Value
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointColor	the color of points	RGB::Black
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	TRUE
Submesh	density of submesh (additional sample points)	6
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	

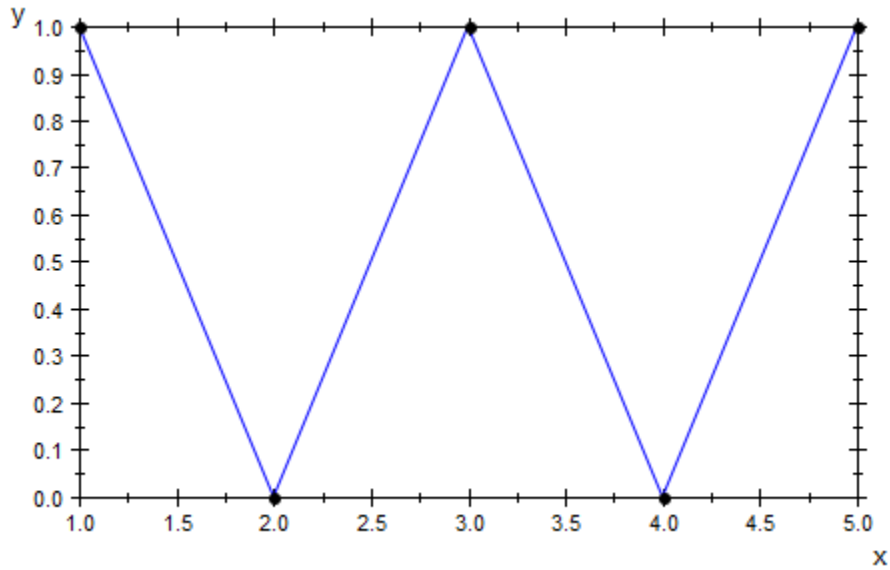
Attribute	Purpose	Default Value
`TitlePositionY`	position of object titles, y component	
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`
`XMax`	final value of parameter “x”	
`XMin`	initial value of parameter “x”	
`XName`	name of parameter “x”	
`XRange`	range of parameter “x”	
`XSubmesh`	density of additional sample points for parameter “x”	`6`

Examples

Example 1

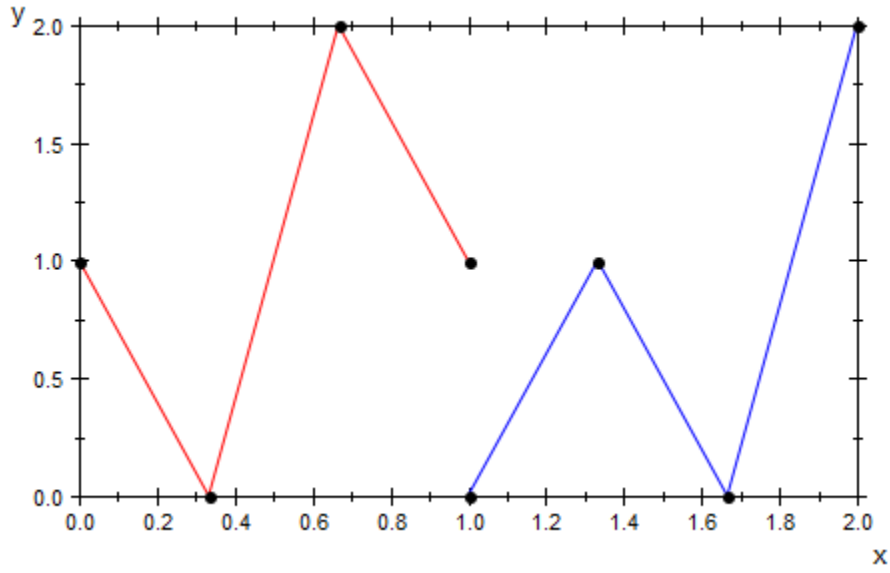
We plot 5 discrete data values as points with equidistant x-values 1, 2, 3, 4, 5:

```
plot(plot::Listplot([1, 0, 1, 0, 1]))
```



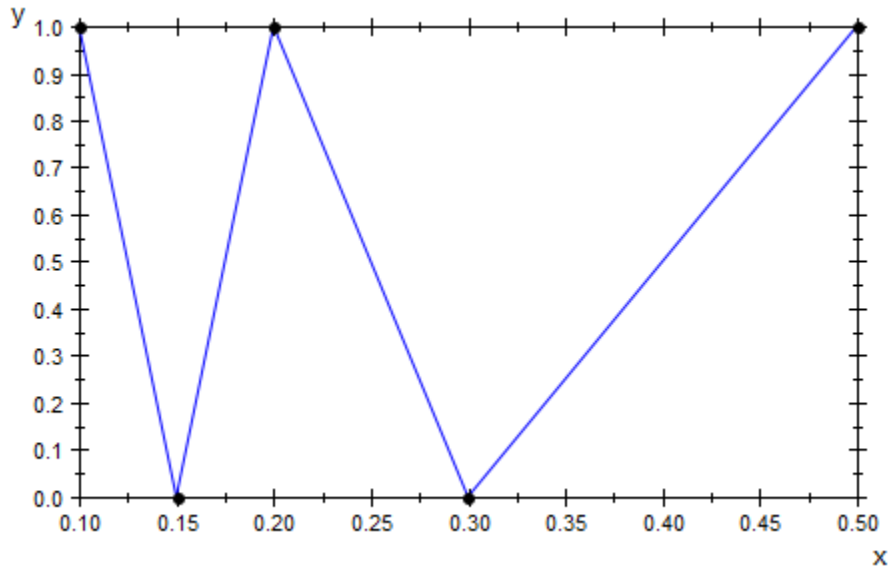
We plot two data samples and place them side by side by specifying suitable ranges for the horizontal variable:

```
plot(plot::Listplot([1, 0, 2, 1], x = 0..1, Color = RGB::Red),
     plot::Listplot([0, 1, 0, 2], x = 1..2, Color = RGB::Blue))
```



We specify x-coordinates for the data points:

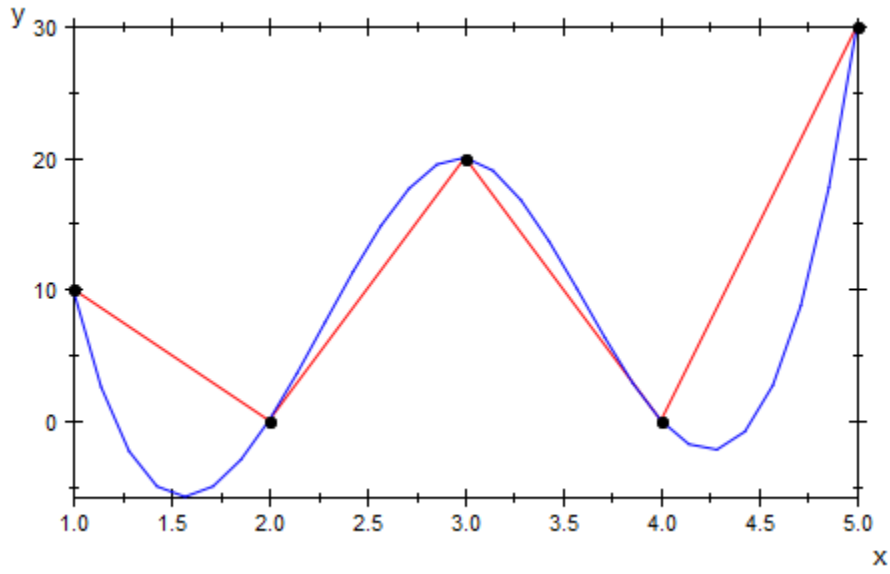
```
plot(plot::Listplot([[0.1, 1], [0.15, 0], [0.2, 1],
                     [0.3, 0], [0.5, 1]]))
```



Example 2

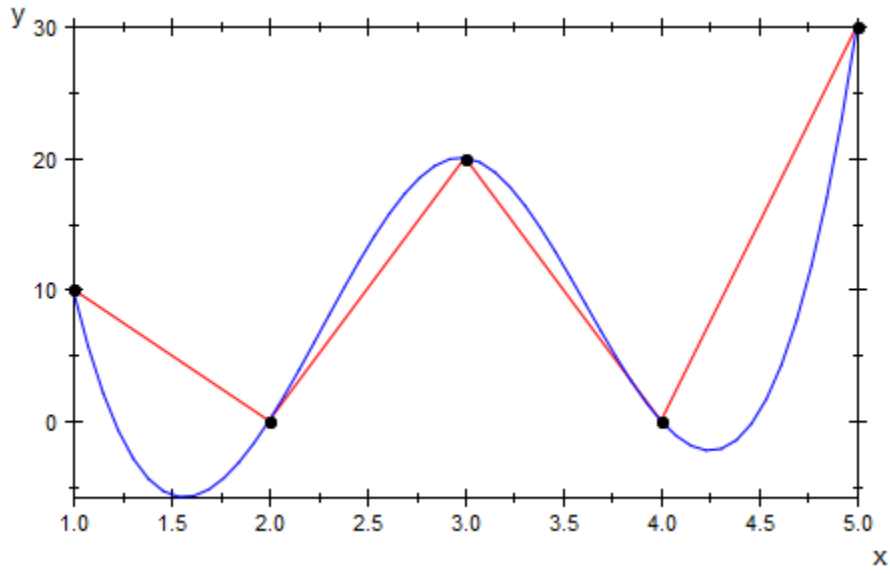
We demonstrate the difference between linear and cubic spline interpolation:

```
plot(plot::Listplot([10, 0, 20, 0, 30], Color = RGB::Red,
                    InterpolationStyle = Linear),
     plot::Listplot([10, 0, 20, 0, 30], Color = RGB::Blue,
                    InterpolationStyle = Cubic))
```



We smoothen the cubic spline curve by increasing the `Submesh` value:

```
plot(plot::Listplot([10, 0, 20, 0, 30], Color = RGB::Red,
                    InterpolationStyle = Linear),
     plot::Listplot([10, 0, 20, 0, 30], Color = RGB::Blue,
                    InterpolationStyle = Cubic, Submesh = 12))
```

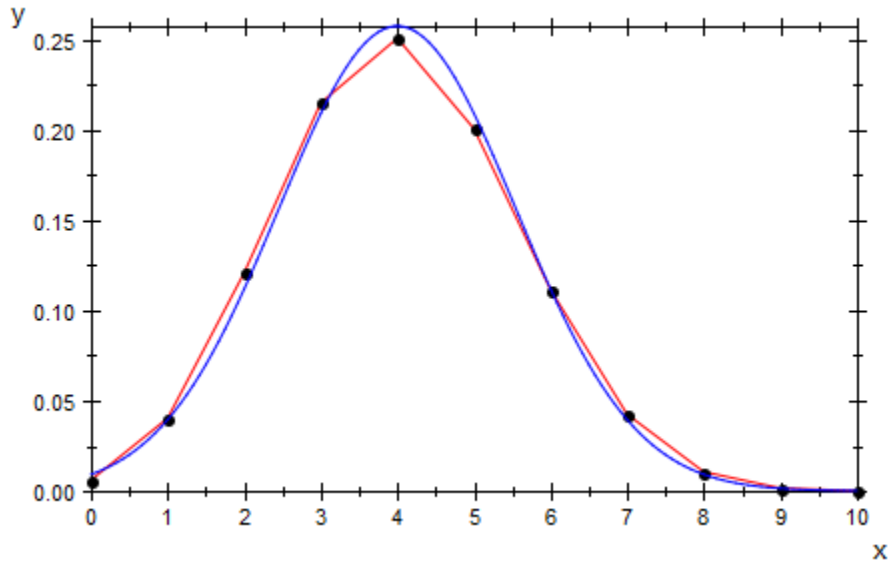


Example 3

A random variable describing the number of successes in n Bernoulli trials with success probability p is binomially distributed with expectation value $n\,p$ and variance $n\,p\,(1 - p)$. For large values of n, the binomial distribution is approximated by a corresponding normal distribution.

We use `plot::Listplot` to visualize the discrete probability values of the binomial distribution. The normal distribution is visualized via `plot::Function2d`:

```
n := 10: p:= 0.4:
plot(plot::Listplot([stats::binomialPF(n, p)(i) $ i = 0..n],
                    x = 0..n, Color = RGB::Red),
     plot::Function2d(stats::normalPDF(n*p, n*p*(1 - p))(x),
                      x = 0..n, Color = RGB::Blue)):
```




```
delete n, p:
```

Parameters

y_1, y_2, …

Vertical coordinates: numerical values or expressions of the animation parameter `a`.

y_1, y_2, … is equivalent to the attribute `Data`.

x

Name of the horizontal coordinate: an identifier or an indexed identifier. It is used as the title of the coordinate axis in *x* direction.

x is equivalent to the attribute `XName`.

x_{min} .. x_{max}

The range of the horizontal coordinate: `xmin`, `xmax` must be numerical real value or expressions of the animation parameter `a`.

x_{min} .. x_{max} is equivalent to the attributes `XRange`, `XMin`, `XMax`.

A_1

A 1-dimensional array of domain type `DOM_ARRAY` or a matrix of category `Cat::Matrix` (e.g., of type `matrix` or `densematrix`) with 1 row or 1 column. The entries must be numerical real values or arithmetical expressions of the animation parameter `a`. The entries in A_1 are regarded as data values $[y_1, y_2]$ etc..

A_1 is equivalent to the attribute `Data`.

x_1, x_2, …

Horizontal coordinates: numerical values or expressions of the animation parameter `a`.

A_2

A 2-dimensional array of domain type `DOM_ARRAY` or a matrix of category `Cat::Matrix` (e.g., of type `matrix` or `densematrix`) with at least two rows and two columns. The entries must be numerical real values or arithmetical expressions of the animation parameter `a`. The i-th row is regarded as the data point (x_i, y_i). If more than 2 columns are provided, only the data in the first two columns are considered; all additional columns are ignored.

A_2 is equivalent to the attribute `Data`.

a

Animation parameter, specified as a = $a_{min}..a_{max}$, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::PointList2d` | `plot::Polygon2d` | `plot::Scatterplot`

plot::Lsys

Lindenmayer systems

Syntax

```
plot::Lsys(alpha, start, trans, …, <a = a_min .. a_max>, options)
```

Description

`plot::Lsys` creates Lindenmayer systems, i.e., string rewriting systems controlling turtle graphics.

Lindenmayer systems, or L-systems for short, are based on the concept of iteratively transforming a string of symbols into another string. After a finite number of iterations, the resulting string is translated into a sequence of movement commands to a "turtle" (see `plot::Turtle`), which can be drawn on the screen.

In `plot::Lsys`, the string of symbols is represented by a string of characters, i.e., a `DOM_STRING`. *Transformation rules* are given as equations mapping strings of length 1 to strings of arbitrary length. *Turtle rules* are given as equations mapping strings of length 1 to simple movement commands: `Line`, `Move`, `Left`, `Right`, `Push`, `Pop`, `Noop`, or a color specification.

The commands are mostly self-explanatory. `Left` and `Right` turn by the amount set in the slot `"RotationAngle"`; the initial direction is "up". `Line` and `Move` move by the amount set in `"StepLength"`, where `Move` does not draw a line. `Push` stores the current state (position, direction, color) on a stack from where it can later be reactivated using `Pop`. `Noop` means "ignore this, no operation". A color specification changes the line color.

The following turtle rules are used by default (but can be disabled by giving other rules for the left-hand sides):

`"F" = Line`, `"f" = Move`, `"[" = Push`, `"]" = Pop`, `"+" = Left`, `"-" = Right`.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`FALSE`
`Color`	the main color	`RGB::Blue`
`Frames`	the number of frames in an animation	`50`
`Generations`	number of iterations of L-system rules	`5`
`IterationRules`	iteration rules of an L-system	
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`LineColor`	color of lines	`RGB::Blue`
`LineWidth`	width of lines	`0.35`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LinesVisible`	visibility of lines	`TRUE`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	

Attribute	Purpose	Default Value
`ParameterRange`	range of the animation parameter	
`RotationAngle`	angle of rotation commands in L-systems	
`StartRule`	start rule of an L-system	
`StepLength`	length of movement commands in L-systems	`1.0`
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`
`Title`	object title	
`TitleFont`	font of object titles	`[" sans-serif ", 11]`
`TitlePosition`	position of object titles	
`TitleAlignment`	horizontal alignment of titles w.r.t. their coordinates	`Center`
`TitlePositionX`	position of object titles, x component	
`TitlePositionY`	position of object titles, y component	
`TurtleRules`	rules translating L-system symbols to turtle movements	
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	

Attribute	Purpose	Default Value
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

As a very simple system, we consider the following iteration rule: “replace each line forward by the sequence “line forward, move forward without painting, line forward.””:

```
l := plot::Lsys(0, "F", "F" = "FfF"):
```

Note that we do not provide an iteration rule for `"f"`. This means “leave `f` alone, do not change it.”

The start state is displayed by plotting the system after zero generations:

```
l::Generations := 0:
plot(l)
```

Increasing the number of generations, we see the effect of our transformation rule:

```
l::Generations := 1:
plot(l)
```

```
l::Generations := 2:
plot(l)
```

```
l::Generations := 3:
plot(l)
```

The following variant of this simple example produces approximations to the Cantor set:

```
l := plot::Lsys(0, "F", "F" = "FfF", "f" = "fff"):
plot(l)
```

Example 2

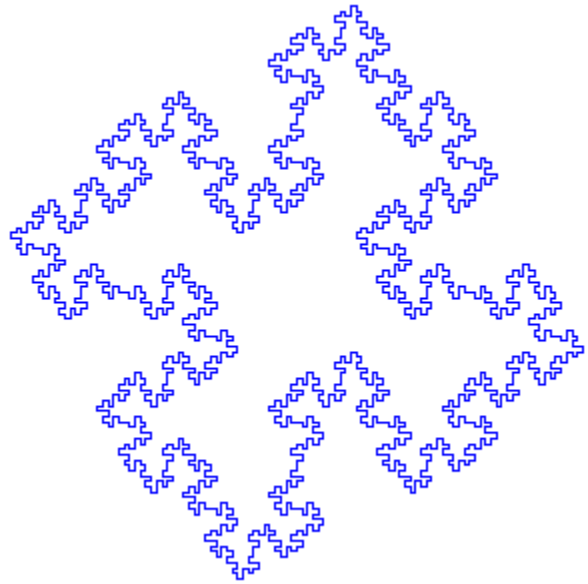
To get more interesting examples, we include rotations into our rules:

```
l := plot::Lsys(PI/2, "F-F-F-F", "F" = "F-F+F+FF-F-F+F",
                Generations = 3)
```

$$\text{plot::Lsys}\left(\frac{\pi}{2}, \text{"F-F-F-F"}, \text{IterationRules} = [\text{"F"} = \text{"F-F+F+FF-F-F+F"}], \text{TurtleRules} = [], \text{Generations} = 3\right)$$

As you can see, `plot::Lsys` has detected that our rule is an iteration rule. We could have used this syntax directly when creating the object. We have not given turtle rules, so the defaults are used:

```
plot(l)
```



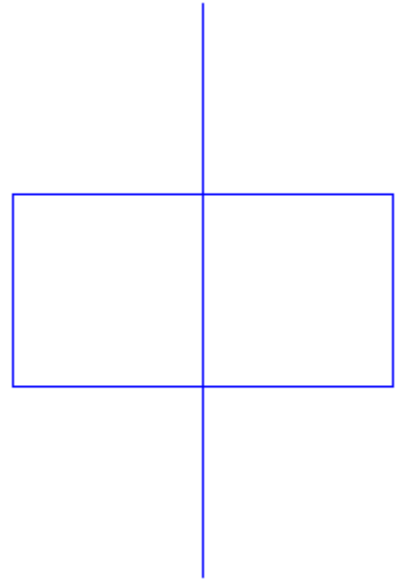
Example 3

The Peano curve is a famous example of a space filling curve. In the limit process, increasing the number of iterations while decreasing the length of the forward steps, it actually fills the plane. There are different constructions known, the one shown here fills a square tilted by 45°:

```
peano := plot::Lsys(PI/2, "F", "F" = "F+F-F-F-F+F+F+F-F"):
```

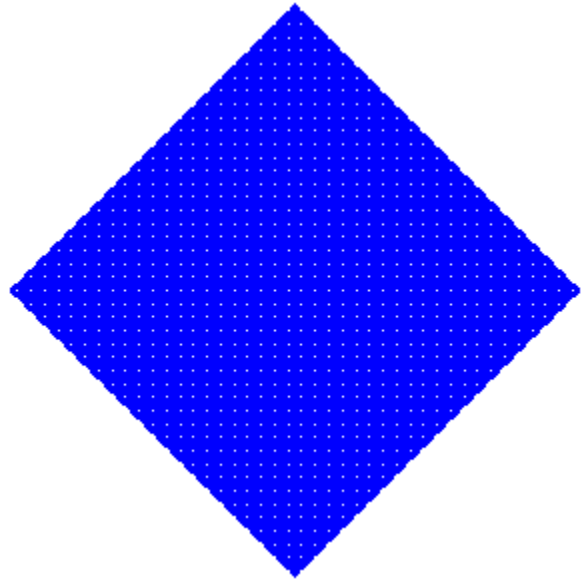
The transformation rule says to replace each straight line with the following construction:

```
peano::Generations := 1:
plot(peano)
```



After a few iterations, the lines already get very close to one another:

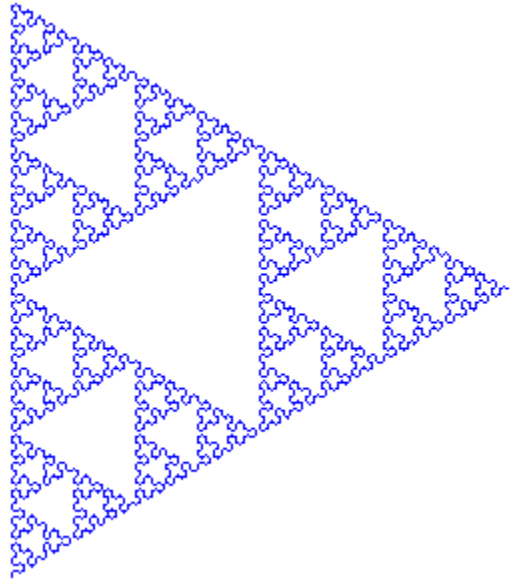
```
peano::Generations := 5:
plot(peano)
```



Example 4

Many L-systems contain different types of lines: While they are drawn exactly the same, their transformation rules are different from one another. The following example shows an image similar to the Sierpinski triangle:

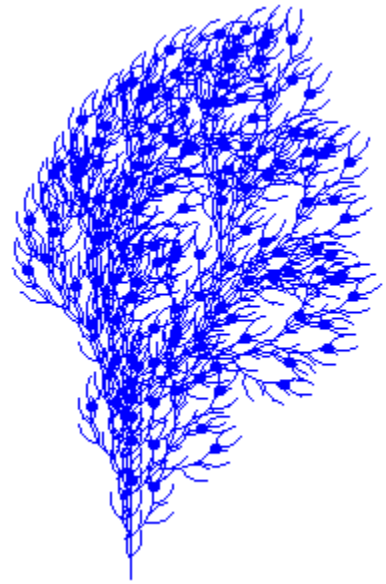
```
l := plot::Lsys(PI/3, "R", "L" = "R+L+R", "R" = "L-R-L",
                          "L" = Line, "R" = Line,
                Generations = 7):
plot(l)
```



Example 5

The `Push` and `Pop` operations can be used to draw “arms” in an L-system:

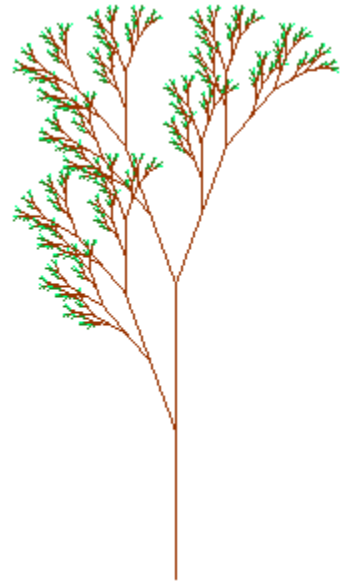
```
plot(plot::Lsys(23*PI/180, "F", "F" = "FF-[-F+F+F]+[+F-F-F]",
                Generations = 4))
```



Example 6

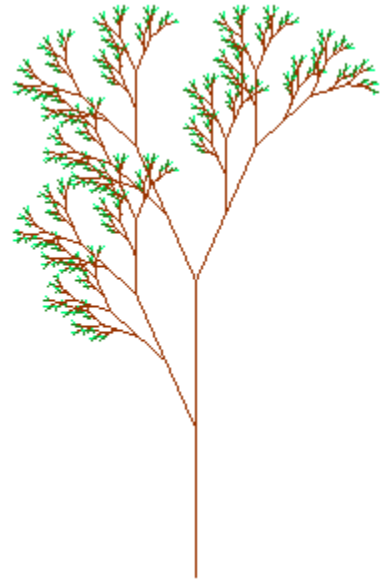
L-systems have been used to simulate plant growth. We show an example here that uses the symbols `B`, `H`, and `G` to change the color of lines:

```
l := plot::Lsys(PI/9, "BL", "L" = "BR[+HL]BR[-GL]+HL",
                "R" = "RR", "L" = Line, "R" = Line,
                "B" = RGB::Brown, "H" = RGB::ForestGreen,
                "G" = RGB::SpringGreen, Generations = 6):
plot(l)
```



The attribute `Generations` can be animated. This way, we can actually make the "plant" "grow:"

```
plot(plot::Lsys(a*PI/45, "BL", "L" = "BR[+HL]BR[-GL]+HL", "R" = "RR",
                "L" = Line, "R" = Line, "B" = RGB::Brown,
                "H" = RGB::ForestGreen, "G" = RGB::SpringGreen,
                 Generations = a, a = 1 .. 6)):
```



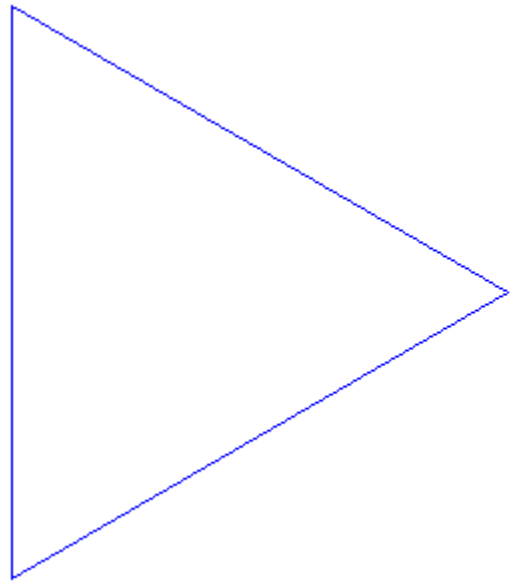
Example 7

L-systems can display a couple of popular fractals. One example is the Koch snowflake, generated by replacing each straight line with a straight line, followed by a left turn of $60°$, another straight line, a right turn of $120°$, another straight line, another left turn of $60°$ and a final straight line:

```
koch := plot::Lsys(PI/3, "F--F--F", "F" = "F+F--F+F"):
```

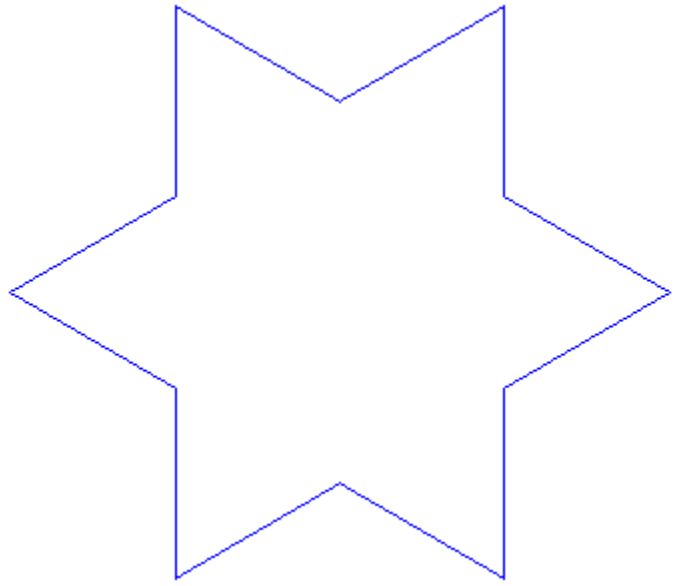
The starting rule has been chosen to be an equilateral triangle:

```
koch::Generations := 0:
plot(koch)
```



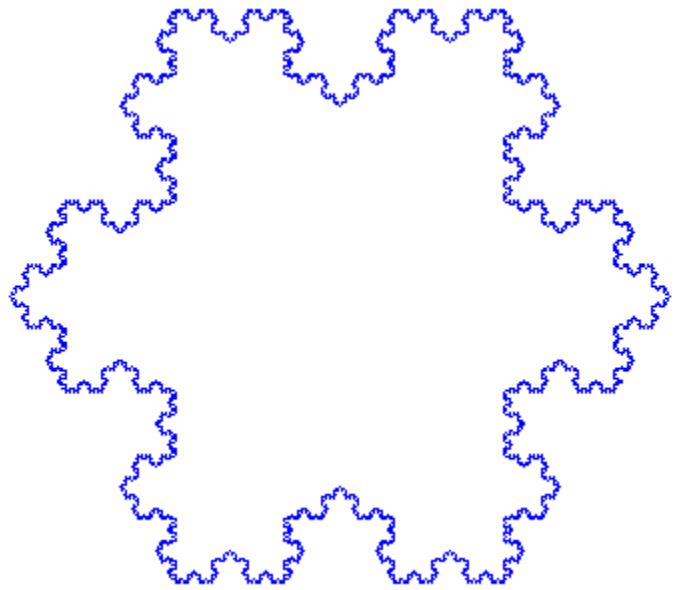
The first generation looks like this:

```
koch::Generations := 1:
plot(koch)
```



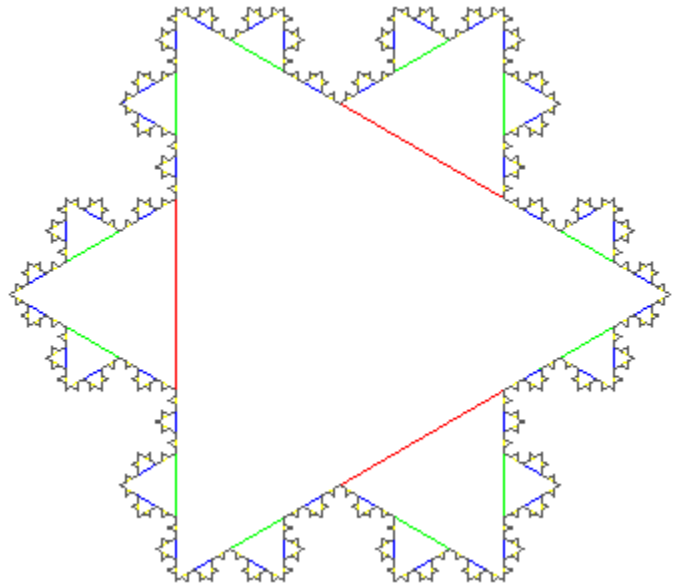
The limit is pretty well approximated after five generations:

```
koch::Generations := 5:
plot(koch)
```



Finally, we use `plot::modify` and the `"StepLength"` slot to show the first couple of iterations superimposed on one another:

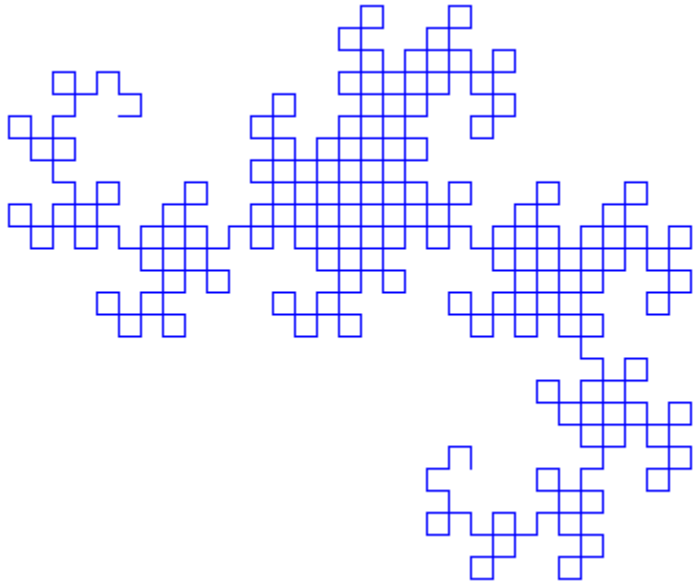
```
colors := [RGB::Red, RGB::Green, RGB::Blue, RGB::Yellow, RGB::DimGrey]:
plot(plot::modify(koch, Generations = i,
                  StepLength = 3^(-i),
                  LineColor = colors[i+1]) $ i = 0..4)
```



Example 8

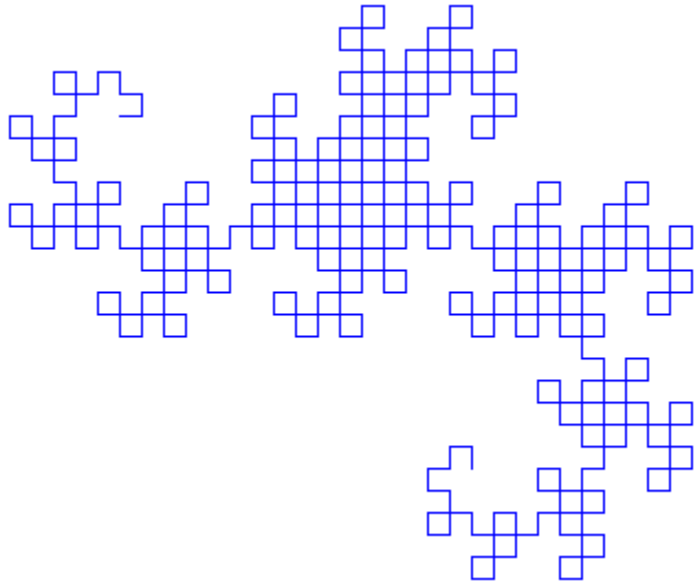
Another well-known example of a fractal generated by an L-system is Heighway's Dragon curve. Informally, it is generated by “drawing a right angle and then replacing each right angle by a smaller right angle” (Gardner). It has been used in the book “Jurassic Park” by Michael Crichton and thereby got another nickname, the “Jurassic Park fractal:”

```
plot(plot::Lsys(PI/2, "L", "L" = "L+R+", "R" = "-L-R",
                "L" = Line, "R" = Line, Generations = 9))
```



It is interesting to note that the iteration rules of this curve are equivalent to appending a mirrored copy of the curve to its end:

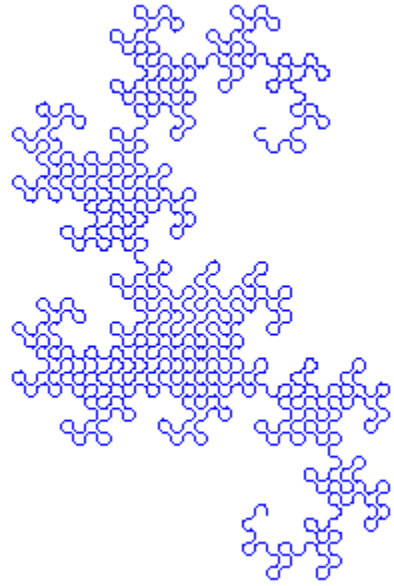
```
plot(plot::Lsys(PI/2, "L", "L" = "L+R+", "R" = "-L-R",
                "L" = Line, "R" = Line, Generations = a,
                a = 1..9))
```



Example 9

While the L-system of the previous example corresponds to the definition found in the literature, the images in at least one popular source show another system (while the definition given is the one from above), namely:

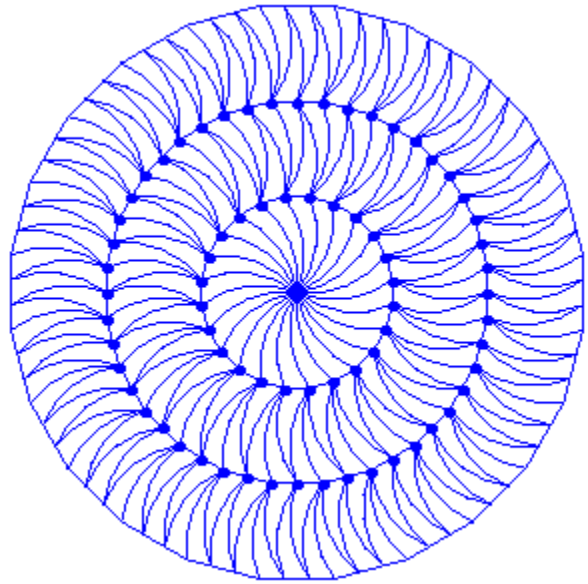
```
plot(plot::Lsys(PI/4, "X-F-Y", "X" = "X+F+Y", "Y" = "X-F-Y",
                "X" = Line, "Y" = Line, Generations = 9)):
```



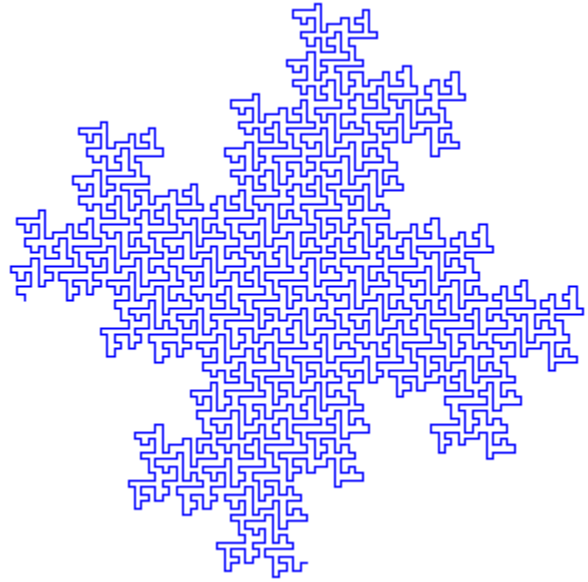
Example 10

An L-system may contain letters that are not meant to show in the final graphic, so they form some sort of “markers” for subsequent iteations. For this purpose, you may use the turtle rule `Noop`:

```
plot(plot::Lsys(PI/12,
                "X+X+X+X+X+X+X+X+X+X+X+X+X+X+X+X+X+X+X+X+X+X+X+X+X",
                "X" = "[F+F+F+F[---X-Y]+++++F++++++++F-F-F-F]",
                "Y" = "[F+F+F+F[---Y]+++++F++++++++F-F-F-F]",
                "X" = Noop, "Y" = Noop,
                Generations = 3))
```



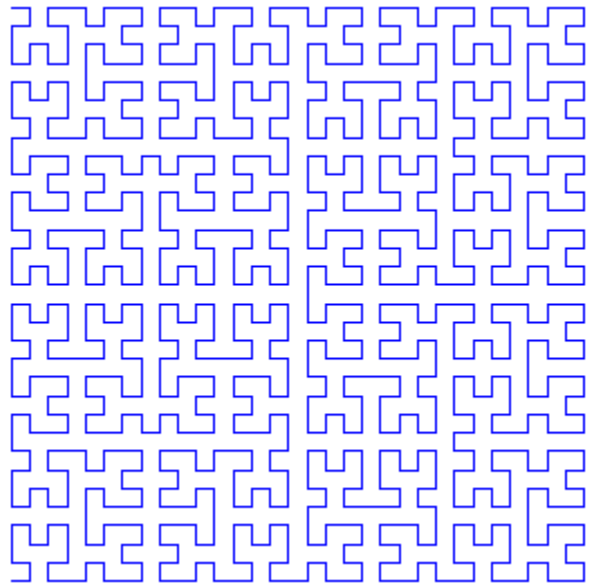
```
plot(plot::Lsys(PI/2, "FB",
                "A" = "FBFA+HFA+FB-FA", "B" = "FB+FA-FB-JFBFA",
                "F" = "", "H" = "-", "J" = "+",
                "A" = Noop, "B" = Noop, "H" = Noop, "J" = Noop))
```



Example 11

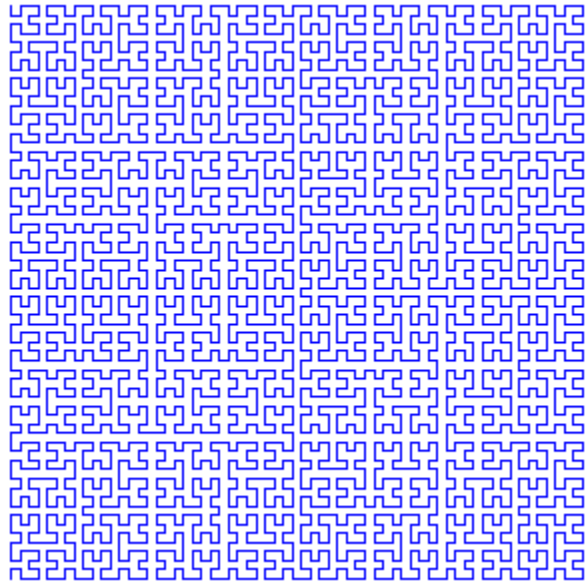
Using this rule, we can use the following formulation of the popular Hilbert curve due to Ken Philip:

```
plot(plot::Lsys(PI/2, "x", "x" = "-yF+xFx+Fy-", "y" = "+xF-yFy-Fx+",
                "x" = Noop, "y" = Noop))
```



To animate the creation process of the Hilbert curve, we adjust the length of the lines to the current number of iteration steps:

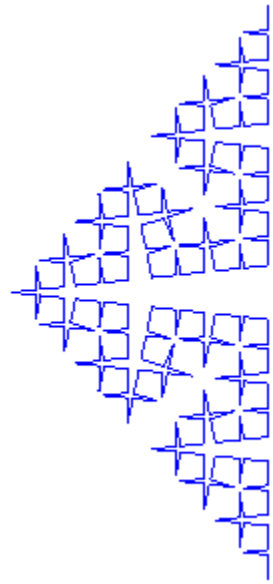
```
plot(plot::Lsys(PI/2, "x", "x" = "-yF+xFx+Fy-", "y" = "+xF-yFy-Fx+",
                "x" = Noop, "y" = Noop,
                Generations = i, StepLength = 1/(2^i-1),
                i = 1..6, Frames = 6))
```



Example 12

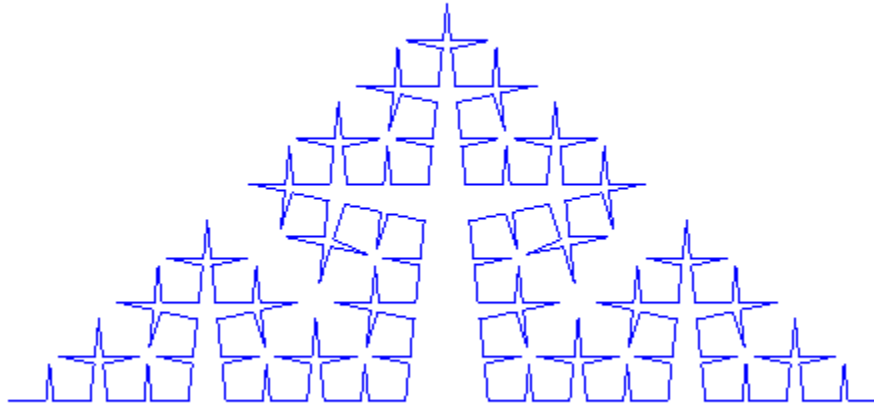
In some cases, systems will need small angles and long strings in order to specify the desired directions. Take for example the following system:

```
plot(plot::Lsys(7*PI/15, "F", "F"="F+F--F+F",
                Generations=4))
```



The rotations to the right use an angle of $\frac{7\,\pi}{15}$, while that to the left (the sharp spike) is a turn of $\frac{14\,\pi}{15}$. It would look more natural, however, to have the turtle start to the right, i.e., at an angle of $-\frac{\pi}{2}$. Since no multiple of $\frac{7\,\pi}{15}$ is equal to $\frac{\pi}{2}$ modulo $2\,\pi$, this requires that we use a smaller angle, adjusting our iteration rule:

```
plot(plot::Lsys(7*PI/30,"+++++++++++++++F",
                "F"="F++F----F++F", Generations=4))
```



Parameters

`alpha`

Angle (in radians) for turning commands. Animatable.

alpha is equivalent to the attribute `RotationAngle`.

`start`

String used as the starting rule.

start is equivalent to the attribute `StartRule`.

`trans, …`

Iteration and Turtle command rules (see below).

trans, … is equivalent to the attributes `IterationRules`, `TurtleRules`.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

Algorithms

Lindenmayer systems are “string rewriting systems.” MuPAD implements only context-free L-systems, which are analyzed in a similar context as context-free grammars.

Many examples of L-systems can be found, among other places, in “The Fractal Geometry of Nature” by Benoît Mandelbrot.

See Also

MuPAD Functions
`plot | plot::copy`

MuPAD Graphical Primitives
`plot::Turtle`

plot::Matrixplot

Surface plot of matrix data

Syntax

```
plot::Matrixplot(A, options)
plot::Matrixplot(A, x = x_min .. x_max, y = y_min .. y_max, <a = a_min .. a_max>, options)
plot::Matrixplot(row_1, row_2, …, options)
plot::Matrixplot(row_1, row_2, …, x = x_min .. x_max, y = y_min .. y_max, <a = a_min .. a_max>, optio
plot::Matrixplot([row_1, row_2, …], options)
plot::Matrixplot([row_1, row_2, …], x = x_min .. x_max, y = y_min .. y_max, <a = a_min .. a_max>, opt
plot::Matrixplot(s, <c_1, c_2, …>, options)
plot::Matrixplot(s, <c_1, c_2, …>, x = x_min .. x_max, y = y_min .. y_max, <a = a_min .. a_max>, opti
plot::Matrixplot(s, <[c_1, c_2, …]>, options)
plot::Matrixplot(s, <[c_1, c_2, …]>, x = x_min .. x_max, y = y_min .. y_max, <a = a_min .. a_max>, op
```

Description

`plot::Matrixplot(A)` visualizes the matrix A as a 3D function graph by interpolating the matrix values as a function of the matrix indices.

`Matrixplot` interprets the indices of a matrix as x and y coordinates and the corresponding matrix entry as the corresponding z coordinate. Thus, the matrix is regarded as a discretized function in 2 variables. The function graph is displayed as a 3D surface using interpolation between the data points.

If no ranges `x = `x_{min}` .. `x_{max}`, y = `y_{min}` .. `y_{max}`` are specified, the matrix entry `A[i, j]` is diplayed as the 3D point $x = j$, $y = i$, $z =$ `A[i, j]` with integer positions i, j. If plot ranges are specified, the matrix indices i, j are used to define an equidistant mesh in the plot range.

The attribute `InterpolationStyle` allows to define the surface via linear or cubic spline interpolation of the data points: Choose between `InterpolationStyle` = `Linear` or `InterpolationStyle` = `Cubic`. The default is linear interpolation. With cubic interpolation, the data surface may be smoothened by setting the numbers m_x, m_y of plot points between the data points via the attribute `Submesh` = `[`m_x`,` m_y`]`. The numbers m_x, m_y must be (small) non-negative integers.

With `InterpolationStyle` = `Linear`, symbolic values and complex numbers are accepted and ignored, leading to gaps in the surface. With `InterpolationStyle` = `Cubic`, symbolic values or complex numbers lead to an error. Cf. “Example 4” on page 24-547.

Per default, the data points are rendered on the surface. Use `PointsVisible` = `FALSE` to make them disappear.

Animations are triggered by specifying a range `a = ``a_{min}`` .. ``a_{max}`` ` for a parameter `a` that is different from the variables `x`, `y`. Thus, in animations, both the ranges `x = ``x_{min}`` .. ``x_{max}`` `, `y = ``y_{min}`` .. ``y_{max}`` ` as well as the animation range `a = ``a_{min}`` .. ``a_{max}`` ` must be specified.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Color`	the main color	`RGB::Red`
`Data`	the (statistical) data to plot	
`Filled`	filled or transparent areas and surfaces	`TRUE`
`FillColor`	color of areas and surfaces	`RGB::Red`
`FillColor2`	second color of areas and surfaces for color blends	`RGB::CornflowerBlue`
`FillColorType`	surface filling types	`Dichromatic`
`FillColorFunction`	functional area/surface coloring	

Attribute	Purpose	Default Value
FillColorDirection	the direction of color transitions on surfaces	[0, 0, 1]
FillColorDirectionX	x-component of the direction of color transitions on surfaces	0
FillColorDirectionY	y-component of the direction of color transitions on surfaces	0
FillColorDirectionZ	z-component of the direction of color transitions on surfaces	1
Frames	the number of frames in an animation	50
InterpolationStyle	interpolation via linear or cubic splines	Linear
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Black.[0.25]
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0

Attribute	Purpose	Default Value
LineColorDirectionY	y-component of the direction of color transitions on lines	0
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointColor	the color of points	RGB::MidnightBlue
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	TRUE
Shading	smooth color blend of surfaces	Smooth
Submesh	density of submesh (additional sample points)	[2, 2]
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	

Attribute	Purpose	Default Value
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XLinesVisible	visibility of parameter lines (x lines)	TRUE
XMax	final value of parameter “x”	
XMin	initial value of parameter “x”	
XName	name of parameter “x”	
XRange	range of parameter “x”	
XSubmesh	density of additional sample points for parameter “x”	2
YLinesVisible	visibility of parameter lines (y lines)	TRUE
YMax	final value of parameter “y”	

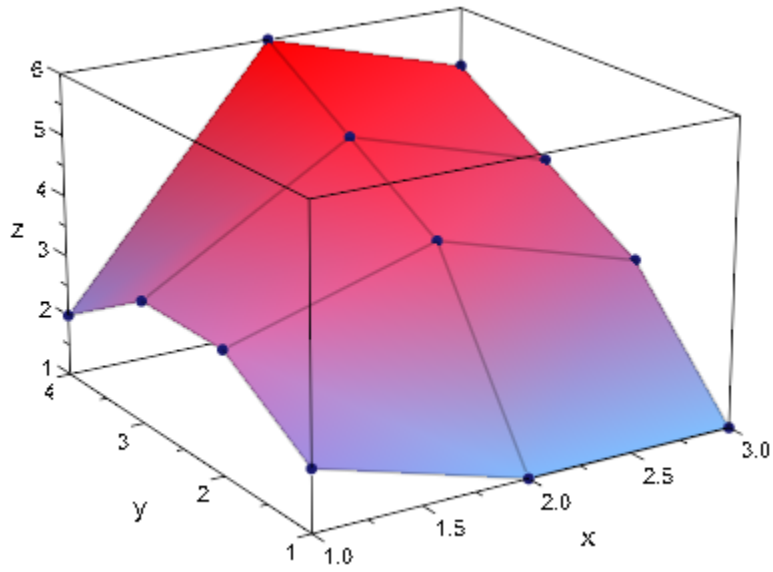
Attribute	Purpose	Default Value
YMin	initial value of parameter "y"	
YName	name of parameter "y"	
YRange	range of parameter "y"	
YSubmesh	density of additional sample points for parameter "y"	2

Examples

Example 1

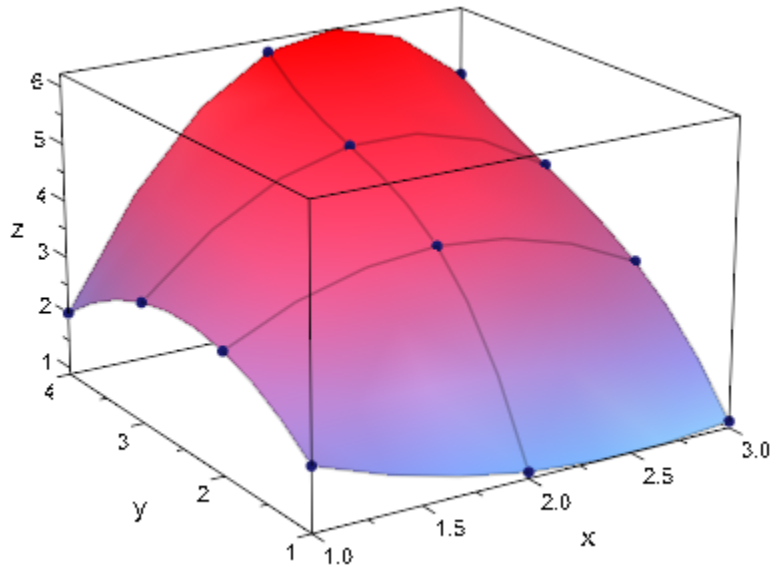
This example demonstrates the general calling syntax. The data are passed in different ways using a list of rows, an array, and a matrix, respectively:

```
A := [[2, 1, 1],
      [3, 4, 3],
      [3, 5, 4],
      [2, 6, 5]]:
plot(plot::Matrixplot(A))
```



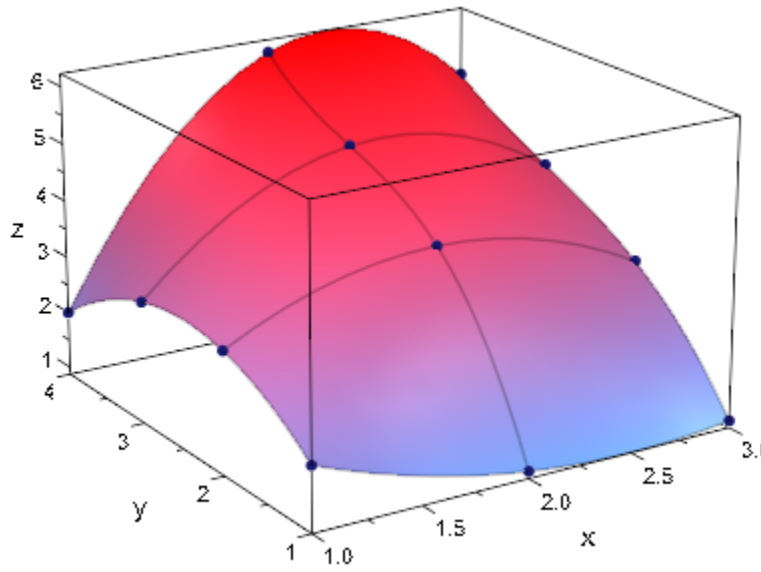
With `InterpolationStyle` = `Cubic`, the matrix data are plotted as a cubic spline surface:

```
A := array(1..4, 1..3, A):
plot(plot::Matrixplot(A, InterpolationStyle = Cubic)):
```



The spline surface can be smoothened by using the `Submesh` attribute to add further evaluation points:

```
A := matrix(A):
plot(plot::Matrixplot(A, Submesh = [6, 6],
                      InterpolationStyle = Cubic)):
```

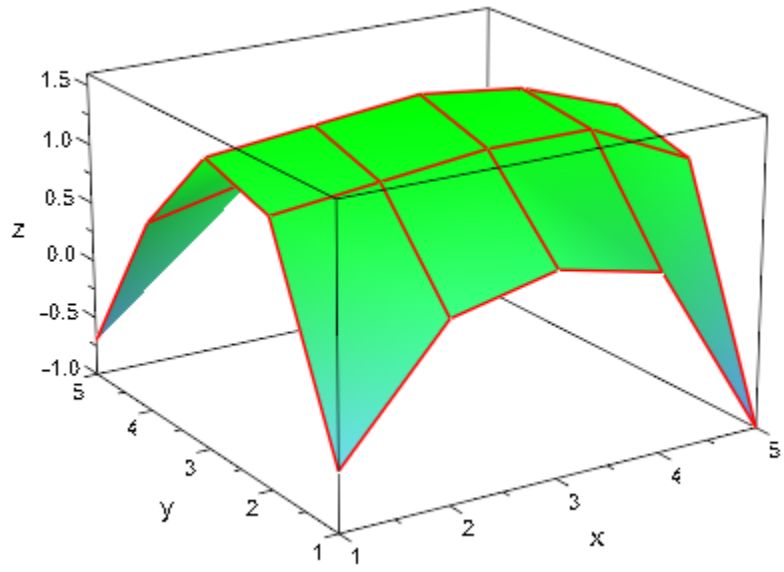



```
delete A:
```

Example 2

Various plot attributes can be specified:

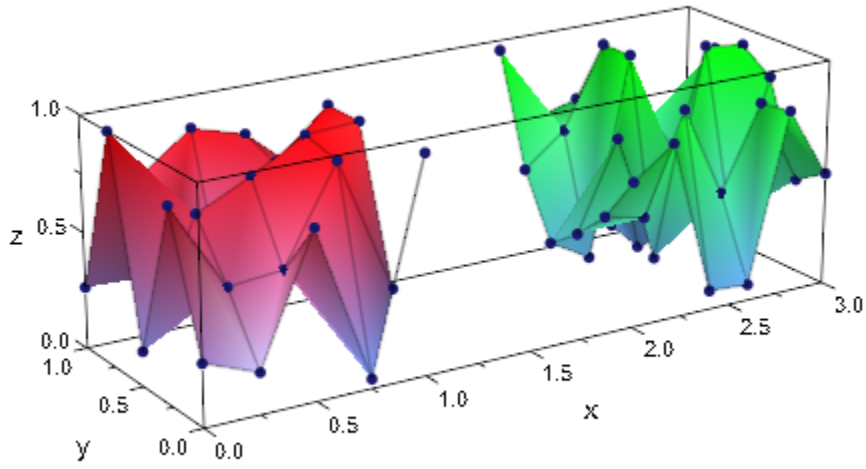
```
plot(plot::Matrixplot(
   [[-0.5,   0.5, 0.7, 0.5, -1  ],
    [ 1.2, 1.3, 1.4, 1.4,  1  ],
    [ 1.4, 1.5, 1.6, 1.5,  1.2],
    [ 0.6, 0.8, 1,   1,    1  ],
    [-0.7, 0.5, 0.5, 0,   -1  ]],
    PointsVisible = FALSE,
    FillColor = RGB::Green,
    LineColor = RGB::Red))
```



Example 3

Choosing appropriate coordinate ranges, we place two matrix plots side by side:

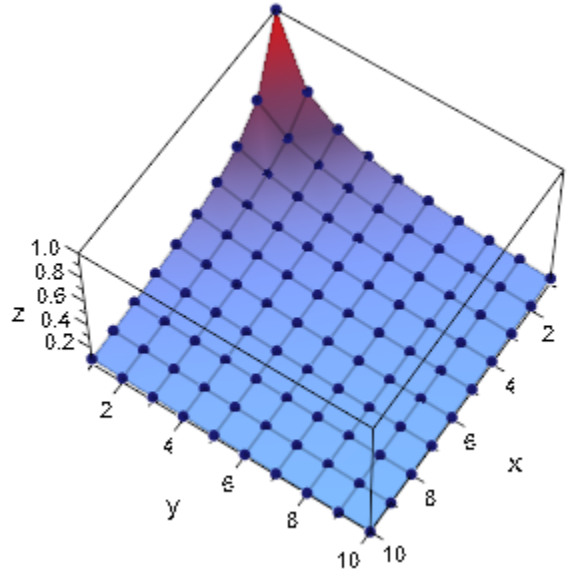
```
plot(plot::Matrixplot(matrix::random(5, 5, frandom),
                      x = 0..1, y = 0..1,
                      Color = RGB::Red),
     plot::Matrixplot(matrix::random(6, 6, frandom),
                      x = 2..3, y = 0..1,
                      Color = RGB::Green),
     Scaling = Constrained)
```



Example 4

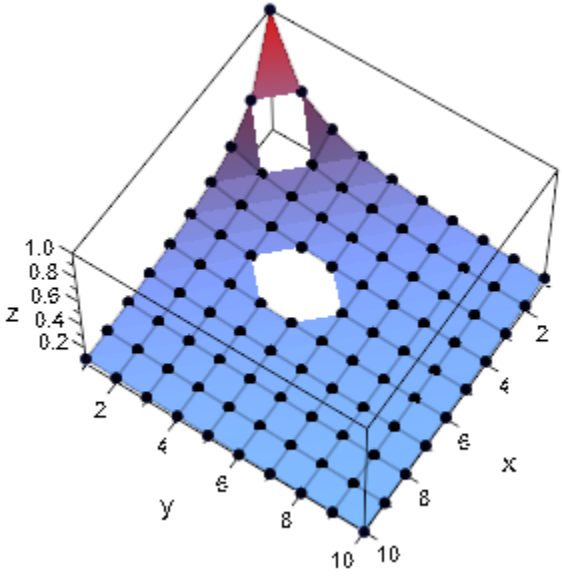
We plot a Hilbert matrix:

```
A := linalg::hilbert(10):
plot(plot::Matrixplot(A), CameraDirection = [3, 2, 1])
```



Some of the entries are replaced by values that cannot be plotted. Consequently, the plot contains holes:

```
A[2, 2] := NIL:
A[4, 5] := infinity:
A[5, 5] := x:
plot(plot::Matrixplot(A), CameraDirection = [3, 2, 1])
```



With `InterpolationStyle` = `Cubic`, an error is raised:

```
plot(plot::Matrixplot(A, InterpolationStyle = Cubic))
```

```
Error: Data contains nonreal numeric values. Use 'Style = Linear' to plot matrices cont
  Evaluating: plot
```

```
delete A:
```

Parameters

A

A matrix of category `Cat::Matrix` or an array containing real numerical values or expressions of the animation parameter `a`.

A is equivalent to the attribute `Data`.

row_1, row_2, …

The matrix rows: each row must be a list of real numerical values or expressions of the animation parameter `a`. All rows must have the same length.

row_1, row_2, … is equivalent to the attribute `Data`.

`s`

A data sample of domain type `stats::sample`.

s is equivalent to the attribute `Data`.

c_1, c_2, …

Column indices of `s`: positive integers. These indices, if given, indicate that only the specified columns should be used. The indexed columns must contain real numerical values or expressions of the animation parameter `a`, If no columns are specified, all columns of `s` are used.

`x`

Name of the first coordinate: an identifier or an indexed identifier. It is used as the title of the coordinate axis in *x* direction.

x is equivalent to the attribute `XName`.

x_{min} .. x_{max}

The range of the first coordinate: x_{min}, x_{max} must be numerical real value or expressions of the animation parameter *a*.

x_{min} .. x_{max} is equivalent to the attributes `XRange`, `XMin`, `XMax`.

`y`

Name of the second coordinate: an identifier or an indexed identifier. It is used as the title of the coordinate axis in *y* direction.

y is equivalent to the attribute `YName`.

y_{min} .. y_{max}

The range of the second coordinate: y_{min}, y_{max} must be numerical real value or expressions of the animation parameter *a*.

y_{min} .. y_{max} is equivalent to the attributes `YRange`, `YMin`, `YMax`.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Function3d` | `plot::Surface`

plot::MuPADCube

"the cube" (the logo)

Syntax

```
plot::MuPADCube(<r>, <[c_x, c_y, c_z]>, <a = a_min .. a_max>, options)
```

Description

`plot::MuPADCube()` creates the "MuPAD cube" as a graphical 3D primitive.

This object only exists for demonstration purposes.

`plot::MuPADCube` accepts the attribute `Colors` which defines the colors of the spheres and the cylinders between the spheres. Its value is a list of RGB or RGBa colors:

- The color list may contain one to four values determining the colors of the spheres.
- If a 5th color is given, it determines the color of the cylinders.
- If the list contains nine colors (not less), the first and last four determine the colors of the 8 spheres. The fifth color determines the color of the cylinders.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Center`	center of objects, rotation center	[0, 0, 0]
`CenterX`	center of objects, rotation center, x-component	0
`CenterY`	center of objects, rotation center, y-component	0
`CenterZ`	center of objects, rotation center, z-component	0

Attribute	Purpose	Default Value
Colors	list of colors to use	[RGB::Green, RGB::Blue, RGB::Red, RGB::Yellow, RGB::Antique]
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
Radius	radius of circles, spheres etc.	1
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center

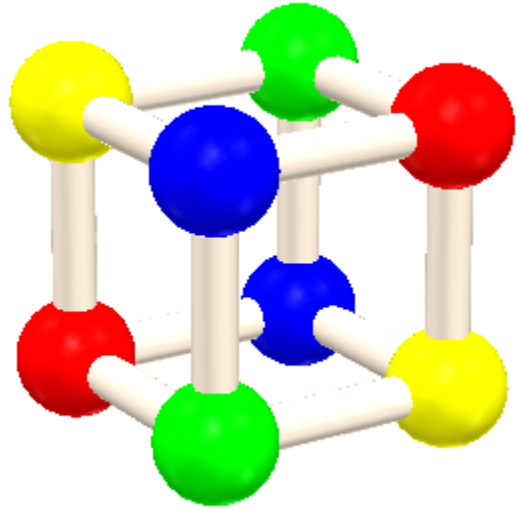
Attribute	Purpose	Default Value
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE

Examples

Example 1

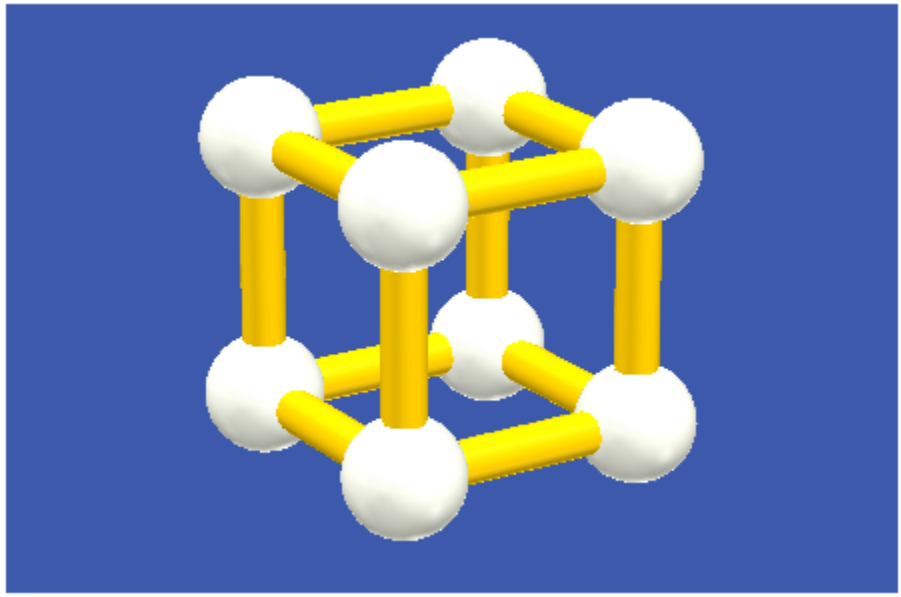
The MuPAD logo:

```
plot(plot::MuPADCube())
```



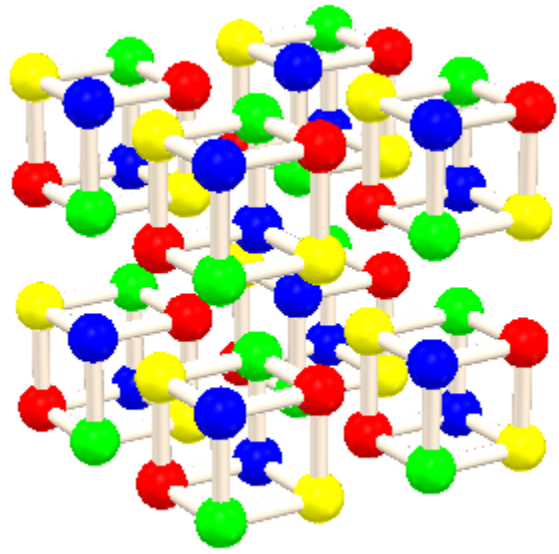
The MuPAD logo with strange colors:

```
plot(plot::MuPADCube(Colors = [RGB::Titanium $ 4, RGB::Gold],
                     BackgroundColor = RGB::Cobalt))
```



A collection of “MuPAD cubes”:

```
plot(plot::MuPADCube(Center = [2*k, 2*l, 2*m])
                     $ k = 0..1 $ l = 0..1 $ m = 0..1)
```



The MuPAD logo with animated size:

```
plot(plot::MuPADCube(1 - abs(a), [0, 0, 0], a = -1..1))
```

Parameters

r

The size of the object (the radius of the surrounding sphere): a real numerical value or an arithmetical expression of the animation parameter `a`. The default value of the radius is `1`.

r is equivalent to the attribute `Radius`.

c_x, c_y, c_z

The coordinates of the center: real numerical values or arithmetical expressions of the animation parameter `a`. By default, a cube centered at the origin is created.

c_x, c_y, c_z are equivalent to the attributes `CenterX`, `CenterY`, `CenterZ`.

a

Animation parameter, specified as a `= a`$_{min}$`..a`$_{max}$, where `a`$_{min}$ is the initial parameter value, and `a`$_{max}$ is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Cone` | `plot::Cylinder` | `plot::Sphere`

plot::Ode2d

2D plots of ODE solutions

Syntax

```
plot::Ode2d(f, [t0, t1, …], Y0, <[G1, <Style = style1>, <Color = c1>], [G2, <Style = styl
plot::Ode2d(f, [Automatic, tstart, tend, tstep], Y0, <[G1, <Style = style1>, <Color = c1>],
plot::Ode2d([t0, t1, …], f, Y0, <[G1, <Style = style1>, <Color = c1>], [G2, <Style = styl
plot::Ode2d([Automatic, tstart, tend, tstep], f, Y0, <[G1, <Style = style1>, <Color = c1>],
```

Description

`plot::Ode2d(f, [t0, t1, …], Y0)` renders two-dimensional projections of the solutions of the initial value problem given by f, t_0 and Y_0.

`plot::Ode2d(f, [t0, t1, …], Y0, [G])` computes a mesh of numerical sample points $Y(t_0)$, $Y(t_1)$, … representing the solution $Y(t)$ of the first order differential equation (dynamical system)

$$\frac{\partial}{\partial t} Y = f(t, Y),\ Y(t_0) = Y_0,\ t_0,\ t \in \mathbb{R},\ Y_0,\ Y(t) \in \mathbb{C}^n$$

.

The procedure

$$G\colon (t, Y) \to [x(t, Y), y(t, Y)] \quad \text{or} \quad G\colon (t, Y) \to [x(t, Y), y(t, Y), z(t, Y)]$$

maps these solution points $(t_i, Y(t_i))$ in $\mathbb{R}\times\mathbb{C}^n$ to a mesh of 2D plot points $[x_i, y_i]$. These points can be connected by straight lines or interpolating splines.

The calling syntax of `plot::Ode2d` and `plot::Ode3d` as well as the functionality of these two procedures is identical. The only difference is that `plot::Ode2d` expects graphical generators G_1, G_2 etc. that produce graphical 2D points, whereas `plot::Ode3d` expects graphical generators producing 3D points.

Internally, a sequence of numerical sample points

`Y_1 := numeric::odesolve(f, t_0..t_1, Y_0, Options)`,

`Y_2 := numeric::odesolve(f, t_1..t_2, Y_1, Options)` etc.

is computed, where `Options` is some combination of `method`, `RelativeError = rtol`, `AbsoluteError = atol`, and `Stepsize = h`. See `numeric::odesolve` for details on the vector field procedure `f`, the initial condition Y_0, and the options.

The utility function `numeric::ode2vectorfield` may be used to produce the input parameters `f, t_0, Y_0` from a set of differential expressions representing the ODE. Cf. "Example 1" on page 24-565.

Each of the "generators of plot data" G_1, G_2 etc. creates a graphical solution curve from the numerical sample points Y_0, Y_1 etc. Each generator `G`, say, is internally called in the form `G(t_0, Y_0), G(t_1, Y_1), ...` to produce a sequence of plot points in 2D.

The solver `numeric::odesolve` returns the solution points Y_0, Y_1 etc. as lists or 1-dimensional arrays (the actual type is determined by the initial value Y_0). Consequently, each generator `G` must accept two arguments `(t, Y)`: `t` is a real parameter, `Y` is a "vector" (either a list or a 1-dimensional array).

Each generator must return a list with 2 elements representing the (*x*, *y*) coordinates of the graphical point associated with a solution point `(t, Y)` of the ODE.

All generators must produce graphical data of the same dimension, i.e., 2D data as lists with 2 elements for `plot::Ode2d`.

Some examples:

`G := (t, Y) -> [t, Y_1]` creates a 2D plot of the first component of the solution vector along the *y*-axis, plotted against the time variable `t` along the *x*-axis

`G := (t, Y) -> [Y_1, Y_2]` creates a 2D phase plot, plotting the first component of the solution along the *x*-axis and the second component along the *y*-axis. The result is a solution curve in phase space (parametrized by the time `t`).

If no generators are given, `plot::Ode2d` by default plots all components of the solution as functions of time, using `[Splines, Points]` as the style.

Note that arbitrary values associated with the solution curve may be displayed graphically by an appropriate generator `G`. See "Example 2" on page 24-567 and "Example 5" on page 24-571.

Several generators `G1, G2` etc. can be specified to generate several curves associated with the same numerical mesh `Y0, Y1, ...`. See "Example 1" on page 24-565, "Example 2" on page 24-567, and "Example 3" on page 24-569.

The graphical data produced by each of the generators `G1, G2` etc. consists of a sequence of mesh points in 2D or 3D, respectively.

With `Style = Points`, the graphical data are displayed as a discrete set of points.

With `Style = Lines`, the graphical data points are displayed as a curve consisting of straight line segments between the sample points. The points themselves are not displayed.

With `Style = Splines`, the graphical data points are displayed as a smooth spline curve connecting the sample points. The points themselves are not displayed.

With `Style = [Splines, Points]` and `Style = [Lines, Points]`, the effects of the styles used are combined, i.e., both the evaluation points and the straight lines or splines, respectively, are displayed.

The plot attributes accepted by `plot::Ode2d,Ode3d` include `Submesh = n`, where n is some positive integer. This attribute only has an effect on the curves which are returned for the graphical generators with `Style = Splines` and `Style = [Splines, Points]`, respectively. It serves for smoothening the graphical spline curve using a sufficiently high number of plot points.

n is the number of plot points between two consecutive numerical points corresponding to the time mesh. The default value is $n = 4$, i.e., the splines are plotted as 5 straight line segments connecting the numerical sample points.

Attributes

Attribute	Purpose	Default Value
`AbsoluteError`	maximal absolute discretization error	
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`

Attribute	Purpose	Default Value
Colors	list of colors to use	[RGB::Blue, RGB::Red, RGB::Green, RGB::MuPADGold, RGB::Orange, RGB::Cyan, RGB::Magenta, RGB::LimeGreen, RGB::CadmiumYellowLight, RGB::AlizarinCrimson, RGB::Aqua, RGB::Lavender, RGB::SeaGreen, RGB::AureolineYellow, RGB::Banana, RGB::Beige, RGB::YellowGreen, RGB::Wheat, RGB::IndianRed, RGB::Black]
Frames	the number of frames in an animation	50
Function	function expression or procedure	
InitialConditions	initial conditions of the ODE	
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineWidth	width of lines	0.35
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
Name	the name of a plot object (for browser and legend)	

Attribute	Purpose	Default Value
ODEMethod	the numerical scheme used for solving the ODE	DOPRI78
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	TRUE
Projectors	project an ODE solution to graphical points	
RelativeError	maximal relative discretization error	
Stepsize	set a constant step size	
Submesh	density of submesh (additional sample points)	4
TimeEnd	end time of the animation	10.0
TimeMesh	the numerical time mesh	
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center

Attribute	Purpose	Default Value
`TitlePositionX`	position of object titles, x component	
`TitlePositionY`	position of object titles, y component	
`USubmesh`	density of additional sample points for parameter “u”	4
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

The following procedure `f` together with the initial value `Y0` represent the initial value problem $\frac{\partial}{\partial t} Y = f(t, Y) = t\,Y - Y^2$, $Y(0) = 2$. In MuPAD, the 1-dimensional vector Y is represented by a list with one element. The body of the function `f` below addresses the first (and only) entry of this list as Y_1 and returns the 1-dimensional vector $t\,Y$ - Y^2 as a list with one element. Also the initial condition Y_0 is a 1-dimensional vector represented by a list. For details on the format of `f`, please see `numeric::odesolve`:

```
f := (t, Y) -> [t*Y[1] - Y[1]^2]:
Y0 := [2]:
```

Alternatively, the utility function `numeric::ode2vectorfield` can be used to generate the input parameters in a more intuitive way:

```
[f, t0, Y0] := [numeric::ode2vectorfield(
      {y'(t) = t*y(t) - y(t)^2, y(0) = 2}, [y(t)])]
```

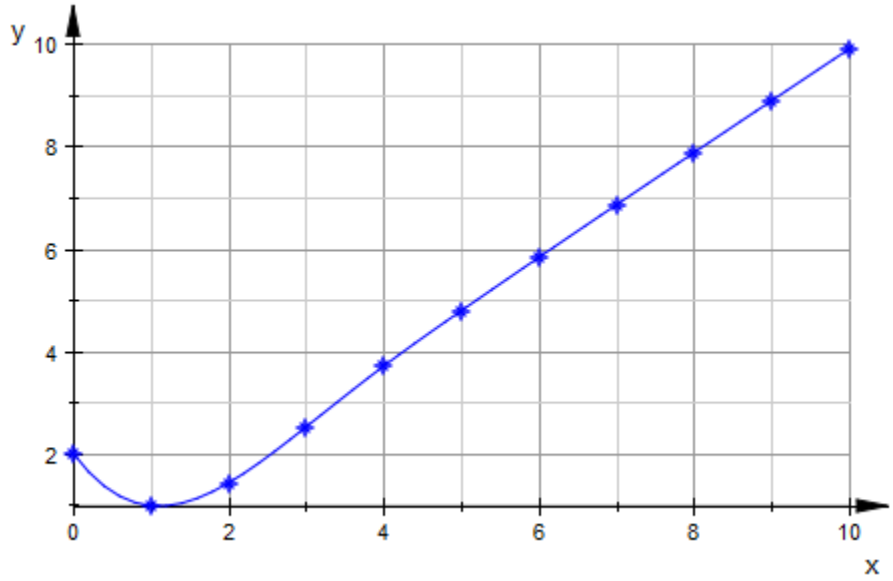
```
[proc f(t, Y) ... end, 0, [2]]
```

The numerical solution is to consist of sample points over the time mesh $t_i = i$, $i = 0, 1, \ldots, 10$. We use the default generator of `plot::Ode2d`. This generates the sample points together with a smooth spline curve connecting these points:

```
p := plot::Ode2d(f, [$ 0..10], Y0,
                 PointSize = 2*unit::mm,
                 PointStyle = Stars):
```

Finally, the ode solution is rendered by a call to `plot`:

```
plot(p, TicksDistance = 2.0, GridVisible = TRUE,
     SubgridVisible = TRUE):
```



Example 2

We consider the nonlinear oscillator $y'' + y^7 = 0$, $y(0) = 1$, $y'(0) = 0$. As a dynamical system for $Y = [y, y']$, we have to solve the following initial value problem $\frac{\partial}{\partial t} Y = f(t, Y)$, $Y(0) = Y_0$. For details on the format of `f`, please see `numeric::odesolve`:

```
f := (t, Y) -> [Y[2], - Y[1]^7]:
Y0 := [1, 0]:
```

The following generator produces a plot of the solution $Y(t)$ against the time parameter t:

```
G1 := (t, Y) -> [t, Y[1]]:
```

For demonstration purposes, plot the function $\frac{y^2}{2} + \frac{y'^2}{2}$. The generator `G2` produces the values $y(t)$, $y'(t)$ along the solution and plots these values against t:

```
G2 := (t, Y) -> [t, Y[1]^2/2 + Y[2]^2/2]:
```

The energy function, or "Hamiltonian", $\frac{y'^2}{2} + \frac{y^8}{8}$ should be conserved along the solution curve because the total energy of the system is constant. We define a corresponding generator `G3` to plot $y(t)$, $y'(t)$ as a function of t:

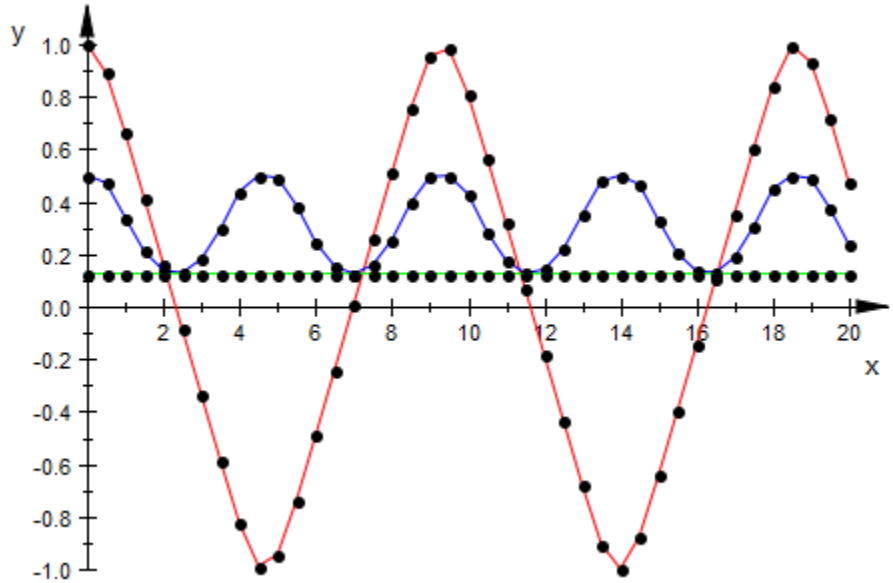
```
G3 := (t, Y) -> [t, Y[1]^8/8 + Y[2]^2/2]:
```

The solution curve is combined with the graph of the function and the Hamiltonian:

```
p := plot::Ode2d(f, [i/2 $ i = 0..40], Y0,
                 [G1, Style = Lines, Color = RGB::Red],
                 [G1, Style = Points, Color = RGB::Black],
                 [G2, Style = Lines, Color = RGB::Blue],
                 [G2, Style = Points, Color = RGB::Black],
                 [G3, Style = Lines, Color = RGB::Green],
                 [G3, Style = Points, Color = RGB::Black],
                 PointSize = 1.5*unit::mm,
                 LineWidth = 0.2*unit::mm
                ):
```

Note that by using each generator twice, we are able to set different colors for the lines and points. The renderer is called:

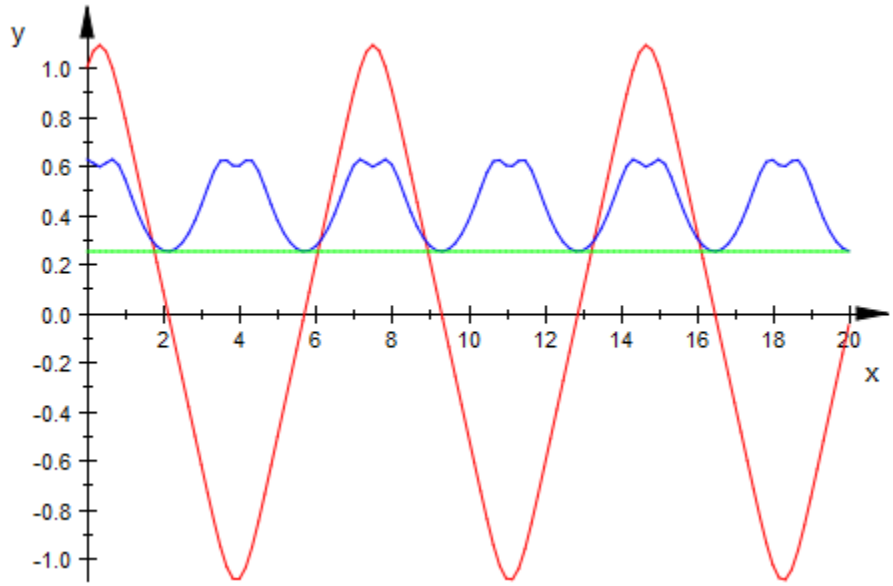
```
plot(p):
```



The Hamiltonian `G3` is constant which verifies the accuracy of the integrator.

To visualize the dependency of the trajectory on the initial conditions, we animate `plot::Ode2d` over different values of $y'(0)$:

```
plot(plot::Ode2d(f, [i/6 $ i = 0..120], [1, a], a = -1/2..1/2,
                 [G1, Style = Lines, Color = RGB::Red],
                 [G2, Style = Lines, Color = RGB::Blue],
                 [G3, Style = Lines, Color = RGB::Green],
                 LineWidth = 0.2*unit::mm, Frames=25))
```



Example 3

We consider the initial value problem $y' = f(t, y) = t \sin(t + y^2)$, $y(0) = 0$:

```
f := (t, y) -> t*sin(t + y^2): Y0:= [0]:
```

Use `numeric::ode2vectorfield` to generate the following vector field which is tangent to the solution curves:

```
p1 := plot::VectorField2d([1, f(t, y)], t = 0..4, y = -1.2..1.2,
                          Mesh = [21, 25], Color = RGB::Black):
```

The following object represents the plot of the solution as a function of `t`:

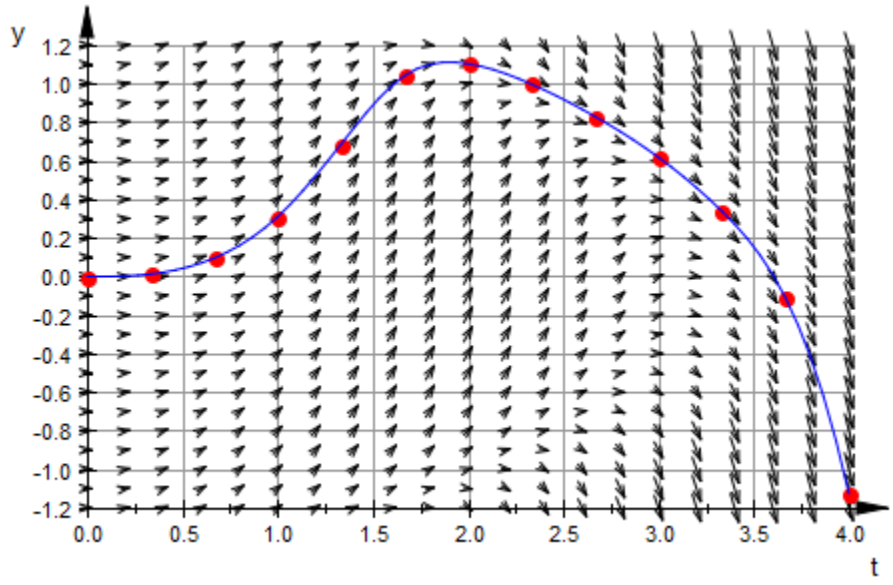
```
p2 := plot::Ode2d(
   (t,Y) -> [f(t, Y[1])], [i/3 $ i=0..12], Y0,
   [(t, Y) -> [t, Y[1]], Style = Points, Color = RGB::Red],
   [(t, Y) -> [t, Y[1]], Style = Splines, Color = RGB::Blue]):
```

We define the point size explicitly:

```
p2::PointSize := 2*unit::mm:
```

Finally, we combine the vector field and the ODE plot to a scene and call the renderer:

```
plot(p1, p2, XTicksDistance = 0.5, YTicksDistance = 0.2,
     Axes = Frame, AxesTitles = ["t", "y"],
     GridVisible = TRUE):
```



Example 4

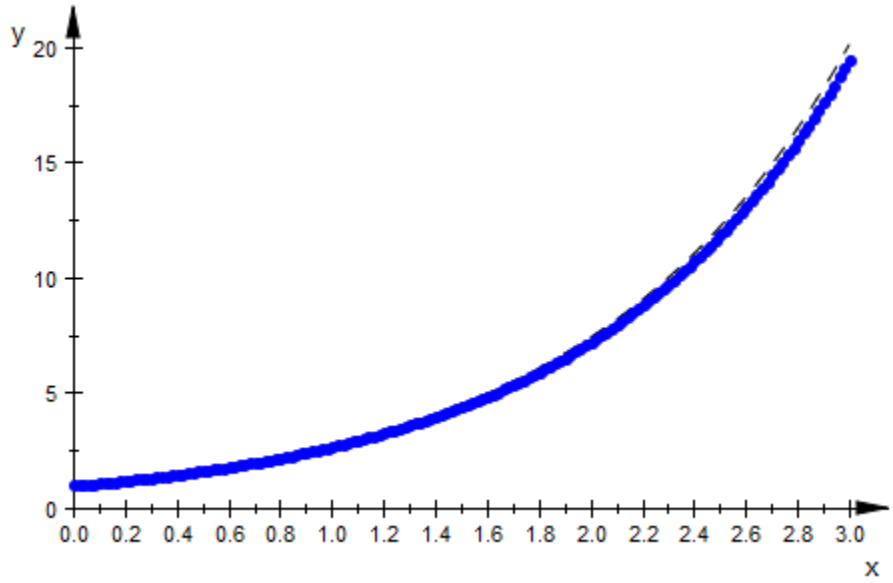
By default, `numeric::odesolve` (which is used by `plot::Ode2d` and `plot::Ode3d` internally) uses adaptive step sizes and a method of order 8. Usually, there is no reason to change these settings, except for demonstrative purposes. In the following animation, we use a straightforward explicit Euler method (of first order) and show how decreasing the step size improves the quality of the calculated solution.

Our differential equation is $y' = y$, obviously fulfilled by the exponential function:

```
[f, t0, Y0] := [numeric::ode2vectorfield(
                  {y'(t)=y(t), y(0)=1}, [y(t)])]:
```

To judge the quality of the numerical solution, we plot the symbolic solution alongside the approximation:

```
plot(plot::Function2d(exp(x), x=0..3,
                      Color = RGB::Black, LineStyle = Dashed),
     plot::Ode2d(f, [Automatic, 0, 3, 1/n], Y0, n = 1..50,
                 EULER1, Stepsize = 1/n,
                 [(t, Y) -> [t, Y[1]], Style=[Lines, Points]]))
```



Example 5

We consider the nonlinear oscillator $y'' + y^3 = \sin(t)$, $y(0) = 0$, $y'(0) = 0.5$. As a dynamical system for $Y = [y, y']$, we have to solve the following initial value problem $\frac{\partial}{\partial t} Y = f(t, Y)$, $Y(0) = Y_0$:

```
f := (t, Y) -> [Y[2], sin(t) - Y[1]^3]:
Y0 := [0, 0.5]:
```

The following generator produces a phase plot in the (x, y) plane, embedded in a 3D plot:

```
G1 := (t, Y) -> [Y[1], Y[2], 0]:
```

Further, we use the z coordinate of the 3D plot to display the value of the function $\frac{y^2}{2} + \frac{y'^2}{2}$ over the phase curve for demonstration purposes:

```
G2 := (t, Y) -> [Y[1], Y[2], (Y[1]^2 + Y[2]^2)/2]:
```

The phase curve in the (x, y) plane is combined with the graph of the function:

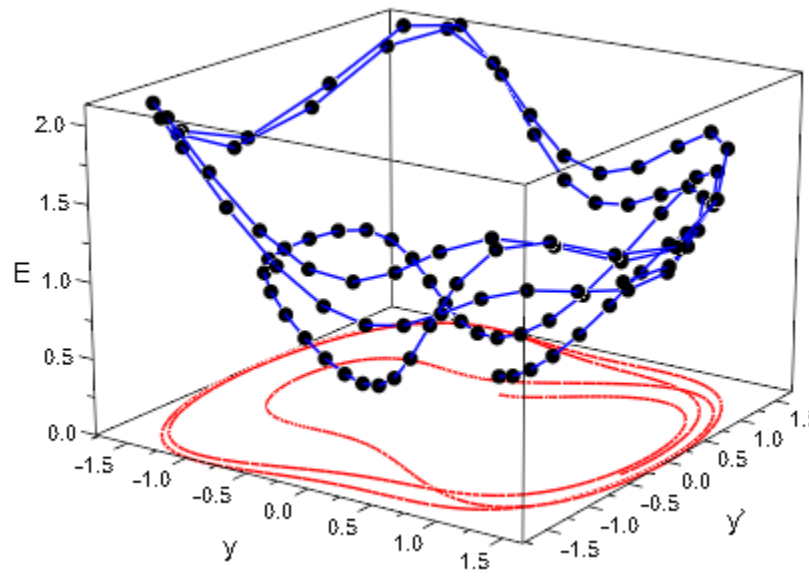
```
p := plot::Ode3d(f, [i/5 $ i = 0..100], Y0,
                 [G1, Style = Splines, Color = RGB::Red],
                 [G2, Style = Points, Color = RGB::Black],
                 [G2, Style = Lines, Color = RGB::Blue]):
```

We set an explicit size of the points used in the representation of the function:

```
p::PointSize := 2*unit::mm:
```

The renderer is called:

```
plot(p, AxesTitles = ["y", "y'", "E"],
     CameraDirection = [10, -15, 5]):
```



Example 6

The Lorenz ODE is the system

$$\frac{\partial}{\partial t}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} p\,(y-x) \\ -x\,z+r\,x-y \\ x\,y-b\,z \end{pmatrix}$$

with fixed parameters p, r, b. As a dynamical system for $Y = [x, y, z]$, we have to solve the ODE $\frac{\partial}{\partial t} Y = f(t, Y)$ with the following vector field:

```
f := proc(t, Y)
     local x, y, z;
     begin
        [x, y, z] := Y:
        [p*(y - x), -x*z + r*x - y, x*y - b*z]
     end_proc:
```

We consider the following parameters and the following initial condition `Y0`:

```
p := 10: r := 28: b := 1:
Y0 := [1, 1, 1]:
```

The following generator `Gxyz` produces a 3D phase plot of the solution. The generator `Gyz` projects the solution curve to the (y, z) plane with $x = 20$; the generator `Gxz` projects the solution curve to the (x, z) plane with $y = -15$; the generator `Gxy` projects the solution curve to the (x, y) plane with $z = 0$:

```
Gxyz := (t, Y) -> Y:
Gyz := (t, Y) -> [ 20,  Y[2], Y[3]]:
Gxz := (t, Y) -> [Y[1], -15,  Y[3]]:
Gxy := (t, Y) -> [Y[1], Y[2],   0 ]:
```

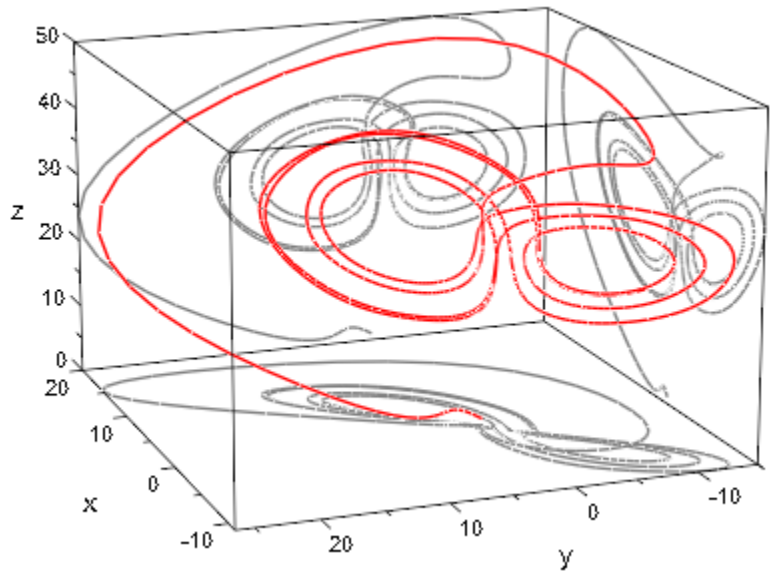
With these generators, we create a 3D plot object consisting of the phase curve and its projections.

```
object := plot::Ode3d(f, [i/10 $ i=1..100], Y0,
          [Gxyz, Style = Splines, Color = RGB::Red],
          [Gyz, Style = Splines, Color = RGB::Grey50],
          [Gxz, Style = Splines, Color = RGB::Grey50],
          [Gxy, Style = Splines, Color = RGB::Grey50],
```

```
            Submesh = 7):
```

Finally, the plot is rendered. This call is somewhat time consuming because it calls the numerical solver `numeric::odesolve` to produce the graphical data:

```
plot(object, CameraDirection = [-220, 110, 150])
```



Parameters

`f`

The vector field of the ODE: a procedure. See `numeric::odesolve` for details.

f is equivalent to the attribute `Function`.

t_0, t_1, …

The time mesh: real numerical values. If data are displayed with `Style = Splines`, these values must be in ascending order.

t_0, t_1, … is equivalent to the attribute `TimeMesh`.

t_{start}, t_{end}, t_{step}

The time mesh: real numerical values. t_{end} must be larger than t_{start} and t_{step} must be positive and should be smaller than *t_end - $t_s$tart*.

t_{start}, t_{end}, t_{step} are equivalent to the attribute `TimeMesh`.

Y_0

The initial condition of the ODE: a list or a 1-dimensional array. See `numeric::odesolve`.

Y_0 is equivalent to the attribute `InitialConditions`.

G_1, G_2, …

"generators of plot data": procedures mapping a solution point `(t, Y(t))` to a list `[x, y]` or `[x, y, z]` representing a plot point in 2D or 3D, respectively.

G_1, G_2, … is equivalent to the attribute `Projectors`.

`method`

Use a specific numerical scheme (see `numeric::odesolve`)

`a`

Animation parameter, specified as a $= a_{min}..a_{max}$, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

Options

`Style`

Option, specified as `Style = style`

Sets the style in which the plot data are displayed. The following styles are available: `Points`, `Lines`, `Splines`, `[Lines, Points]`, and `[Splines, Points]`. The default style is `[Splines, Points]`.

Color

Option, specified as `Color = c`

Sets the RGB color `c` in which the plot data are displayed. The default color of the *i*th generator is the *i*th entry of the attribute `Colors`.

RelErr

Option, specified as `RelErr = rtol`

Sets a numerical discretization tolerance (see `numeric::odesolve`)

AbsErr

Option, specified as `AbsErr = atol`

Sets a numerical discretization tolerance (see `numeric::odesolve`)

Stepsize

Option, specified as `Stepsize = h`

Sets a constant stepsize (see `numeric::odesolve`)

See Also

MuPAD Functions

`numeric::ode2vectorfield` | `numeric::odesolve` | `numeric::odesolve2`

MuPAD Graphical Primitives

`plot::Curve2d` | `plot::Curve3d` | `plot::Ode3d` | `plot::PointList2d` | `plot::PointList3d` | `plot::Polygon2d` | `plot::Polygon3d` | `plot::Streamlines2d`

plot::Ode3d

3D plots of ODE solutions

Syntax

```
plot::Ode3d(f, [t0, t1, …], Y0, <[G1, <Style = style1>, <Color = c1>], [G2, <Style = styl
plot::Ode3d(f, [Automatic, tstart, tend, tstep], Y0, <[G1, <Style = style1>, <Color = c1>],
plot::Ode3d([t0, t1, …], f, Y0, <[G1, <Style = style1>, <Color = c1>], [G2, <Style = styl
plot::Ode3d([Automatic, tstart, tend, tstep], f, Y0, <[G1, <Style = style1>, <Color = c1>],
```

Description

`plot::Ode3d(f, [t0, t1, …], Y0)` renders three-dimensional projections of the solutions of the initial value problem given by f, t_0 and Y_0.

`plot::Ode3d(f, [t0, t1, …], Y0, [G])` computes a mesh of numerical sample points $Y(t_0)$, $Y(t_1)$, … representing the solution $Y(t)$ of the first order differential equation (dynamical system)

$$\frac{\partial}{\partial t} Y = f(t, Y),\ Y(t_0) = Y_0,\ t_0,\ t \in \mathbb{R},\ Y_0,\ Y(t) \in \mathbb{C}^n$$

.

The procedure

$$G\colon (t, Y) \to [x(t, Y), y(t, Y)] \quad \text{or} \quad G\colon (t, Y) \to [x(t, Y), y(t, Y), z(t, Y)]$$

maps these solution points $(t_i, Y(t_i))$ in $\mathbb{R}\times\mathbb{C}^n$ to a mesh of 3D plot points $[x_i, y_i, z_i]$. These points can be connected by straight lines or interpolating splines.

The calling syntax of `plot::Ode2d` and `plot::Ode3d` as well as the functionality of these two procedures is identical. The only difference is that `plot::Ode2d` expects graphical generators G_1, G_2 etc. that produce graphical 2D points, whereas `plot::Ode3d` expects graphical generators producing 3D points.

Internally, a sequence of numerical sample points

`Y_1 := numeric::odesolve(f, t_0..t_1, Y_0, Options)`,

`Y_2 := numeric::odesolve(f, t_1..t_2, Y_1, Options)` etc.

is computed, where `Options` is some combination of `method`, `RelativeError = rtol`, `AbsoluteError = atol`, and `Stepsize = h`. See `numeric::odesolve` for details on the vector field procedure `f`, the initial condition Y_0, and the options.

The utility function `numeric::ode2vectorfield` may be used to produce the input parameters `f`, t_0, Y_0 from a set of differential expressions representing the ODE. Cf. "Example 1" on page 24-582.

Each of the "generators of plot data" G_1, G_2 etc. creates a graphical solution curve from the numerical sample points Y_0, Y_1 etc. Each generator `G`, say, is internally called in the form $G(t_0, Y_0), G(t_1, Y_1), \ldots$ to produce a sequence of plot points in 3D.

The solver `numeric::odesolve` returns the solution points Y_0, Y_1 etc. as lists or 1-dimensional arrays (the actual type is determined by the initial value Y_0). Consequently, each generator `G` must accept two arguments `(t, Y)`: `t` is a real parameter, `Y` is a "vector" (either a list or a 1-dimensional array).

Each generator must return a list with 3 elements representing the (*x*, *y*, *z*) coordinates of the graphical point associated with a solution point `(t, Y)` of the ODE.

All generators must produce graphical data of the same dimension, i.e., for `plot::Ode3d`, 3D data as lists with 3 elements.

Some examples:

`G := (t, Y) -> [Y_1, Y_2, Y_3]` creates a 3D phase plot of the first three components of the solution curve.

If no generators are given, `plot::Ode3d` by default plots each group of two components as functions of time with the same style.

Note that arbitrary values associated with the solution curve may be displayed graphically by an appropriate generator `G`. See "Example 2" on page 24-584 and "Example 5" on page 24-588.

Several generators G_1, G_2 etc. can be specified to generate several curves associated with the same numerical mesh Y_0, Y_1, ... See "Example 1" on page 24-582, "Example 2" on page 24-584, and "Example 3" on page 24-586.

The graphical data produced by each of the generators G_1, G_2 etc. consists of a sequence of mesh points in 2D or 3D, respectively.

With `Style = Points`, the graphical data are displayed as a discrete set of points.

With `Style = Lines`, the graphical data points are displayed as a curve consisting of straight line segments between the sample points. The points themselves are not displayed.

With `Style = Splines`, the graphical data points are displayed as a smooth spline curve connecting the sample points. The points themselves are not displayed.

With `Style = [Splines, Points]` and `Style = [Lines, Points]`, the effects of the styles used are combined, i.e., both the evaluation points and the straight lines or splines, respectively, are displayed.

The plot attributes accepted by `plot::Ode2d,Ode3d` include `Submesh = n`, where n is some positive integer. This attribute only has an effect on the curves which are returned for the graphical generators with `Style = Splines` and `Style = [Splines, Points]`, respectively. It serves for smoothening the graphical spline curve using a sufficiently high number of plot points.

n is the number of plot points between two consecutive numerical points corresponding to the time mesh. The default value is $n = 4$, i.e., the splines are plotted as 5 straight line segments connecting the numerical sample points.

Attributes

Attribute	Purpose	Default Value
`AbsoluteError`	maximal absolute discretization error	
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Colors`	list of colors to use	`[RGB::Blue, RGB::Red, RGB::Green, RGB::MuPADGold, RGB::Orange, RGB::Cyan, RGB::Magenta, RGB::LimeGreen,`

Attribute	Purpose	Default Value
		RGB::CadmiumYellowLight, RGB::AlizarinCrimson, RGB::Aqua, RGB::Lavender, RGB::SeaGreen, RGB::AureolineYellow, RGB::Banana, RGB::Beige, RGB::YellowGreen, RGB::Wheat, RGB::IndianRed, RGB::Black]
Frames	the number of frames in an animation	50
Function	function expression or procedure	
InitialConditions	initial conditions of the ODE	
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineWidth	width of lines	0.35
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
Name	the name of a plot object (for browser and legend)	
ODEMethod	the numerical scheme used for solving the ODE	DOPRI78
ParameterEnd	end value of the animation parameter	

Attribute	Purpose	Default Value
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	TRUE
Projectors	project an ODE solution to graphical points	
RelativeError	maximal relative discretization error	
Stepsize	set a constant step size	
Submesh	density of submesh (additional sample points)	4
TimeEnd	end time of the animation	10.0
TimeMesh	the numerical time mesh	
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	

Attribute	Purpose	Default Value
TitlePositionZ	position of object titles, z component	
USubmesh	density of additional sample points for parameter "u"	4
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE

Examples

Example 1

The following procedure `f` together with the initial value `Y0` represent the initial value problem $\frac{\partial}{\partial t} Y = f(t, Y) = t\,Y - Y^2$, $Y(0) = 2$. In MuPAD, the 1-dimensional vector Y is represented by a list with one element. The body of the function `f` below addresses the first (and only) entry of this list as Y_1 and returns the 1-dimensional vector $t\,Y$ - Y^2 as a list with one element. Also the initial condition Y_0 is a 1-dimensional vector represented by a list:

```
f := (t, Y) -> [t*Y[1] - Y[1]^2]:
Y0 := [2]:
```

Alternatively, the utility function `numeric::ode2vectorfield` can be used to generate the input parameters in a more intuitive way:

```
[f, t0, Y0] := [numeric::ode2vectorfield(
```

```
      {y'(t) = t*y(t) - y(t)^2, y(0) = 2}, [y(t)])]
```

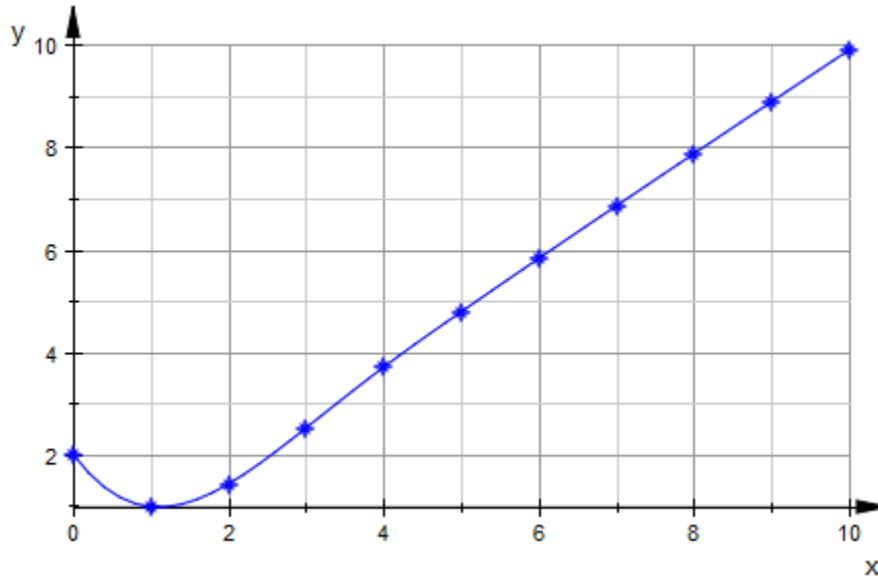
```
[proc f(t, Y) ... end, 0, [2]]
```

The numerical solution is to consist of sample points over the time mesh $t_i = i$, $i = 0, 1, \ldots, 10$. We use the default generator of `plot::Ode2d`. This generates the sample points together with a smooth spline curve connecting these points:

```
p := plot::Ode2d(f, [$ 0..10], Y0,
                 PointSize = 2*unit::mm,
                 PointStyle = Stars):
```

Finally, the ode solution is rendered by a call to `plot`:

```
plot(p, TicksDistance = 2.0, GridVisible = TRUE,
     SubgridVisible = TRUE):
```



Example 2

We consider the nonlinear oscillator $y'' + y^7 = 0$, $y(0) = 1$, $y'(0) = 0$. As a dynamical system for $Y = [y, y']$, we have to solve the following initial value problem $\frac{\partial}{\partial t} Y = f(t, Y)$, $Y(0) = Y_0$:

```
f := (t, Y) -> [Y[2], - Y[1]^7]:
Y0 := [1, 0]:
```

The following generator produces a plot of the solution $Y(t)$ against the time parameter t:

```
G1 := (t, Y) -> [t, Y[1]]:
```

Further, we are interested in the values of the function $F = \frac{y^2}{2} + \frac{y'^2}{2}$. The generator `G2` produces the values $F(y(t), y'(t))$ along the solution and plots these values against t:

```
G2 := (t, Y) -> [t, Y[1]^2/2 + Y[2]^2/2]:
```

The energy function (the "Hamiltonian") $H = \frac{y'^2}{2} + \frac{y^8}{8}$ should be conserved along the solution curve. We define a corresponding generator `G3` to plot $H(y(t), y'(t))$ as a function of t:

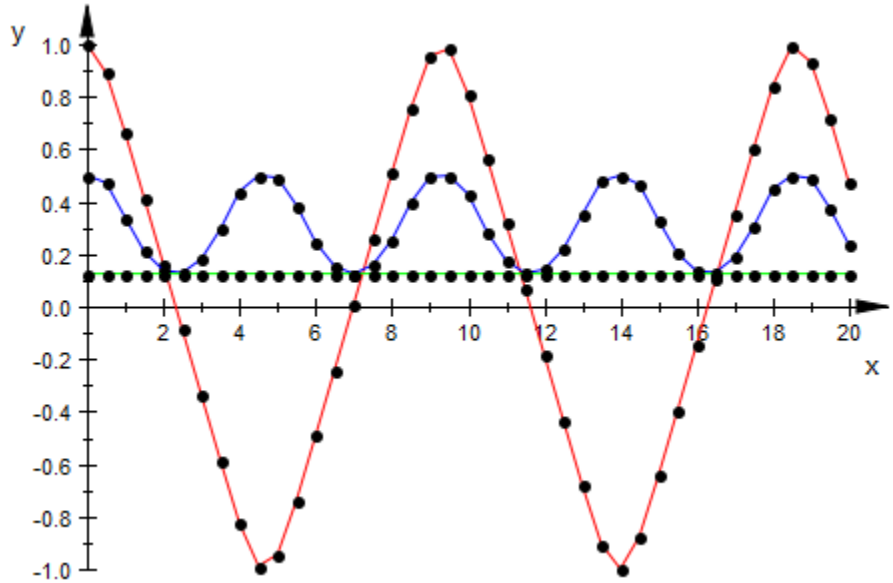
```
G3 := (t, Y) -> [t, Y[1]^8/8 + Y[2]^2/2]:
```

The solution curve is combined with the graph of the function $F(t) = F(y(t), y'(t))$ and the conserved energy $H(t) = H(y(t), y'(t))$:

```
p := plot::Ode2d(f, [i/2 $ i = 0..40], Y0,
                 [G1, Style = Lines, Color = RGB::Red],
                 [G1, Style = Points, Color = RGB::Black],
                 [G2, Style = Lines, Color = RGB::Blue],
                 [G2, Style = Points, Color = RGB::Black],
                 [G3, Style = Lines, Color = RGB::Green],
                 [G3, Style = Points, Color = RGB::Black],
                 PointSize = 1.5*unit::mm,
                 LineWidth = 0.2*unit::mm
                ):
```

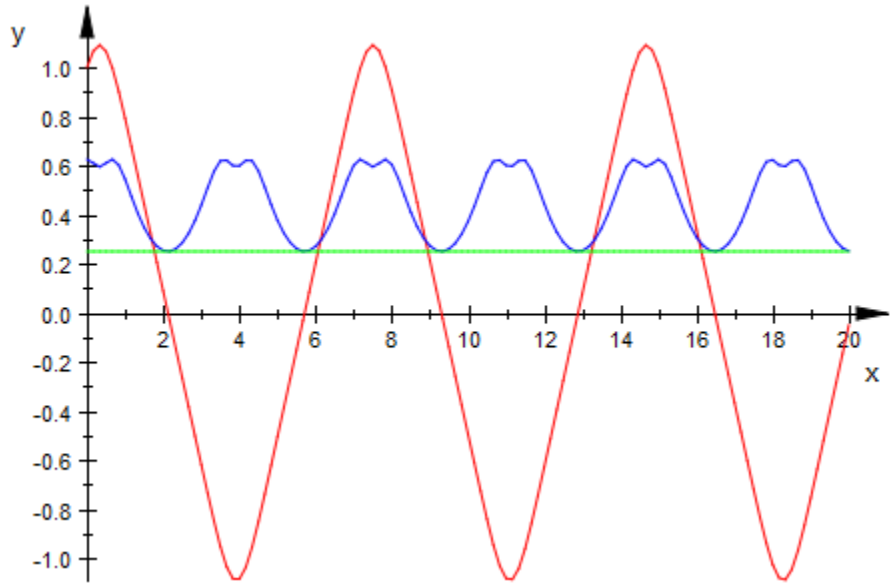
Note that by using each generator twice, we are able to set different colors for the lines and points. The renderer is called:

```
plot(p):
```



To visualize the dependency of the trajectory on the initial conditions, we animate `plot::Ode2d` over different values of $y'(0)$:

```
plot(plot::Ode2d(f, [i/6 $ i = 0..120], [1, a], a = -1/2..1/2,
                 [G1, Style = Lines, Color = RGB::Red],
                 [G2, Style = Lines, Color = RGB::Blue],
                 [G3, Style = Lines, Color = RGB::Green],
                 LineWidth = 0.2*unit::mm, Frames=25))
```



Example 3

We consider the initial value problem $y' = f(t, y) = t \sin(t + y^2)$, $y(0) = 0$:

```
f := (t, y) -> t*sin(t + y^2): Y0:= [0]:
```

The following vector field is tangent to the solution curves:

```
p1 := plot::VectorField2d([1, f(t, y)], t = 0..4, y = -1.2..1.2,
                          Mesh = [21, 25], Color = RGB::Black):
```

The following object represents the plot of the solution as a function of `t`:

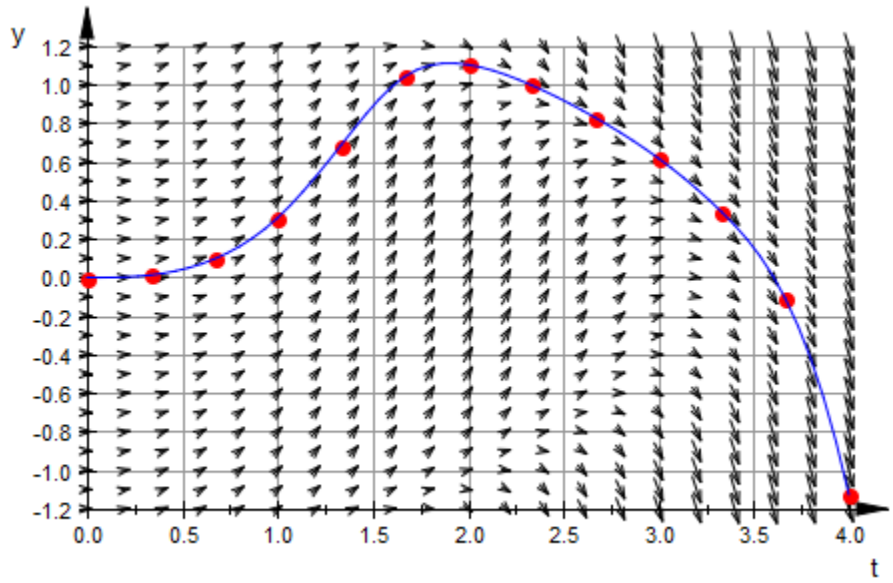
```
p2 := plot::Ode2d(
   (t,Y) -> [f(t, Y[1])], [i/3 $ i=0..12], Y0,
   [(t, Y) -> [t, Y[1]], Style = Points, Color = RGB::Red],
   [(t, Y) -> [t, Y[1]], Style = Splines, Color = RGB::Blue]):
```

We define the point size explicitly:

```
p2::PointSize := 2*unit::mm:
```

Finally, we combine the vector field and the ODE plot to a scene and call the renderer:

```
plot(p1, p2, XTicksDistance = 0.5, YTicksDistance = 0.2,
     Axes = Frame, AxesTitles = ["t", "y"],
     GridVisible = TRUE):
```



Example 4

By default, `numeric::odesolve` (which is used by `plot::Ode2d` and `plot::Ode3d` internally) uses adaptive step sizes and a method of order 8. Usually, there is no reason to change these settings, except for demonstrative purposes. In the following animation, we use a straightforward explicit Euler method (of first order) and show how decreasing the step size improves the quality of the calculated solution.

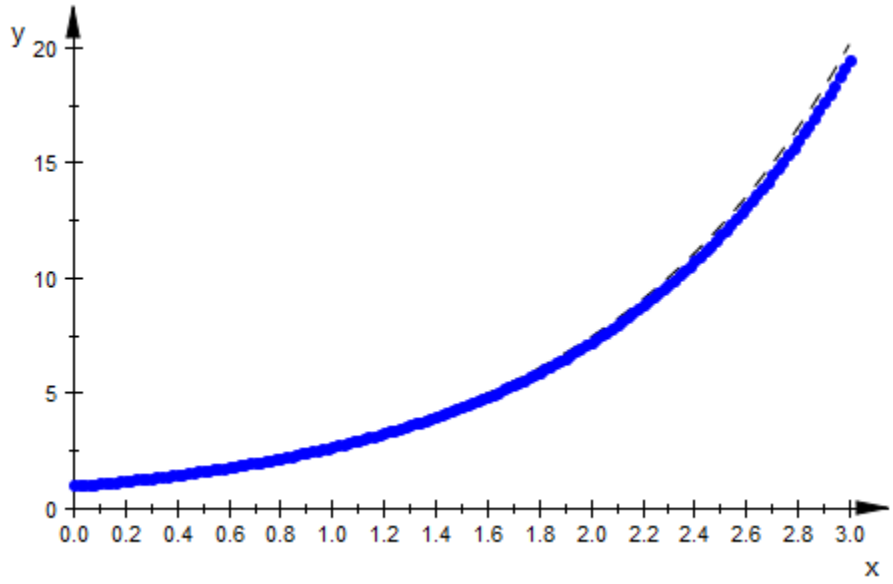
Our differential equation is $y' = y$, obviously fulfilled by the exponential function:

```
[f, t0, Y0] := [numeric::ode2vectorfield(
                   {y'(t)=y(t), y(0)=1}, [y(t)])]:
```

To judge the quality of the numerical solution, we plot the symbolic solution alongside the approximation:

```
plot(plot::Function2d(exp(x), x=0..3,
                      Color = RGB::Black, LineStyle = Dashed),
```

```
plot::Ode2d(f, [Automatic, 0, 3, 1/n], Y0, n = 1..50,
            EULER1, Stepsize = 1/n,
            [(t, Y) -> [t, Y[1]], Style=[Lines, Points]]))
```



Example 5

We consider the nonlinear oscillator $y'' + y^3 = \sin(t)$, $y(0) = 0$, $y'(0) = 0.5$. As a dynamical system for $Y = [y, y']$, we have to solve the following initial value problem $\frac{\partial}{\partial t} Y = f(t, Y)$, $Y(0) = Y_0$:

```
f := (t, Y) -> [Y[2], sin(t) - Y[1]^3]:
Y0 := [0, 0.5]:
```

The following generator produces a phase plot in the (x, y) plane, embedded in a 3D plot:

```
G1 := (t, Y) -> [Y[1], Y[2], 0]:
```

Further, we use the z coordinate of the 3D plot to display the value of the "energy" function $E = \frac{y^2}{2} + \frac{y'^2}{2}$ over the phase curve:

```
G2 := (t, Y) -> [Y[1], Y[2], (Y[1]^2 + Y[2]^2)/2]:
```

The phase curve in the (x, y) plane is combined with the graph of the energy function:

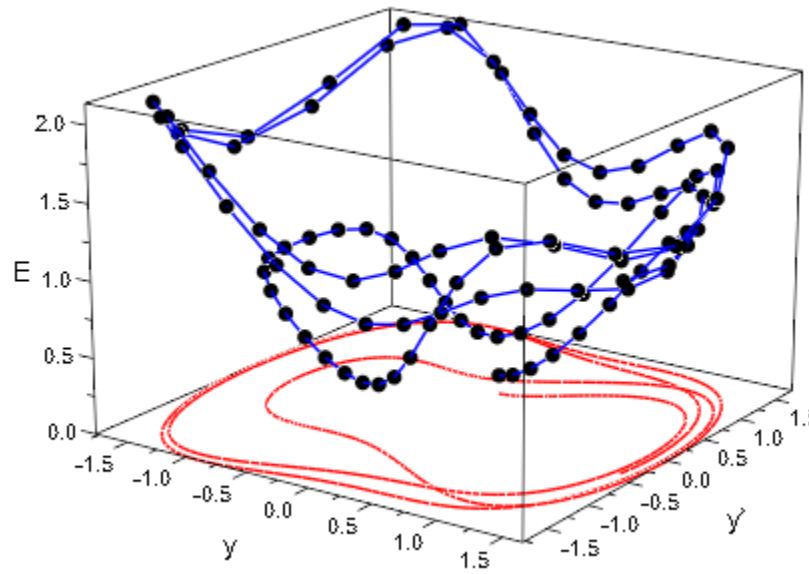
```
p := plot::Ode3d(f, [i/5 $ i = 0..100], Y0,
                 [G1, Style = Splines, Color = RGB::Red],
                 [G2, Style = Points, Color = RGB::Black],
                 [G2, Style = Lines, Color = RGB::Blue]):
```

We set an explicit size of the points used in the representation of the energy:

```
p::PointSize := 2*unit::mm:
```

The renderer is called:

```
plot(p, AxesTitles = ["y", "y'", "E"],
     CameraDirection = [10, -15, 5]):
```



Example 6

The Lorenz ODE is the system

$$\frac{\partial}{\partial t}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} p\,(y-x) \\ -x\,z+r\,x-y \\ x\,y-b\,z \end{pmatrix}$$

with fixed parameters p, r, b. As a dynamical system for $Y = [x, y, z]$, we have to solve the ODE $\frac{\partial}{\partial t} Y = f(t, Y)$ with the following vector field:

```
f := proc(t, Y)
     local x, y, z;
     begin
        [x, y, z] := Y:
        [p*(y - x), -x*z + r*x - y, x*y - b*z]
     end_proc:
```

We consider the following parameters and the following initial condition `Y0`:

```
p := 10: r := 28: b := 1:
Y0 := [1, 1, 1]:
```

The following generator `Gxyz` produces a 3D phase plot of the solution. The generator `Gyz` projects the solution curve to the (y, z) plane with $x = 20$; the generator `Gxz` projects the solution curve to the (x, z) plane with $y = -15$; the generator `Gxy` projects the solution curve to the (x, y) plane with $z = 0$:

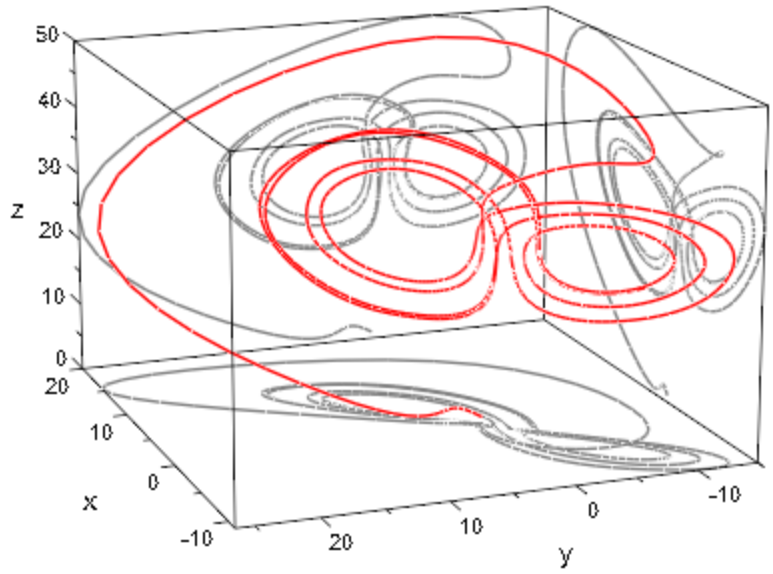
```
Gxyz := (t, Y) -> Y:
Gyz := (t, Y) -> [ 20,  Y[2], Y[3]]:
Gxz := (t, Y) -> [Y[1], -15,  Y[3]]:
Gxy := (t, Y) -> [Y[1], Y[2],   0 ]:
```

With these generators, we create a 3D plot object consisting of the phase curve and its projections.

```
object := plot::Ode3d(f, [i/10 $ i=1..100], Y0,
           [Gxyz, Style = Splines, Color = RGB::Red],
           [Gyz, Style = Splines, Color = RGB::Grey50],
           [Gxz, Style = Splines, Color = RGB::Grey50],
           [Gxy, Style = Splines, Color = RGB::Grey50],
           Submesh = 7):
```

Finally, the plot is rendered. This call is somewhat time consuming because it calls the numerical solver `numeric::odesolve` to produce the graphical data:

```
plot(object, CameraDirection = [-220, 110, 150])
```



Parameters

f

The vector field of the ODE: a procedure. See `numeric::odesolve` for details.

f is equivalent to the attribute `Function`.

t_0, t_1, …

The time mesh: real numerical values. If data are displayed with `Style = Splines`, these values must be in ascending order.

t_0, t_1, … is equivalent to the attribute `TimeMesh`.

t_{start}, t_{end}, t_{step}

The time mesh: real numerical values. t_{end} must be larger than t_{start} and t_{step} must be positive and should be smaller than t_end - t_start.

t_{start}, t_{end}, t_{step} are equivalent to the attribute `TimeMesh`.

Y_0

The initial condition of the ODE: a list or a 1-dimensional array. See `numeric::odesolve`.

Y_0 is equivalent to the attribute `InitialConditions`.

G_1, G_2, …

"generators of plot data": procedures mapping a solution point `(t, Y(t))` to a list `[x, y]` or `[x, y, z]` representing a plot point in 2D or 3D, respectively.

G_1, G_2, … is equivalent to the attribute `Projectors`.

`method`

Use a specific numerical scheme (see `numeric::odesolve`)

`a`

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

Options

`Style`

Option, specified as `Style = style`

Sets the style in which the plot data are displayed. The following styles are available: `Points`, `Lines`, `Splines`, `[Lines, Points]`, and `[Splines, Points]`. The default style is `[Splines, Points]`.

`Color`

Option, specified as `Color = c`

Sets the RGB color `c` in which the plot data are displayed. The default color of the *i*th generator is the *i*th entry of the attribute `Colors`.

RelErr

Option, specified as `RelErr = rtol`

Sets a numerical discretization tolerance (see `numeric::odesolve`)

AbsErr

Option, specified as `AbsErr = atol`

Sets a numerical discretization tolerance (see `numeric::odesolve`)

Stepsize

Option, specified as `Stepsize = h`

Sets a constant stepsize (see `numeric::odesolve`)

See Also

MuPAD Functions

`numeric::ode2vectorfield` | `numeric::odesolve` | `numeric::odesolve2`

MuPAD Graphical Primitives

`plot::Curve2d` | `plot::Curve3d` | `plot::Ode2d` | `plot::PointList2d` | `plot::PointList3d` | `plot::Polygon2d` | `plot::Polygon3d` | `plot::Streamlines2d`

plot::Parallelogram2d

2D parallelograms

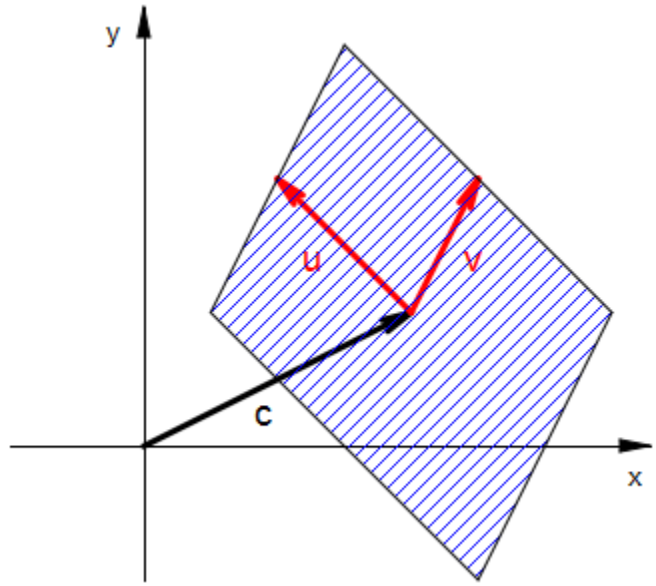
Syntax

```
plot::Parallelogram2d([c_x, c_y], [u_x, u_y], [v_x, v_y], <a = a_min .. a_max>, options)
```

Description

`plot::Parallelogram2d(c, u, v)` defines a 2D parallelogram $\{\vec{c} + \lambda\,\vec{u} + \mu\,\vec{v} \mid \lambda \in [-1, 1], \mu \in [-1, 1]\}$ with center $\vec{c}$ and vectors $\vec{u}$, $\vec{v}$ spanning the plane of the parallelogram. This is a rectangle with sides of length $2\,|\vec{u}|$, $2\,|\vec{v}|$ if the vectors $\vec{u}$ and $\vec{v}$ are orthogonal.

`plot::Parallelogram2d` creates a 2D parallelogram with center $\vec{c}$ = `[c_x, c_y]` and sides given by the vectors $2\,\vec{u}$ = `[2 u_x, 2 u_y]` and $2\,\vec{v}$ = `[2 v_x, 2 v_y]`. The corners of the parallelogram are given by $\vec{c} - \vec{u} - \vec{v}$, $\vec{c} - \vec{u} + \vec{v}$, $\vec{c} + \vec{u} - \vec{v}$, and $\vec{c} + \vec{u} + \vec{v}$:



By default, the area of the parallelogram is filled with the color specified by the attribute `Color` or, equivalently, `FillColor`. With `Filled` = `False`, only the border lines of the parallelogram are visible. Their color is set by the attribute `LineColor`.

Alternatively, the center and the spanning vectors can be given as vectors.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`
`Center`	center of objects, rotation center	[0, 0]
`CenterX`	center of objects, rotation center, x-component	0

Attribute	Purpose	Default Value
CenterY	center of objects, rotation center, y-component	0
Color	the main color	RGB::Blue
Filled	filled or transparent areas and surfaces	FALSE
FillColor	color of areas and surfaces	RGB::Red
FillPattern	type of area filling	DiagonalLines
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Blue
LineWidth	width of lines	0.35
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
Tangent1	first vector spanning parallelograms	[0, 1]

Attribute	Purpose	Default Value
Tangent2	second vector spanning parallelograms	[1, 0]
Tangent1X	first vector spanning parallelograms, x component	0
Tangent1Y	first vector spanning parallelograms, y component	1
Tangent2X	second vector spanning parallelograms, x component	1
Tangent2Y	second vector spanning parallelograms, y component	0
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	

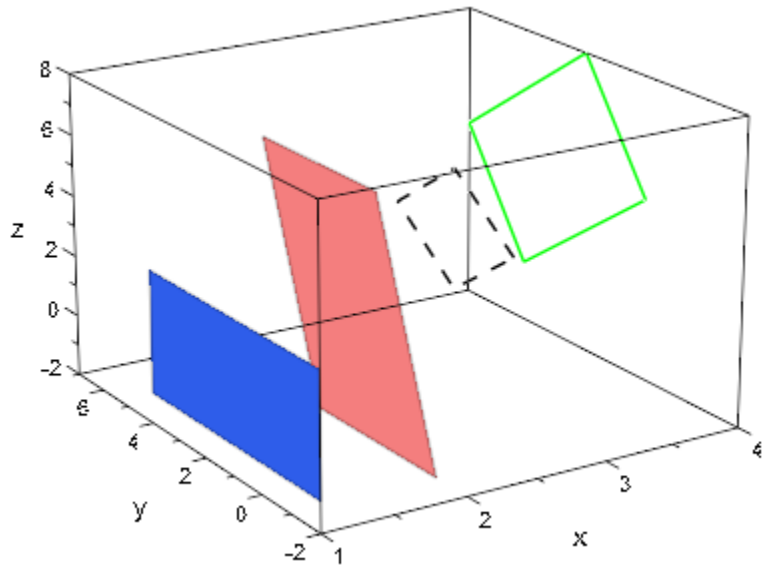
Attribute	Purpose	Default Value
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE

Examples

Example 1

We plot several rectangles and parallelograms using different presentation styles:

```
plot(plot::Parallelogram3d([1, 1, 1], [0, 0, 2], [0, 3, 0]),
     plot::Parallelogram3d([2, 2, 2], [0, 1, 4], [0, 2, 0],
                           FillColor = RGB::Red.[0.5]),
     plot::Parallelogram3d([3, 3, 3], [0, 1, 1], [0, 1, -1],
                           Filled = FALSE, LineStyle = Dashed,
                           LineColor = RGB::Black),
     plot::Parallelogram3d([4, 4, 4], [0, 1, 2], [0, 2, -2],
                           Filled = FALSE, LineColor = RGB::Green)
):
```

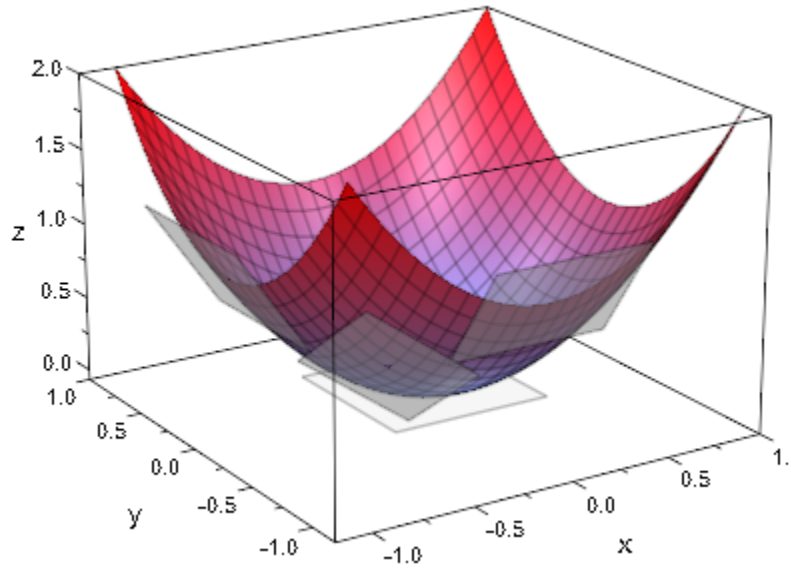


Example 2

We use `plot::Parallelogram3d` to visualize tangent planes of a surface. The first surface is the graph of the function $f(x, y) = x^2 + y^2$. At a point $(x, y, f(x, y))$ on the graph, the tangent vectors in the x and y direction are given by $(1, 0, 2x)$ and $(0, 1, 2y)$, respectively. After normalization to the length 0.4, they yield the tangent vectors u, v used in the construction of the tangent planes:

```
f := (x, y) -> x^2 + y^2:
c:= (x, y) -> [x, y, f(x, y)]:
u := (x, y) -> [0.4/sqrt(1+4*x^2), 0, 0.8*x/sqrt(1+4*x^2)]:
v := (x, y) -> [0, 0.4/sqrt(1+4*y^2), 0.8*y/sqrt(1+4*y^2)]:
plot(plot::Function3d(f(x, y), x = -1..1, y = -1..1),
     plot::Parallelogram3d(c(0, 0), u(0, 0), v(0, 0),
                           Color = RGB::Grey.[0.5]),
     plot::Parallelogram3d(c(0, -1), u(0, -1), v(0, -1),
                           Color = RGB::Grey.[0.5]),
     plot::Parallelogram3d(c(-1, 0), u(-1, 0), v(-1, 0),
                           Color = RGB::Grey.[0.5]),
     plot::Parallelogram3d(c(-1/2, -1/2), u(-1/2, -1/2),
                           v(-1/2, -1/2),
```

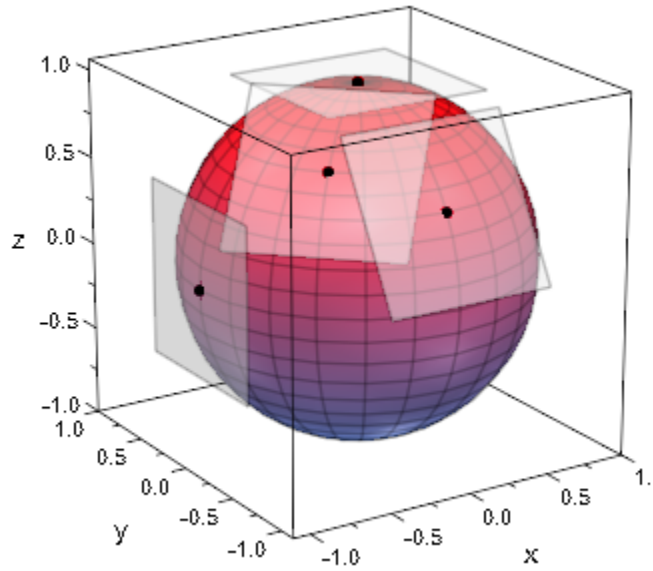
```
                                Color = RGB::Grey.[0.5])):
```



The second surface is a sphere, parametrized by spherical coordinates p and t (polar and azimuth angle). At a point $(x(p, t), y(p, t), z(p, t))$ on the sphere, the tangent vectors in the p and t direction are given by differention of x, y, z w.r.t. p and t, respectively. After normalization to the length 0.5, they yield the tangent vectors u, v used in the construction of the tangent planes:

```
x := (p, t) -> cos(p)*sin(t):
y := (p, t) -> sin(p)*sin(t):
z := (p, t) -> cos(t):
c := (p, t) -> [x(p, t), y(p, t), z(p, t)]:
u := (p, t) -> [-0.5*sin(p), 0.5*cos(p), 0]:
v := (p, t) -> [0.5*cos(p)*cos(t), 0.5*sin(p)*cos(t),
                -0.5*sin(t)]:
plot(plot::Surface(c(p, t), p = 0..2*PI, t = 0..PI),
     plot::Point3d(c(0, 0), Color = RGB::Black),
     plot::Parallelogram3d(c(0, 0), u(0, 0), v(0, 0),
                           Color = RGB::Grey.[0.5]),
     plot::Point3d(c(-3*PI/4, PI/4), Color = RGB::Black),
     plot::Parallelogram3d(c(-3*PI/4, PI/4),
                           u(-3*PI/4, PI/4),
```

```
                        v(-3*PI/4, PI/4),
                        Color = RGB::Grey.[0.5]),
      plot::Point3d(c(-PI/2, PI/3), Color = RGB::Black),
      plot::Parallelogram3d(c(-PI/2, PI/3),
                        u(-PI/2, PI/3),
                        v(-PI/2, PI/3),
                        Color = RGB::Grey.[0.5]),
      plot::Point3d(c(PI, PI/2), Color = RGB::Black),
      plot::Parallelogram3d(c(PI, PI/2),
                        u(PI, PI/2),
                        v(PI, PI/2),
                        Color = RGB::Grey.[0.5]),
      Scaling = Constrained):
```




```
delete f, c, u, v, x, y, z:
```

Parameters

c_x, c_y

Coordinates of the center: real numerical values or expressions of the animation parameter `a`.

c_x, c_y are equivalent to the attributes `CenterX`, `CenterY`.

u_x, u_y

Components of the first vector spanning the parallelogram: real numerical values or expressions of the animation parameter `a`.

u_x, u_y are equivalent to the attributes `Tangent1X`, `Tangent1Y`.

v_x, v_y

Components of the second vector spanning the parallelogram: real numerical values or expressions of the animation parameter `a`.

v_x, v_y are equivalent to the attributes `Tangent2X`, `Tangent2Y`.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Box` | `plot::Line3d` | `plot::Parallelogram3d` | `plot::Polygon3d` | `plot::Rectangle`

plot::Parallelogram3d

3D parallelograms

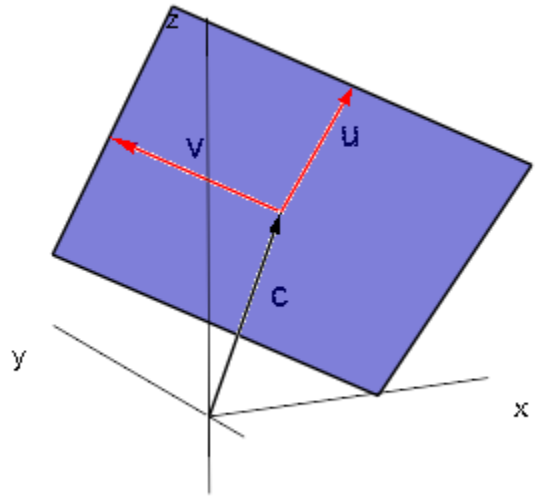
Syntax

```
plot::Parallelogram3d([cx, cy, cz], [ux, uy, uz], [vx, vy, vz], <a = amin .. amax>, options)
```

Description

`plot::Parallelogram3d(c, u, v)` defines a 3D parallelogram $\{\vec{c} + \lambda\,\vec{u} + \mu\,\vec{v} \mid \lambda \in [-1, 1], \mu \in [-1, 1]\}$ with center $\vec{c}$ and vectors $\vec{u}$, $\vec{v}$ spanning the plane of the parallelogram. This is a rectangle with sides of length $2\,|\vec{u}|$, $2\,|\vec{v}|$ if the vectors $\vec{u}$ and $\vec{v}$ are orthogonal.

`plot::Parallelogram3d` creates a 3D parallelogram with center $\vec{c}$ = `[cx, cy, cz]` and sides given by the vectors $2\,\vec{u}$ = `[2 ux, 2 uy, 2 uz]` and $2\,\vec{v}$ = `[2 vx, 2 vy, 2 vz]`. The corners of the parallelogram are given by $\vec{c} - \vec{u} - \vec{v}$, $\vec{c} - \vec{u} + \vec{v}$, $\vec{c} + \vec{u} - \vec{v}$, and $\vec{c} + \vec{u} + \vec{v}$:



By default, the area of the parallelogram is filled with the color specified by the attribute `Color` or, equivalently, `FillColor`. With `Filled` = `False`, only the border lines of the parallelogram are visible. Their color is set by the attribute `LineColor`.

Alternatively, the center and the spanning vectors can be given as vectors.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Center`	center of objects, rotation center	`[0, 0, 0]`
`CenterX`	center of objects, rotation center, x-component	`0`
`CenterY`	center of objects, rotation center, y-component	`0`

Attribute	Purpose	Default Value
CenterZ	center of objects, rotation center, z-component	0
Color	the main color	RGB::LightBlue
Filled	filled or transparent areas and surfaces	TRUE
FillColor	color of areas and surfaces	RGB::LightBlue
FillColor2	second color of areas and surfaces for color blends	RGB::CornflowerBlue
FillColorType	surface filling types	Flat
FillColorDirection	the direction of color transitions on surfaces	[0, 0, 1]
FillColorDirectionX	x-component of the direction of color transitions on surfaces	0
FillColorDirectionY	y-component of the direction of color transitions on surfaces	0
FillColorDirectionZ	z-component of the direction of color transitions on surfaces	1
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Black.[0.25]
LineWidth	width of lines	0.35
LineStyle	solid, dashed or dotted lines?	Solid

Attribute	Purpose	Default Value
LinesVisible	visibility of lines	TRUE
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
Tangent1	first vector spanning parallelograms	[0, 1, 0]
Tangent2	second vector spanning parallelograms	[1, 0, 0]
Tangent1X	first vector spanning parallelograms, x component	0
Tangent1Y	first vector spanning parallelograms, y component	1
Tangent2X	second vector spanning parallelograms, x component	1
Tangent1Z	first vector spanning parallelograms, z component	0
Tangent2Y	second vector spanning parallelograms, y component	0
Tangent2Z	second vector spanning parallelograms, z component	0

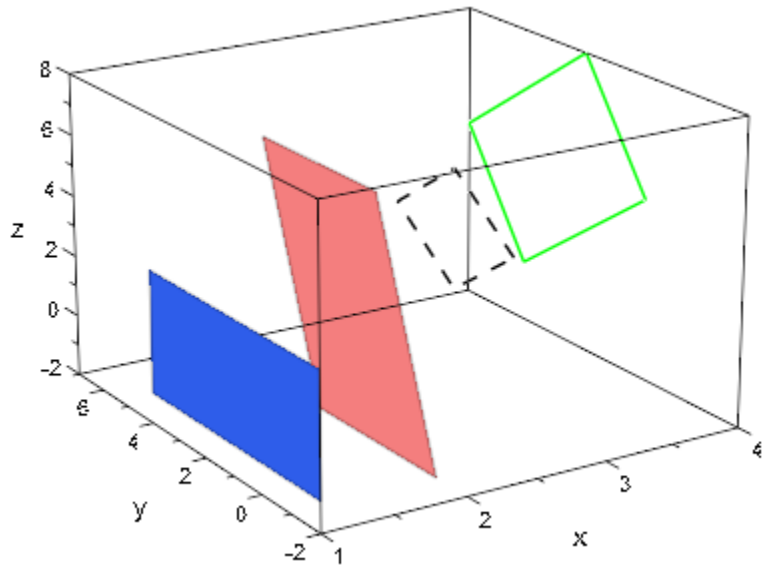
Attribute	Purpose	Default Value
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE

Examples

Example 1

We plot several rectangles and parallelograms using different presentation styles:

```
plot(plot::Parallelogram3d([1, 1, 1], [0, 0, 2], [0, 3, 0]),
     plot::Parallelogram3d([2, 2, 2], [0, 1, 4], [0, 2, 0],
                           FillColor = RGB::Red.[0.5]),
     plot::Parallelogram3d([3, 3, 3], [0, 1, 1], [0, 1, -1],
                           Filled = FALSE, LineStyle = Dashed,
                           LineColor = RGB::Black),
     plot::Parallelogram3d([4, 4, 4], [0, 1, 2], [0, 2, -2],
                           Filled = FALSE, LineColor = RGB::Green)
):
```

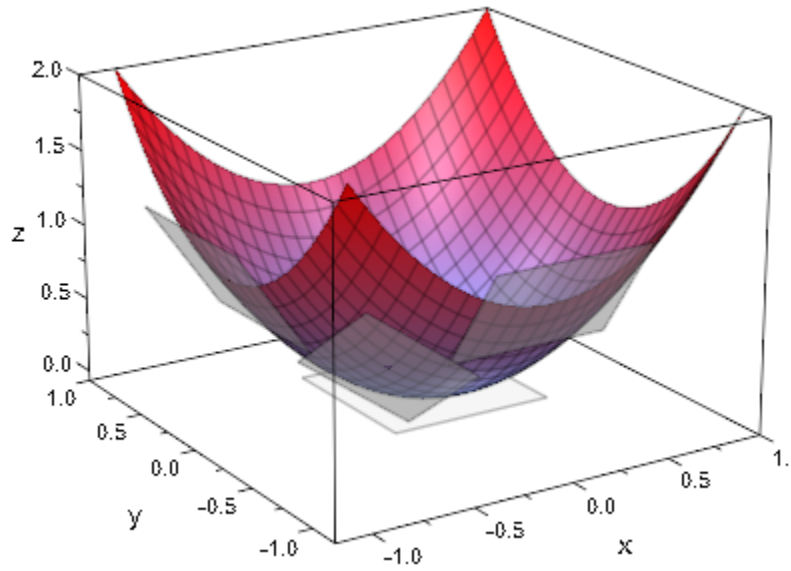


Example 2

We use `plot::Parallelogram3d` to visualize tangent planes of a surface. The first surface is the graph of the function $f(x, y) = x^2 + y^2$. At a point $(x, y, f(x, y))$ on the

graph, the tangent vectors in the x and y direction are given by $(1, 0, 2x)$ and $(0, 1, 2y)$, respectively. After normalization to the length 0.4, they yield the tangent vectors u, v used in the construction of the tangent planes:

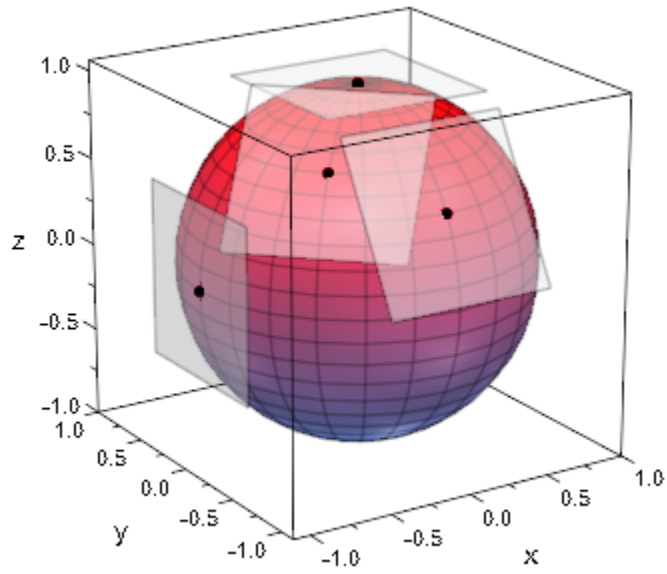
```
f := (x, y) -> x^2 + y^2:
c:= (x, y) -> [x, y, f(x, y)]:
u := (x, y) -> [0.4/sqrt(1+4*x^2), 0, 0.8*x/sqrt(1+4*x^2)]:
v := (x, y) -> [0, 0.4/sqrt(1+4*y^2), 0.8*y/sqrt(1+4*y^2)]:
plot(plot::Function3d(f(x, y), x = -1..1, y = -1..1),
     plot::Parallelogram3d(c(0, 0), u(0, 0), v(0, 0),
                           Color = RGB::Grey.[0.5]),
     plot::Parallelogram3d(c(0, -1), u(0, -1), v(0, -1),
                           Color = RGB::Grey.[0.5]),
     plot::Parallelogram3d(c(-1, 0), u(-1, 0), v(-1, 0),
                           Color = RGB::Grey.[0.5]),
     plot::Parallelogram3d(c(-1/2, -1/2), u(-1/2, -1/2),
                           v(-1/2, -1/2),
                           Color = RGB::Grey.[0.5])):
```



The second surface is a sphere, parametrized by spherical coordinates p and t (polar and azimuth angle). At a point $(x(p, t), y(p, t), z(p, t))$ on the sphere, the tangent vectors

in the p and t direction are given by differention of x, y, z w.r.t. p and t, respectively. After normalization to the length 0.5, they yield the tangent vectors u, v used in the construction of the tangent planes:

```
x := (p, t) -> cos(p)*sin(t):
y := (p, t) -> sin(p)*sin(t):
z := (p, t) -> cos(t):
c := (p, t) -> [x(p, t), y(p, t), z(p, t)]:
u := (p, t) -> [-0.5*sin(p), 0.5*cos(p), 0]:
v := (p, t) -> [0.5*cos(p)*cos(t), 0.5*sin(p)*cos(t),
                -0.5*sin(t)]:
plot(plot::Surface(c(p, t), p = 0..2*PI, t = 0..PI),
     plot::Point3d(c(0, 0), Color = RGB::Black),
     plot::Parallelogram3d(c(0, 0), u(0, 0), v(0, 0),
                           Color = RGB::Grey.[0.5]),
     plot::Point3d(c(-3*PI/4, PI/4), Color = RGB::Black),
     plot::Parallelogram3d(c(-3*PI/4, PI/4),
                           u(-3*PI/4, PI/4),
                           v(-3*PI/4, PI/4),
                           Color = RGB::Grey.[0.5]),
     plot::Point3d(c(-PI/2, PI/3), Color = RGB::Black),
     plot::Parallelogram3d(c(-PI/2, PI/3),
                           u(-PI/2, PI/3),
                           v(-PI/2, PI/3),
                           Color = RGB::Grey.[0.5]),
     plot::Point3d(c(PI, PI/2), Color = RGB::Black),
     plot::Parallelogram3d(c(PI, PI/2),
                           u(PI, PI/2),
                           v(PI, PI/2),
                           Color = RGB::Grey.[0.5]),
     Scaling = Constrained):
```




```
delete f, c, u, v, x, y, z:
```

Parameters

c_x, c_y, c_z

Coordinates of the center: real numerical values or expressions of the animation parameter `a`.

c_x, c_y, c_z are equivalent to the attributes `CenterX`, `CenterY`, `CenterZ`.

u_x, u_y, u_z

Components of the first vector spanning the parallelogram: real numerical values or expressions of the animation parameter `a`.

u_x, u_y, u_z are equivalent to the attributes `Tangent1X`, `Tangent1Y`, `Tangent1Z`.

v_x, v_y, v_z

Components of the second vector spanning the parallelogram: real numerical values or expressions of the animation parameter `a`.

v_x, v_y, v_z are equivalent to the attributes `Tangent2X`, `Tangent2Y`, `Tangent2Z`.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Box` | `plot::Line3d` | `plot::Parallelogram2d` | `plot::Polygon3d` | `plot::Rectangle`

plot::Piechart2d

2D pie charts

Syntax

`plot::Piechart2d([d1, d2, …], <a = amin .. amax>, options)`

`plot::Piechart2d(A, <a = amin .. amax>, options)`

Description

`plot::Piechart2d([d1, d2, d3, …])` creates a 2D pie chart with pieces of size ratios $d_1 : d_2 : d_3 : \ldots$.

With the input data `d1`, `d2` etc., the i-th piece of the pie has the opening angle `2*PI*d_i/(d_1 + d_2 + Symbol::dots)`.

The attribute `Titles` allows to attach titles to the pieces of the pie. In contrast to the overall title of the pie chart (`Title`, `TitleFont`), the titles of the pieces react to `TextFont`.

The attribute `Moves` allows to move the pieces away from the pie center for highlighting.

The attributes `Center` and `Radius` allow to position and scale a pie chart relative to other graphical objects in the same scene.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`
`Center`	center of objects, rotation center	`[0, 0]`

Attribute	Purpose	Default Value
CenterX	center of objects, rotation center, x-component	0
CenterY	center of objects, rotation center, y-component	0
Color	the main color	
Colors	list of colors to use	[RGB::Blue, RGB::Red, RGB::Green, RGB::MuPADGold, RGB::Orange, RGB::Cyan, RGB::Magenta, RGB::LimeGreen, RGB::CadmiumYellowLight, RGB::AlizarinCrimson]
Data	the (statistical) data to plot	[1]
Filled	filled or transparent areas and surfaces	TRUE
FillPattern	type of area filling	Solid
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Black
LineWidth	width of lines	0.35
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
Moves	displacements of pieces in pie charts	[0]

Attribute	Purpose	Default Value
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
Radius	radius of circles, spheres etc.	1
TextFont	font of text objects	[" sans-serif ", 11]
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
Titles	list of titles for object parts	[" "]
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	

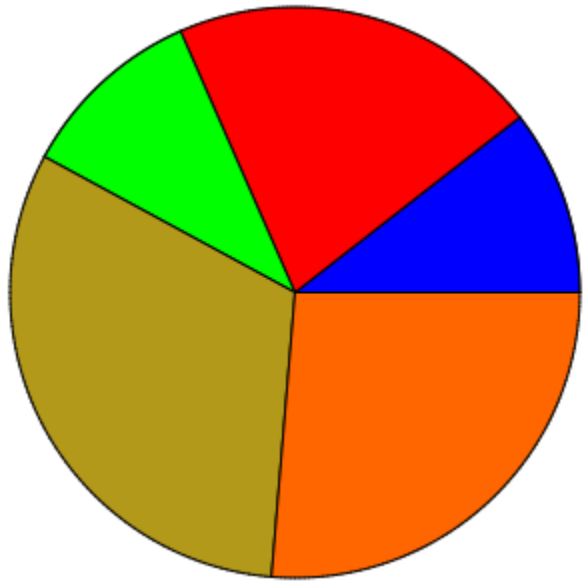
Attribute	Purpose	Default Value
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

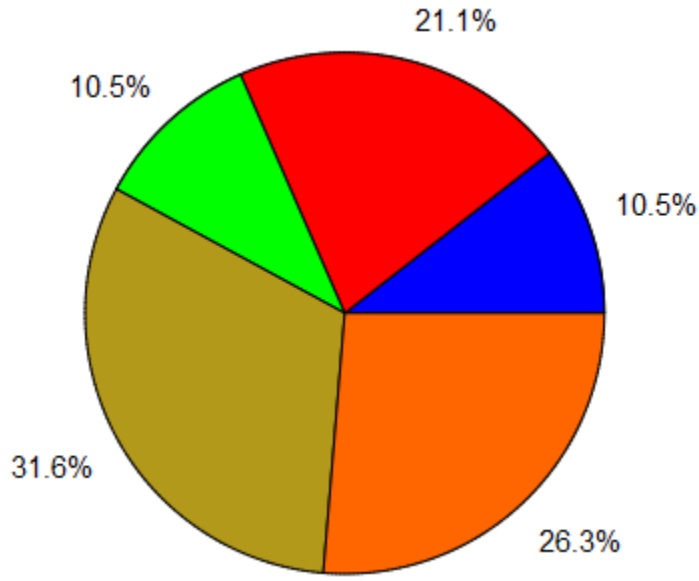
We create a 2D pie chart with pieces of the size ratios $1 : 2 : 1 : 3 : 2.5$.

```
p := plot::Piechart2d([1, 2, 1, 3, 2.5]):
plot(p)
```



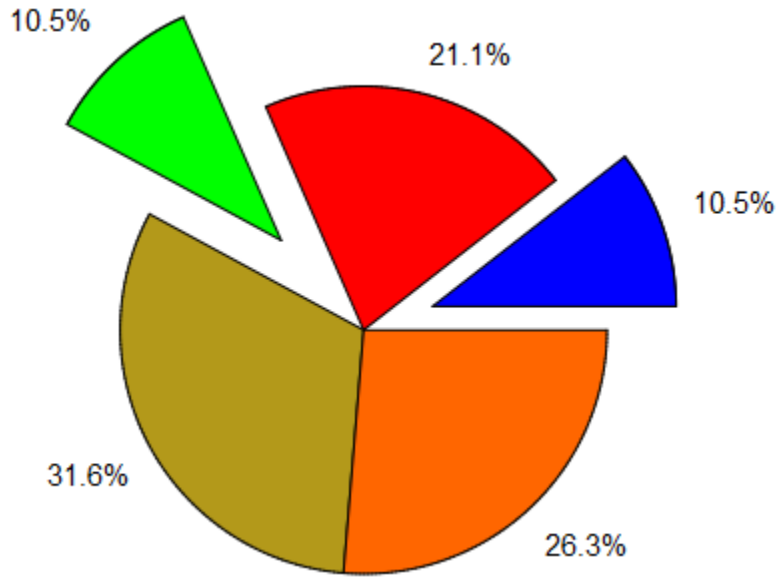
We set titles for the pieces:

```
p::Titles := ["10.5%", "21.1%", "10.5%", "31.6%", "26.3%"]:
plot(p)
```



Pieces can be moved away from the pie center with the attribute `Moves`. One or more moves can be given as a `list` of values `[f1, f2, ...]`. The "move factors" f_1, f_2 etc. are positive real values that represent fractions of the pie radius. The *i*-th piece is moved away from the center by f_i. If not all pieces are to be, one may specify `Moves = [n1 = f1, n2 = f2, ...]`, such that only the pieces with indices n_1, n_2 etc. are moved:

```
p::Moves := [1 = 0.3, 3 = 0.5]:
plot(p)
```

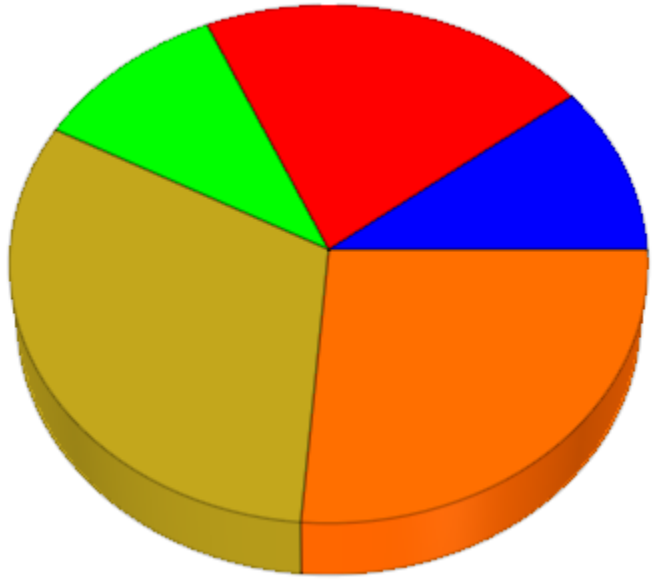



```
delete p:
```

Example 2

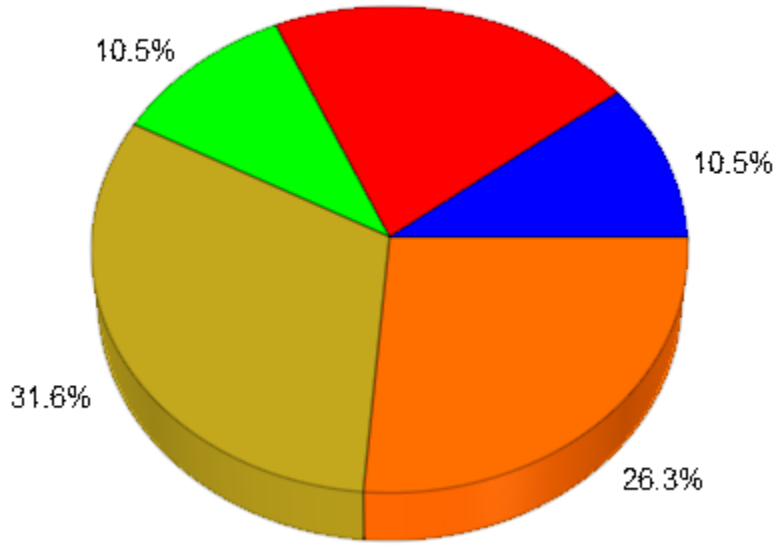
We create a 3D pie chart with pieces of the size ratios $1 : 2 : 1 : 3 : 2.5$.

```
p := plot::Piechart3d([1, 2, 1, 3, 2.5]):
plot(p)
```



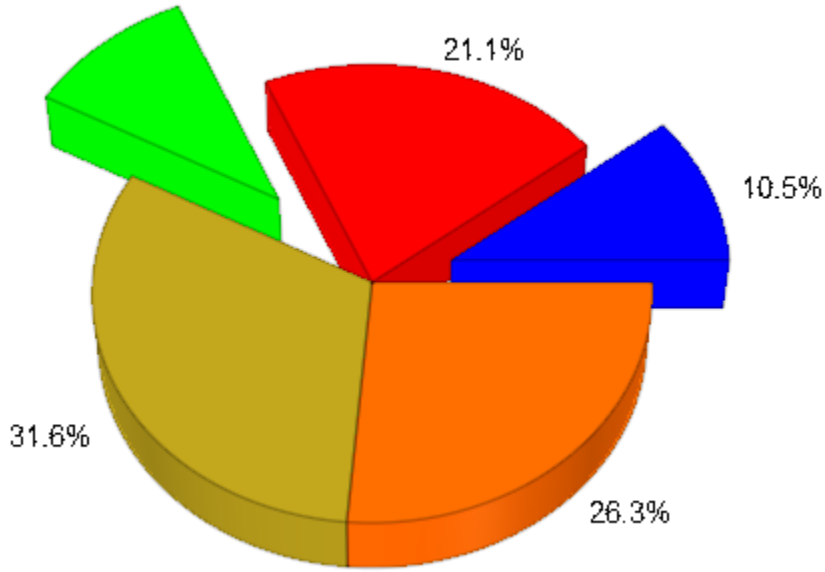
We set titles for the pieces:

```
p::Titles := ["10.5%", "21.1%", "10.5%", "31.6%", "26.3%"]:
plot(p)
```



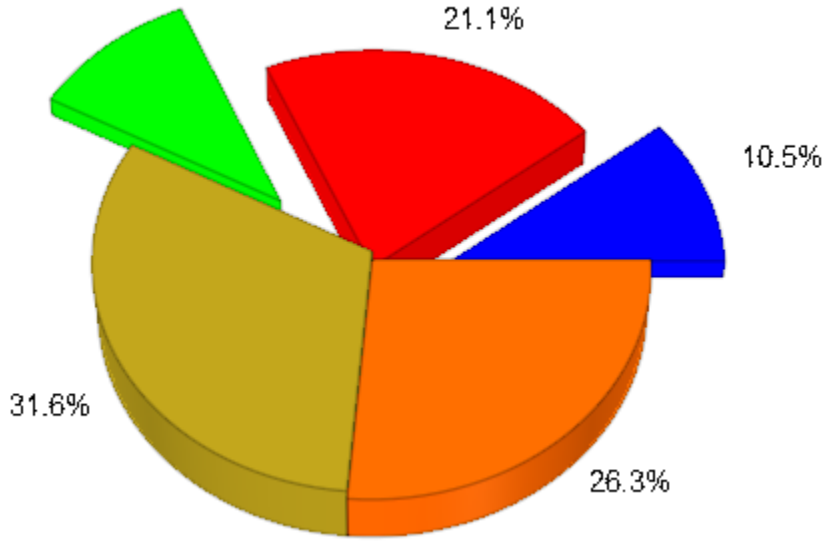
Some pieces are moved away from the center:

```
p::Moves := [1 = 0.3, 3 = 0.5]:
plot(p)
```



The heights of the pieces in a 3D pie chart can vary:

```
p::Heights := [0.1, 0.2, 0.1, 0.3, 0.25]:
plot(p)
```




```
delete p:
```

Example 3

A pie chart can be animated. We plot a pie chart with an animated radius. The pieces move in and out, changing their size:

```
m1 := piecewise([abs(a - PI/4) <= PI/4, sin(2*a)^2/3],
                [abs(a - PI/4) > PI/4, 0]):
m2 := piecewise([abs(a - 3*PI/4) <= PI/4, sin(2*a)^2/3],
                [abs(a - 3*PI/4) > PI/4, 0]):
m3 := piecewise([abs(a - 5*PI/4) <= PI/4, sin(2*a)^2/3],
                [abs(a - 5*PI/4) > PI/4, 0]):
m4 := piecewise([abs(a - 7*PI/4) <= PI/4, sin(2*a)^2/3],
                [abs(a - 7*PI/4) > PI/4, 0]):
p := plot::Piechart3d([5 + sin(a)/4, 2, 1 + sin(a)/2, 4],
                      Title = "crazy pie chart",
                      TitlePosition = [0, 15, 5],
                      TitleFont = [Italic, 18],
                      Center = [0, 0, 0],
                      Radius = 10 + sin(2*a),
                      Heights = [1.5 + sin(a), 1.5 + cos(2*a),
```

```
                                1.5 + sin(a), 1.5 + cos(4*a)],
                    Titles = [1 = "piece 1", 2 = "piece 2",
                              3 = "piece 3", 4 = "piece 4"],
                    Moves = [m1, m2, m3, m4],
                    a = 0..2*PI):
plot(p):
```


```
delete m1, m2, m3, m4, p:
```

Parameters

d_1, d_2, …

The sizes of the pieces: non-negative real values or arithmetical expressions of the animation parameter `a`.

d_1, d_2, … is equivalent to the attribute `Data`.

A

A `matrix` or `array` containing the data d_1, d_2 etc.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Bars2d` | `plot::Bars3d` | `plot::Boxplot` | `plot::Histogram2d` | `plot::Matrixplot` | `plot::Piechart3d`

More About

- "Create Bar Charts, Histograms, and Pie Charts"

plot::Piechart3d

3D pie charts

Syntax

`plot::Piechart3d([`d_1`, `d_2`, …], <a = `a_{min}` .. `a_{max}`>, options)`

`plot::Piechart3d(A, <a = `a_{min}` .. `a_{max}`>, options)`

Description

`plot::Piechart3d([`d_1`, `d_2`, `d_3`, …])` creates a corresponding 3D pie chart.

With the input data d_1, d_2 etc., the *i*-th piece of the pie has the opening angle `2*PI*d_i/(d_1 + d_2 + Symbol::dots)`.

The attribute `Titles` allows to attach titles to the pieces of the pie. In contrast to the overall title of the pie chart (`Title`, `TitleFont`), the titles of the pieces react to `TextFont`.

The attribute `Moves` allows to move the pieces away from the pie center for highlighting.

In 3D, the attribute `Heights` allows to vary the heights of the pieces.

The attributes `Center` and `Radius` allow to position and scale a pie chart relative to other graphical objects in the same scene.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Billboarding`	text orientation in space or towards observer	`TRUE`
`Center`	center of objects, rotation center	`[0, 0, 0]`

Attribute	Purpose	Default Value
CenterX	center of objects, rotation center, x-component	0
CenterY	center of objects, rotation center, y-component	0
CenterZ	center of objects, rotation center, z-component	0
Color	the main color	
Colors	list of colors to use	[RGB::Blue, RGB::Red, RGB::Green, RGB::MuPADGold, RGB::Orange, RGB::Cyan, RGB::Magenta, RGB::LimeGreen, RGB::CadmiumYellowLight, RGB::AlizarinCrimson]
Data	the (statistical) data to plot	[1]
Filled	filled or transparent areas and surfaces	TRUE
Frames	the number of frames in an animation	50
Heights	heights of pieces in pie charts	[0.3]
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Black.[0.25]
LineWidth	width of lines	0.35
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE

Attribute	Purpose	Default Value
`Moves`	displacements of pieces in pie charts	[0]
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`Radius`	radius of circles, spheres etc.	`1`
`TextFont`	font of text objects	`[" sans-serif ", 11]`
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`
`Title`	object title	
`Titles`	list of titles for object parts	`[" "]`
`TitleFont`	font of object titles	`[" sans-serif ", 11]`
`TitlePosition`	position of object titles	
`TitleAlignment`	horizontal alignment of titles w.r.t. their coordinates	`Center`
`TitlePositionX`	position of object titles, x component	
`TitlePositionY`	position of object titles, y component	
`TitlePositionZ`	position of object titles, z component	

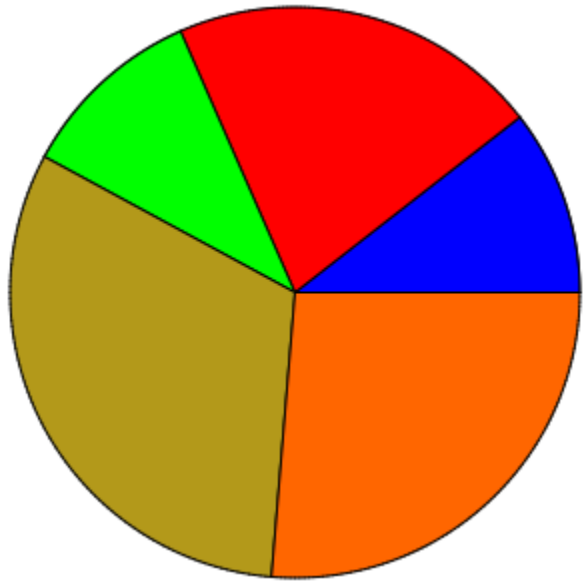
Attribute	Purpose	Default Value
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

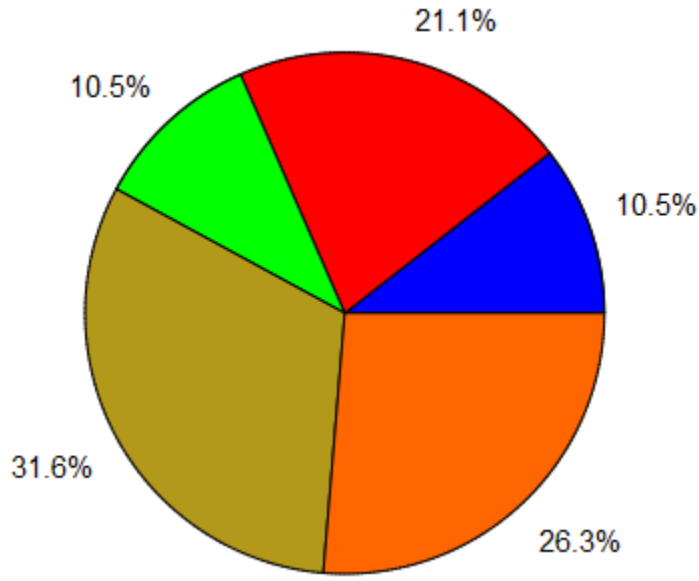
We create a 2D pie chart with pieces of the size ratios $1 : 2 : 1 : 3 : 2.5$.

```
p := plot::Piechart2d([1, 2, 1, 3, 2.5]):
plot(p)
```



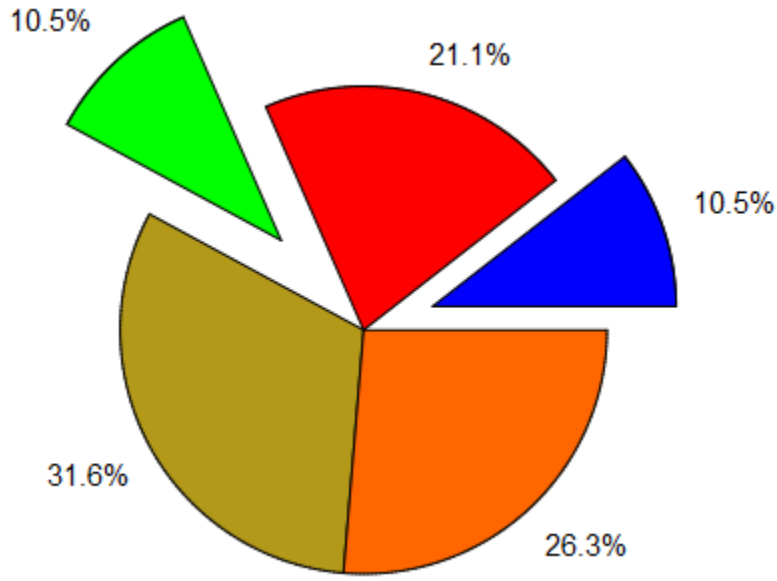
We set titles for the pieces:

```
p::Titles := ["10.5%", "21.1%", "10.5%", "31.6%", "26.3%"]:
plot(p)
```



Pieces can be moved away from the pie center with the attribute `Moves`. One or more moves can be given as a `list` of values `[f1, f2, ...]`. The "move factors" f_1, f_2 etc. are positive real values that represent fractions of the pie radius. The *i*-th piece is moved away from the center by f_i. If not all pieces are to be, one may specify `Moves = [n1 = f1, n2 = f2, ...]`, such that only the pieces with indices n_1, n_2 etc. are moved:

```
p::Moves := [1 = 0.3, 3 = 0.5]:
plot(p)
```

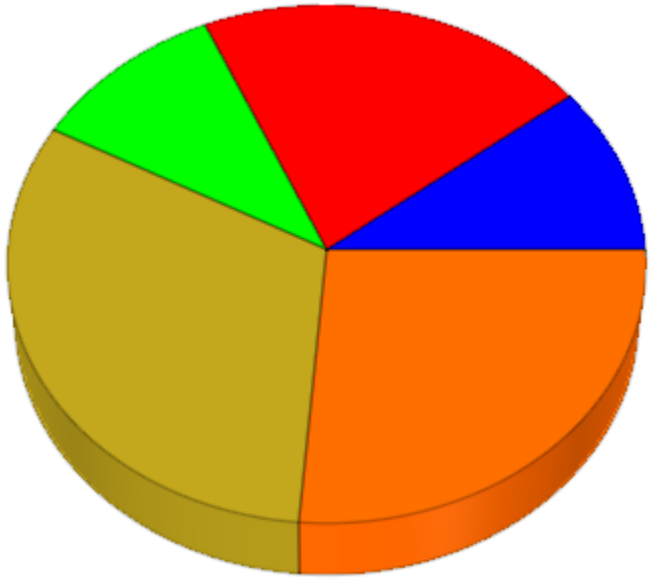



```
delete p:
```

Example 2

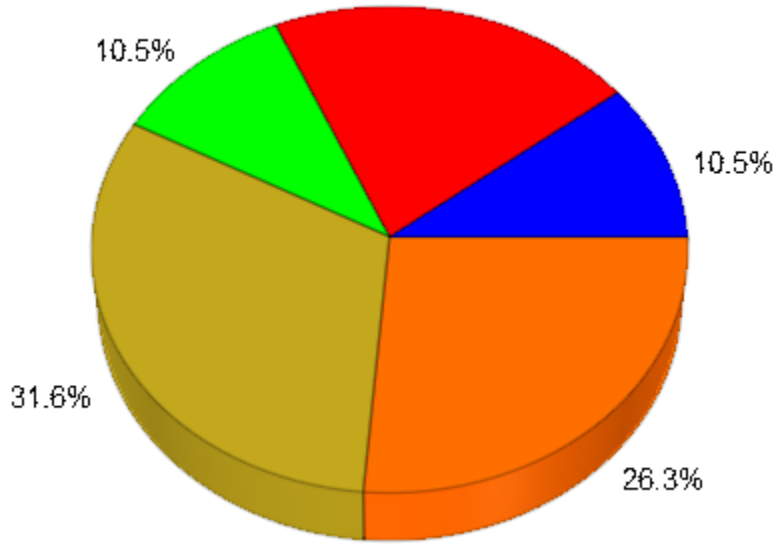
We create a 3D pie chart with pieces of the size ratios $1 : 2 : 1 : 3 : 2.5$.

```
p := plot::Piechart3d([1, 2, 1, 3, 2.5]):
plot(p)
```



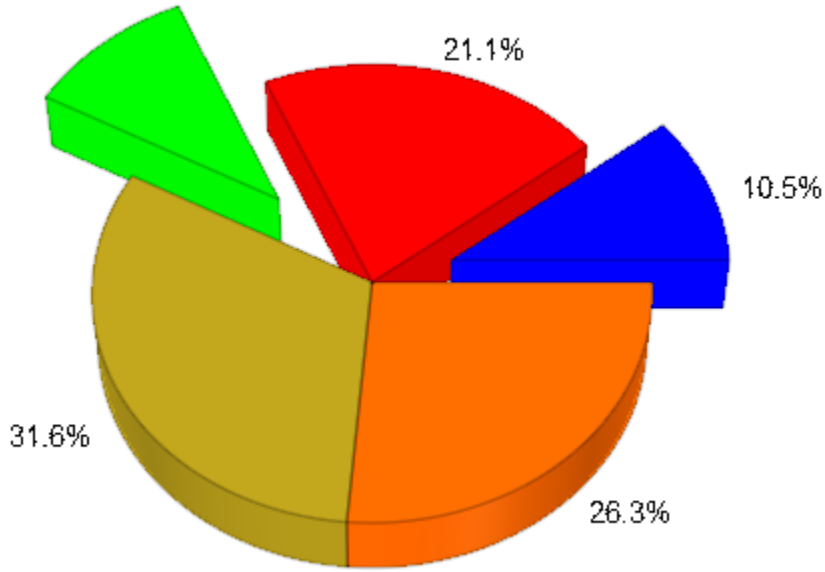
We set titles for the pieces:

```
p::Titles := ["10.5%", "21.1%", "10.5%", "31.6%", "26.3%"]:
plot(p)
```



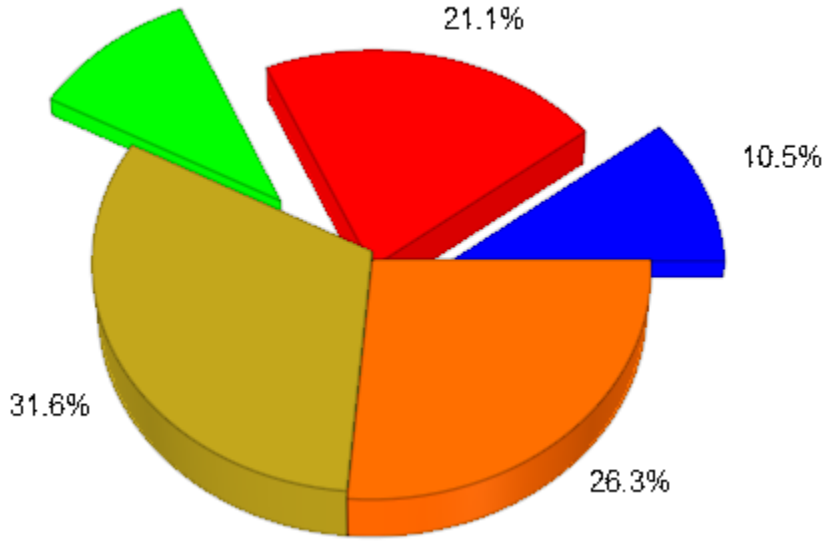
Some pieces are moved away from the center:

```
p::Moves := [1 = 0.3, 3 = 0.5]:
plot(p)
```



The heights of the pieces in a 3D pie chart can vary:

```
p::Heights := [0.1, 0.2, 0.1, 0.3, 0.25]:
plot(p)
```




```
delete p:
```

Example 3

A pie chart can be animated. We plot a pie chart with an animated radius. The pieces move in and out, changing their size:

```
m1 := piecewise([abs(a - PI/4) <= PI/4, sin(2*a)^2/3],
                [abs(a - PI/4) > PI/4, 0]):
m2 := piecewise([abs(a - 3*PI/4) <= PI/4, sin(2*a)^2/3],
                [abs(a - 3*PI/4) > PI/4, 0]):
m3 := piecewise([abs(a - 5*PI/4) <= PI/4, sin(2*a)^2/3],
                [abs(a - 5*PI/4) > PI/4, 0]):
m4 := piecewise([abs(a - 7*PI/4) <= PI/4, sin(2*a)^2/3],
                [abs(a - 7*PI/4) > PI/4, 0]):
p := plot::Piechart3d([5 + sin(a)/4, 2, 1 + sin(a)/2, 4],
                      Title = "crazy pie chart",
                      TitlePosition = [0, 15, 5],
                      TitleFont = [Italic, 18],
                      Center = [0, 0, 0],
                      Radius = 10 + sin(2*a),
                      Heights = [1.5 + sin(a), 1.5 + cos(2*a),
```

```
                          1.5 + sin(a), 1.5 + cos(4*a)],
                Titles = [1 = "piece 1", 2 = "piece 2",
                          3 = "piece 3", 4 = "piece 4"],
                Moves = [m1, m2, m3, m4],
                a = 0..2*PI):
plot(p):
```


```
delete m1, m2, m3, m4, p:
```

Parameters

d_1, d_2, …

The sizes of the pieces: non-negative real values or arithmetical expressions of the animation parameter `a`.

d_1, d_2, … is equivalent to the attribute `Data`.

A

A `matrix` or `array` containing the data d_1, d_2 etc.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions

`plot` | `plot::copy`

MuPAD Graphical Primitives

`plot::Bars2d` | `plot::Bars3d` | `plot::Boxplot` | `plot::Histogram2d` | `plot::Matrixplot` | `plot::Piechart2d`

More About

- "Create Bar Charts, Histograms, and Pie Charts"

plot::Plane

Infinite plane in 3D

Syntax

```
plot::Plane([x, y, z], <[n_x, n_y, n_z]>, <a = a_min .. a_max>, options)

plot::Plane(X, <N>, <a = a_min .. a_max>, options)

plot::Plane(XN, <a = a_min .. a_max>, options)

plot::Plane(p_1, p_2, p_3, <a = a_min .. a_max>, options)

plot::Plane(p_123, <a = a_min .. a_max>, options)
```

Description

`plot::Plane(x, n)` creates the (infinite) plane with normal vector n passing through the point x.

`plot::Plane` provides a graphical plane in 3D that does not require a specification, which part of the plane is to be seen in the picture. The visible part of the plane is determined automatically by the `ViewingBox` of the entire 3D scene.

The contribution of a plane of type `plot::Plane` to the `ViewingBox` of a 3D scene consists only of the single point `[x, y, z]` (this is p_1, if the plane is specified by three points p_1, p_2, p_3 on the plane).

Thus, two planes with the same normal but different points may be mathematically equivalent, but may produce different pictures due to different viewing boxes. Cf. “Example 3” on page 24-646.

By default, a mesh of lines is displayed on the plane. Use the attribute `Mesh = [n_1, n_2]` with positive integer values n_1, n_2 to control the number of mesh lines.

Attributes

Attribute	Purpose	Default Value
AffectViewingBox	influence of objects on the ViewingBox of a scene	TRUE
Color	the main color	RGB::LightBlue
Filled	filled or transparent areas and surfaces	TRUE
FillColor	color of areas and surfaces	RGB::LightBlue
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Black.[0.25]
LinesVisible	visibility of lines	TRUE
Mesh	number of sample points	[15, 15]
Name	the name of a plot object (for browser and legend)	
Normal	normal vector of circles and discs, etc. in 3D	[0, 0, 1]
NormalX	normal vector of circles and discs, etc. in 3D, x-component	0
NormalY	normal vector of circles and discs, etc. in 3D, y-component	0
NormalZ	normal vector of circles and discs, etc. in 3D, z-component	1

Attribute	Purpose	Default Value
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`Position`	positions of cameras, lights, and text objects	`[0, 0, 0]`
`PositionX`	x-positions of cameras, lights, and text objects	`0`
`PositionY`	y-positions of cameras, lights, and text objects	`0`
`PositionZ`	z-positions of cameras, lights, and text objects	`0`
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`
`Title`	object title	
`TitleFont`	font of object titles	`[" sans-serif ", 11]`
`TitlePosition`	position of object titles	
`TitleAlignment`	horizontal alignment of titles w.r.t. their coordinates	`Center`
`TitlePositionX`	position of object titles, x component	
`TitlePositionY`	position of object titles, y component	
`TitlePositionZ`	position of object titles, z component	

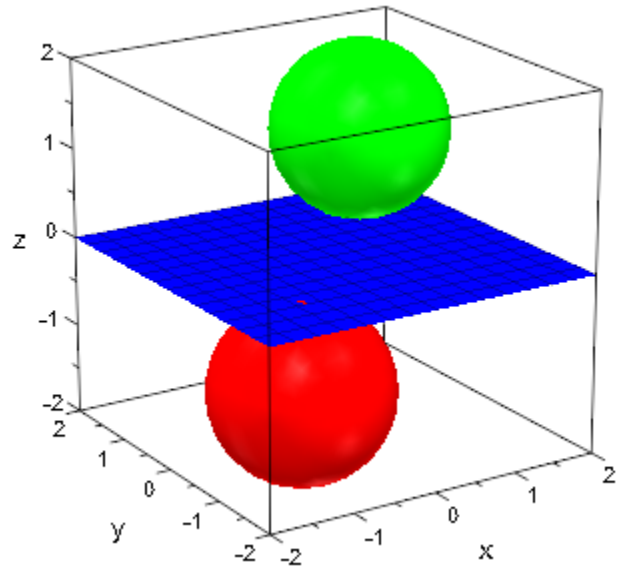
Attribute	Purpose	Default Value
UMesh	number of sample points for parameter “u”	15
VMesh	number of sample points for parameter “v”	15
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE

Examples

Example 1

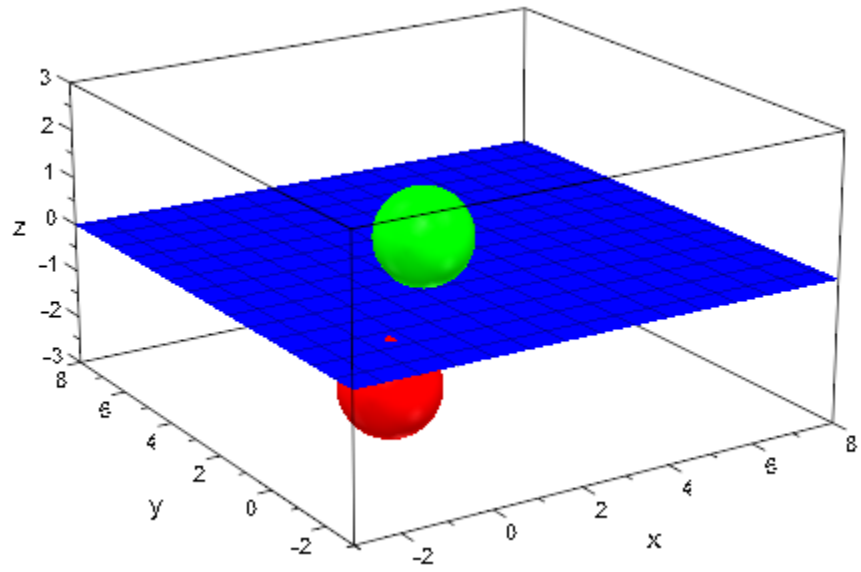
We generate two spheres and a plane:

```
plot(plot::Sphere(1, [-1, -1, -1], Color = RGB::Red),
     plot::Sphere(1, [ 1,  1,  1], Color = RGB::Green),
     plot::Plane([0, 0, 0], [0, 0, 1], Color = RGB::Blue)):
```



We specify an explicit `ViewingBox` for the scene:

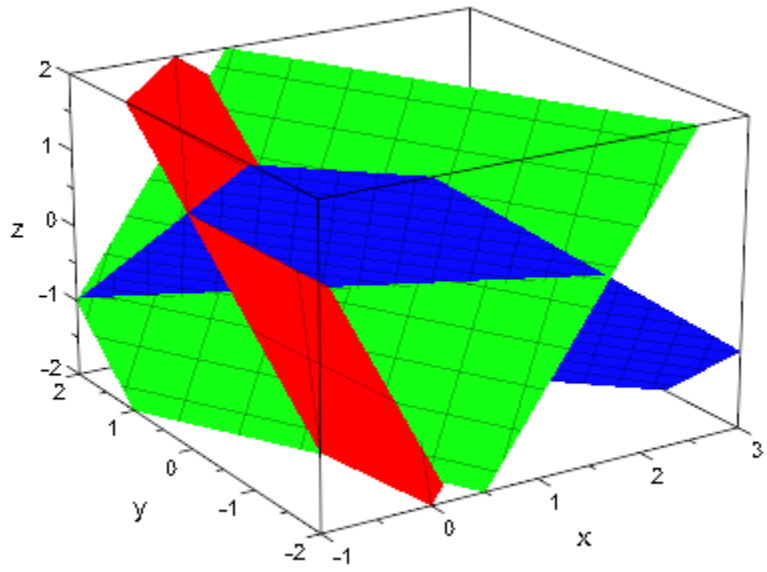
```
plot(plot::Sphere(1, [-1, -1, -1], Color = RGB::Red),
     plot::Sphere(1, [ 1,  1,  1], Color = RGB::Green),
     plot::Plane([0, 0, 0], [0, 0, 1], Color = RGB::Blue),
     ViewingBox = [-3..8, -3..8, -3..3]):
```



Example 2

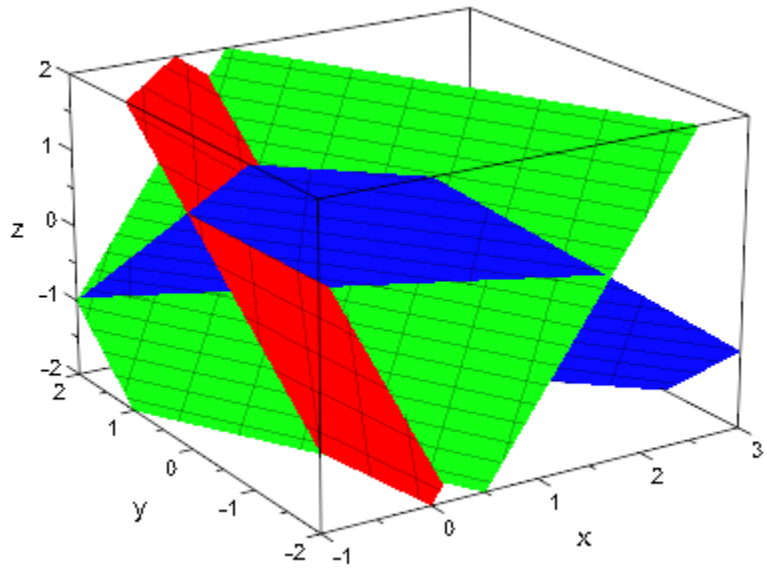
We demonstrate the effect of the attribute `Mesh` that controls the number of mesh lines displayed on planes:

```
plot(plot::Plane([0, 0, 0], [1, -1, 1], Color = RGB::Red,
                 Mesh = [5, 5]),
     plot::Plane([0, 1, 0], [2, 1, -1], Color = RGB::Green,
                 Mesh = [10, 10]),
     plot::Plane([1, -1, 0], [1, 1, 1], Color = RGB::Blue,
                 Mesh = [20, 20]),
     ViewingBox = [-1..3, -2..2, -2..2])
```



We change the number of mesh lines:

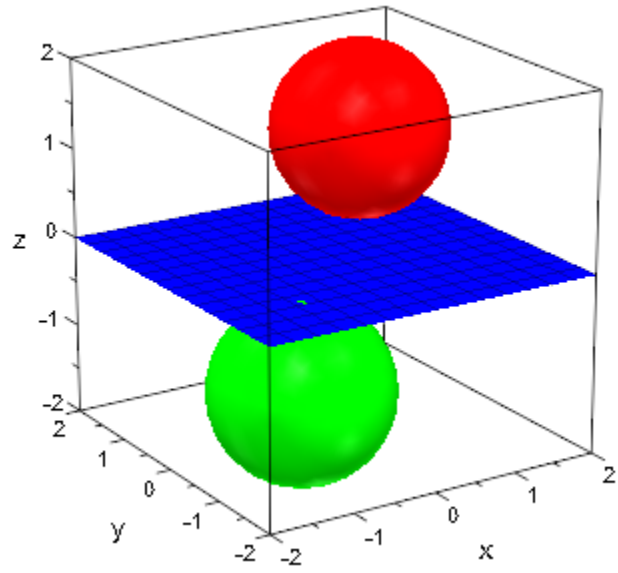
```
plot(plot::Plane([0, 0, 0], [1, -1, 1], Color = RGB::Red,
                 Mesh = [10, 10]),
     plot::Plane([0, 1, 0], [2, 1, -1], Color = RGB::Green,
                 Mesh = [20, 10]),
     plot::Plane([1, -1, 0], [1, 1, 1], Color = RGB::Blue,
                 Mesh = [15, 5]),
     ViewingBox = [-1..3, -2..2, -2..2])
```



Example 3

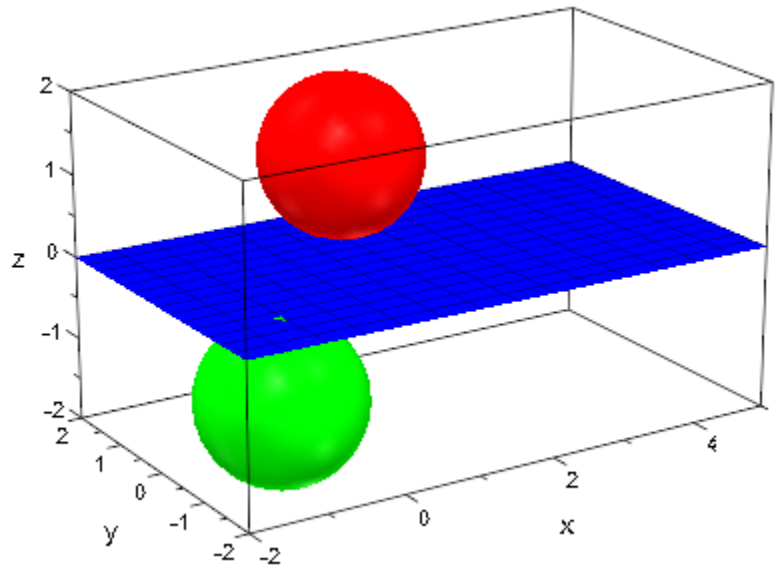
The contribution of a plane to the automatic `ViewingBox` of the whole scene consists only of the point used to specify the plane. In the following scene, this point is the origin. It lies inside the `ViewingBox` generated by the two spheres. Thus, the `ViewingBox` of the scene is determined by the two spheres only:

```
plot(plot::Sphere(1, [1, 1, 1], Color = RGB::Red),
     plot::Sphere(1, [-1, -1, -1], Color = RGB::Green),
     plot::Plane([0, 0, 0], [0, 0, 1], Color = RGB::Blue)):
```



Now, a different point `[5, 0, 0]` is used to specify the same plane. It does not lie inside the `ViewingBox` generated by the two spheres and thus enlarges the `ViewingBox` of the scene:

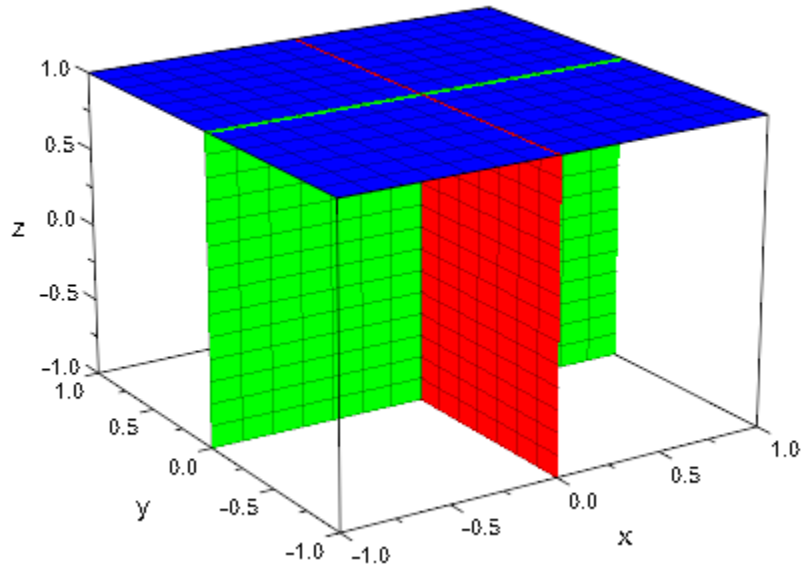
```
plot(plot::Sphere(1, [1, 1, 1], Color = RGB::Red),
     plot::Sphere(1, [-1, -1, -1], Color = RGB::Green),
     plot::Plane([5, 0, 0], [0, 0, 1], Color = RGB::Blue)):
```



Example 4

We create animated planes:

```
plot(plot::Plane([0, 0, 0], [cos(a), sin(a), 0], a = 0..PI,
                 Color = RGB::Red),
     plot::Plane([0, 0, 0], [0, cos(a), sin(a)], a = 0..PI,
                 Color = RGB::Green),
     plot::Plane([0, 0, a], [0, 0, 1], a = 0..1,
                 Color = RGB::Blue),
     ViewingBox = [-1..1, -1..1, -1..1])
```



Parameters

x, y, z

The coordinates of a point on the plane: numerical real values or arithmetical expressions in the animation parameter `a`.

x, y, z are equivalent to the attributes `PositionX`, `PositionY`, `PositionZ`.

n_x, n_y, n_z

The components of the normal vector; n_x, n_y, n_z must be numerical real values or arithmetical expressions in the animation parameter `a`. If no normal is specified, the normal (0, 0, 1) is used.

n_x, n_y, n_z are equivalent to the attributes `NormalX`, `NormalY`, `NormalZ`.

X

A matrix of category `Cat::Matrix` with three entries that provide the coordinates `x`, `y`, `z` of a point on the plane.

X is equivalent to the attribute `Position`.

`N`

A matrix of category `Cat::Matrix` with three entries that provide the components n_x, n_y, n_z of the normal.

N is equivalent to the attribute `Normal`.

`XN`

A matrix of category `Cat::Matrix` with 3 rows and 2 columns. The first column provides the coordinates `x`, `y`, `z` of a point on the plane, the second column provides the components n_x, n_y, n_z of the normal.

XN is equivalent to the attributes `Position`, `Normal`.

p_1, p_2, p_3

Three points on the plane: either lists with 3 entries each or matrices of category `Cat::Matrix` with 3 entries each. The point p_1 correponds to the attribute `Position`, the normal of the plane (the attribute `Normal`) is computed as the cross product $(p_2 - p_1) \times (p_3 - p_1)$.

p_{123}

A matrix of category `Cat::Matrix` with 3 rows and 3 columns. Each column corresponds to a point on the plane.

`a`

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Parallelogram3d` | `plot::Surface`

plot::Point2d

2D points

Syntax

```
plot::Point2d(x, y, <a = a_min .. a_max>, options)

plot::Point2d([x, y], <a = a_min .. a_max>, options)

plot::Point2d(matrix([x, y]), <a = a_min .. a_max>, options)
```

Description

`plot::Point2d(x, y)` creates a two-dimensional point with the coordinates (x, y).

`plot::Point2d` creates graphical points in two dimensions.

Starting with MuPAD 3.0 software, each type of graphical elements has a fixed dimension. Therefore, `plot::Point2d` and `plot::Point3d` are distinct, but very similar, types.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`
`Color`	the main color	`RGB::MidnightBlue`
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	

Attribute	Purpose	Default Value
`LegendEntry`	add this object to the legend?	`FALSE`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`PointSize`	the size of points	`1.5`
`PointColor`	the color of points	`RGB::MidnightBlue`
`PointStyle`	the presentation style of points	`FilledCircles`
`Position`	positions of cameras, lights, and text objects	`[0, 0]`
`PositionX`	x-positions of cameras, lights, and text objects	`0`
`PositionY`	y-positions of cameras, lights, and text objects	`0`
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`
`Title`	object title	
`TitleFont`	font of object titles	`[" sans-serif ", 11]`
`TitlePosition`	position of object titles	
`TitleAlignment`	horizontal alignment of titles w.r.t. their coordinates	`Center`

Attribute	Purpose	Default Value
`TitlePositionX`	position of object titles, x component	
`TitlePositionY`	position of object titles, y component	
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

We create three points:

```
p1 := plot::Point2d(1, 3, PointSize = 4*unit::mm);
p2 := plot::Point2d(2, 2, PointSize = 5*unit::mm);
p3 := plot::Point2d(3, 1, Color = RGB::Green,
                    PointSize = 6*unit::mm);
```

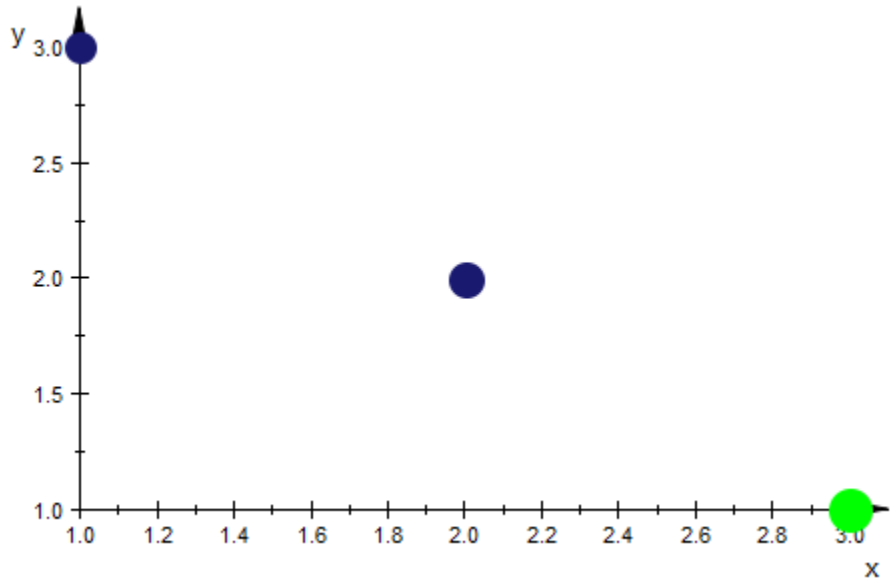
plot::Point2d(1, 3, PointSize = 4)

plot::Point2d(2, 2, PointSize = 5)

plot::Point2d(3, 1, PointColor = RGB::Green, PointSize = 6)

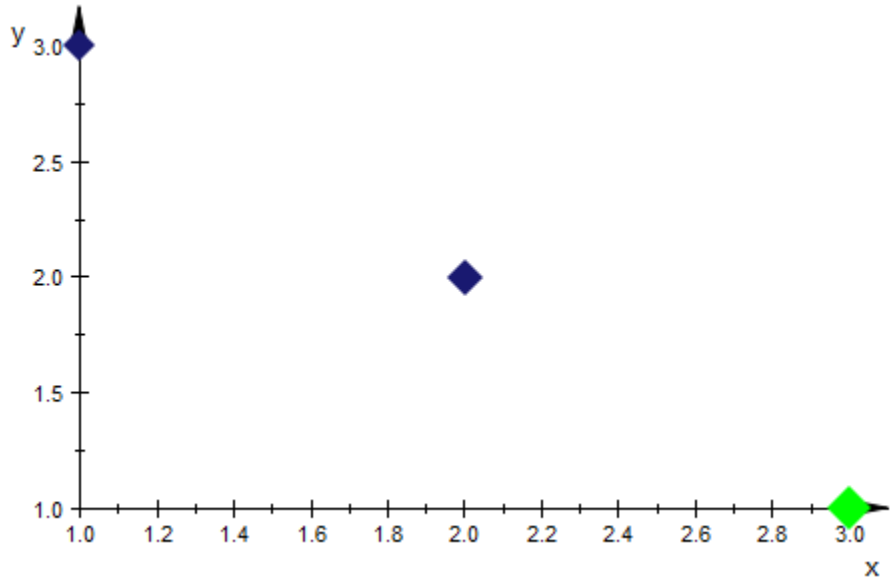
To have these points displayed, we use `plot`:

```
plot(p1, p2, p3)
```



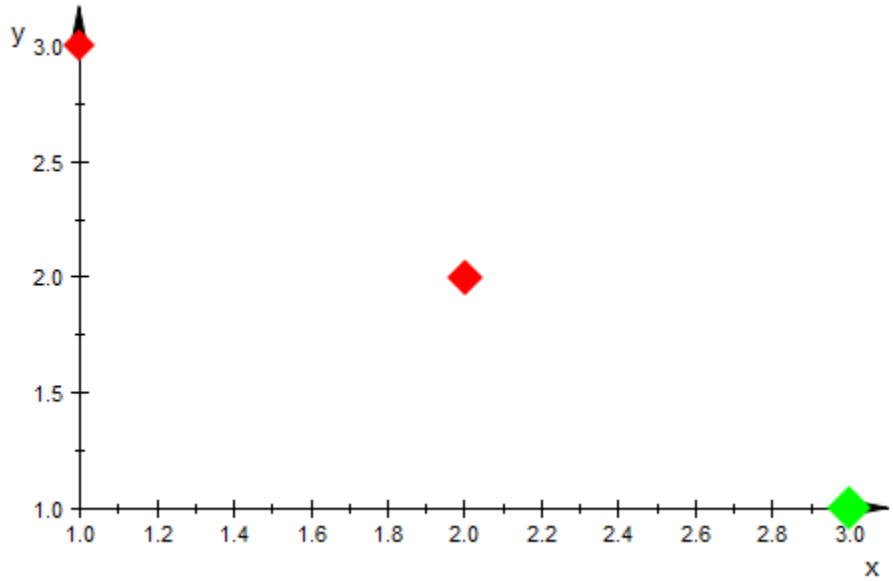
It is possible to set global options directly in the call to `plot`:

```
plot(p1, p2, p3, PointStyle = FilledDiamonds)
```



These options are regarded as the new *defaults*. This implies that objects having an option set explicitly will silently ignore these options. The green point stays green:

```
plot(p1, p2, p3, PointStyle = FilledDiamonds,
     PointColor = RGB::Red)
```



Example 2

The point position can be animated. As an example, we combine a point with a curve that traces the path of the point:

```
x := t -> sin(3*t);
y := t -> cos(5*t);
p := plot::Point2d([x(t), y(t)], t = 0..2*PI);
c := plot::Curve2d([x(t), y(t)], t = 0..tmax, tmax = 0..2*PI)
```

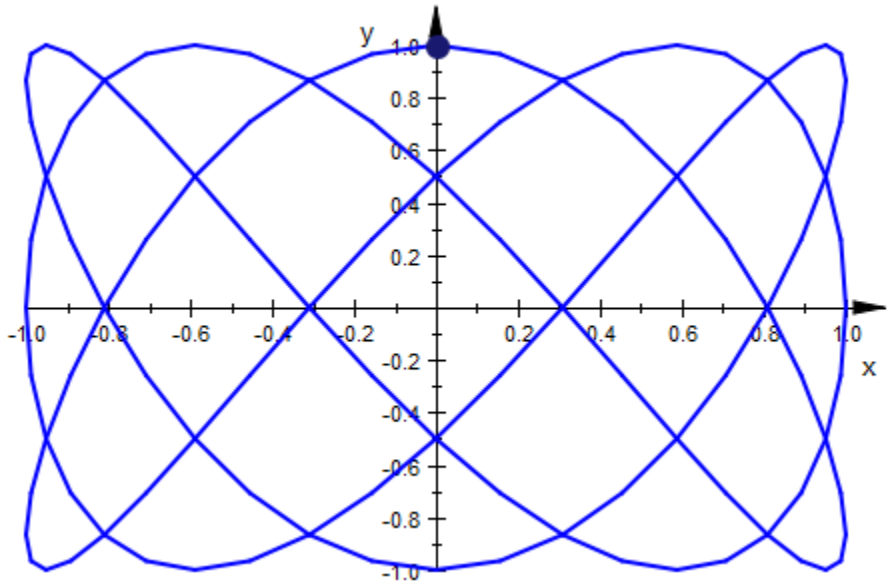
$$t \to \sin(3\,t)$$

$$t \to \cos(5\,t)$$

$$\text{plot::Point2d}(\sin(3\,t),\ \cos(5\,t),\ t = 0..2\,\pi)$$

$$\text{plot::Curve2d}([\sin(3\,t),\ \cos(5\,t)],\ t = 0..\text{tmax})$$

```
plot(c, p, PointSize = 3*unit::mm, LineWidth = 0.5*unit::mm)
```



Parameters

x, y

Arithmetical expressions

x, y are equivalent to the attributes `Position`, `PositionX`, `PositionY`.

a

Animation parameter, specified as a = $a_{min}..a_{max}$, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

Algorithms

For reasons of efficiency and clarity in the object browser, you should avoid generating large numbers of `plot::Point2d` and `plot::Point3d` objects. None of the

domains in the plot package do. For alternatives, consider `plot::PointList2d` and `plot::PointList3d`.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Point3d` | `plot::Polygon2d` | `plot::Polygon3d`

plot::Point3d

3D points

Syntax

```
plot::Point3d(x, y, z, <a = a_min .. a_max>, options)
```

```
plot::Point3d([x, y, z], <a = a_min .. a_max>, options)
```

```
plot::Point3d(matrix([x, y, z]), <a = a_min .. a_max>, options)
```

Description

`plot::Point3d(x, y, z)` creates a three-dimensional point with the coordinates (*x*, *y*, *z*).

`plot::Point3d` creates graphical points in three dimensions.

Starting with MuPAD 3.0 software, each type of graphical elements has a fixed dimension. Therefore, `plot::Point2d` and `plot::Point3d` are distinct, but very similar, types.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Color`	the main color	`RGB::MidnightBlue`
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`

Attribute	Purpose	Default Value
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`PointSize`	the size of points	`1.5`
`PointColor`	the color of points	`RGB::MidnightBlue`
`PointStyle`	the presentation style of points	`FilledCircles`
`Position`	positions of cameras, lights, and text objects	[0, 0, 0]
`PositionX`	x-positions of cameras, lights, and text objects	`0`
`PositionY`	y-positions of cameras, lights, and text objects	`0`
`PositionZ`	z-positions of cameras, lights, and text objects	`0`
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`
`Title`	object title	
`TitleFont`	font of object titles	`[" sans-serif ", 11]`
`TitlePosition`	position of object titles	
`TitleAlignment`	horizontal alignment of titles w.r.t. their coordinates	`Center`

Attribute	Purpose	Default Value
`TitlePositionX`	position of object titles, x component	
`TitlePositionY`	position of object titles, y component	
`TitlePositionZ`	position of object titles, z component	
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

We create three points:

```
p1 := plot::Point2d(1, 3, PointSize = 4*unit::mm);
p2 := plot::Point2d(2, 2, PointSize = 5*unit::mm);
p3 := plot::Point2d(3, 1, Color = RGB::Green,
                    PointSize = 6*unit::mm);
```

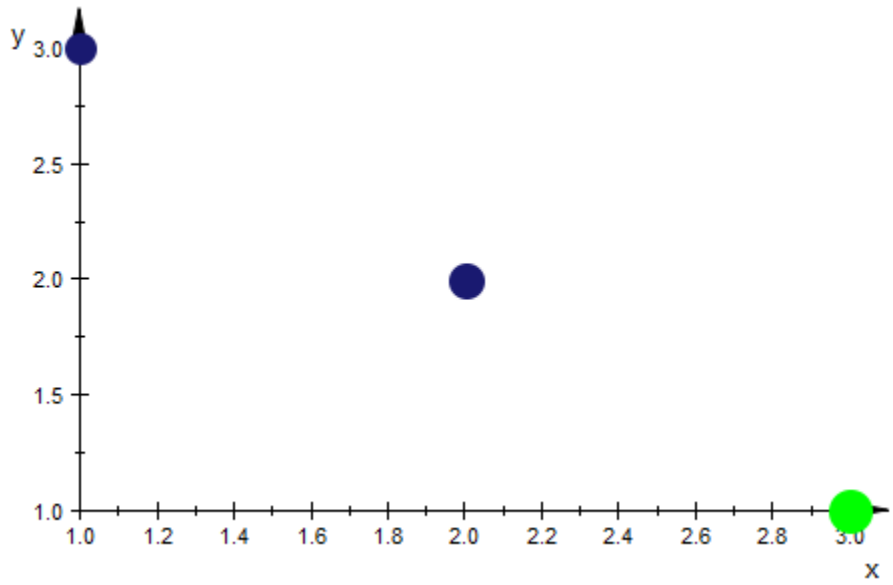
plot::Point2d(1, 3, PointSize = 4)

plot::Point2d(2, 2, PointSize = 5)

```
plot::Point2d(3, 1, PointColor = RGB::Green, PointSize = 6)
```

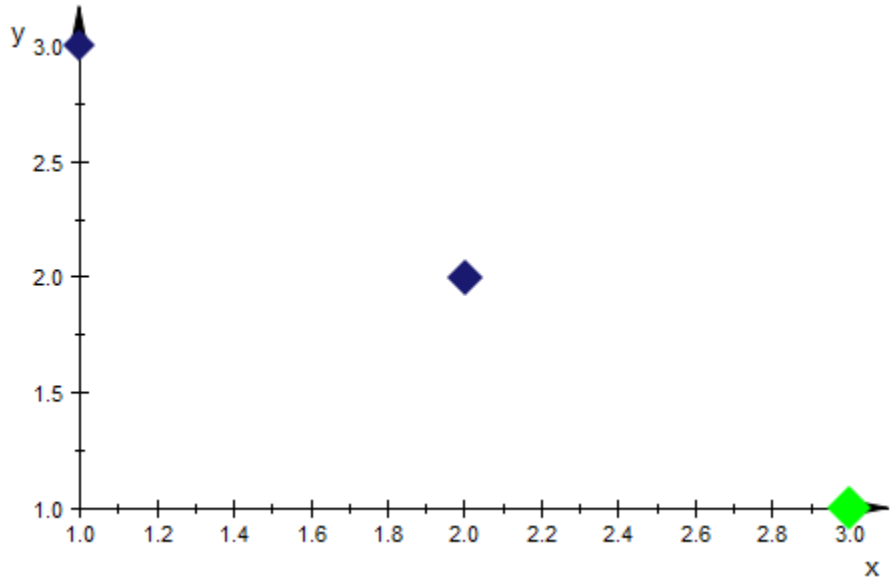
To have these points displayed, we use `plot`:

```
plot(p1, p2, p3)
```



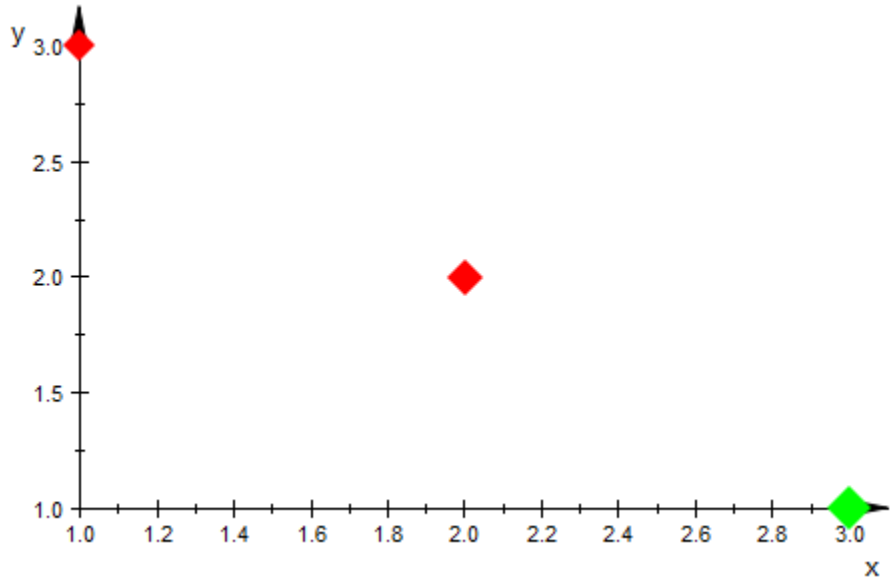
It is possible to set global options directly in the call to `plot`:

```
plot(p1, p2, p3, PointStyle = FilledDiamonds)
```



These options are regarded as the new *defaults*. This implies that objects having an option set explicitly will silently ignore these options. The green point stays green:

```
plot(p1, p2, p3, PointStyle = FilledDiamonds,
     PointColor = RGB::Red)
```



Example 2

The point position can be animated. As an example, we combine a point with a curve that traces the path of the point:

```
x := t -> sin(3*t);
y := t -> cos(5*t);
p := plot::Point2d([x(t), y(t)], t = 0..2*PI);
c := plot::Curve2d([x(t), y(t)], t = 0..tmax, tmax = 0..2*PI)
```

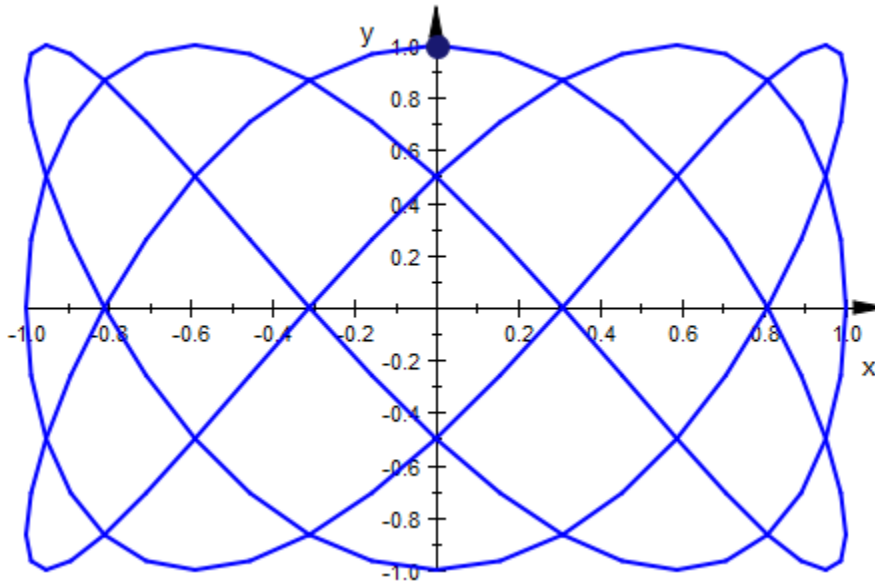
$$t \to \sin(3\,t)$$

$$t \to \cos(5\,t)$$

$$\text{plot::Point2d}(\sin(3\,t), \cos(5\,t), t = 0..2\,\pi)$$

$$\text{plot::Curve2d}([\sin(3\,t), \cos(5\,t)], t = 0..\text{tmax})$$

```
plot(c, p, PointSize = 3*unit::mm, LineWidth = 0.5*unit::mm)
```



Parameters

x, y, z

Arithmetical expressions

x, y, z are equivalent to the attributes `Position`, `PositionX`, `PositionY`, `PositionZ`.

a

Animation parameter, specified as a = $a_{min}..a_{max}$, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

Algorithms

For reasons of efficiency and clarity in the object browser, you should avoid generating large numbers of `plot::Point2d` and `plot::Point3d` objects. None of the

domains in the plot package do. For alternatives, consider `plot::PointList2d` and `plot::PointList3d`.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Point2d` | `plot::Polygon2d` | `plot::Polygon3d`

plot::PointList2d

Finite lists of 2D points

Syntax

`plot::PointList2d(pts, <a = a_min .. a_max>, options)`

`plot::PointList2d(M_2d, <a = a_min .. a_max>, options)`

Description

`plot::PointList2d` holds lists of points in 2D.

These types are containers for a (large) finite number of points. They let you avoid constructing large numbers of objects of type `plot::Point2d` for two reasons. First, the point types have non-negligible overhead and constructing and plotting a large number of them (say, five thousand) takes more time than plotting the same number of points in a single container object. Second, and this may be even more important, having five thousand points in the object browser takes a significant amount of memory and is not as lucid as having a single point list displayed there.

The attribute `Points2d` is displayed in the inspector in the user interface only for short lists.

`plot::PointList2d, PointList3d` internally use lists for storing the points. It is therefore not advisable to add a large number of points one-by-one. See “Example 2” on page 24-673 for a better method of collecting data.

If you specify the color of one point, you must specify the colors of all other points in the list. See “Example 3” on page 24-674.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`

Attribute	Purpose	Default Value
`Color`	the main color	`RGB::MidnightBlue`
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`Points2d`	list of 2D points	
`PointSize`	the size of points	`1.5`
`PointColor`	the color of points	`RGB::MidnightBlue`
`PointStyle`	the presentation style of points	`FilledCircles`
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`
`Title`	object title	
`TitleFont`	font of object titles	`[" sans-serif ", 11]`
`TitlePosition`	position of object titles	

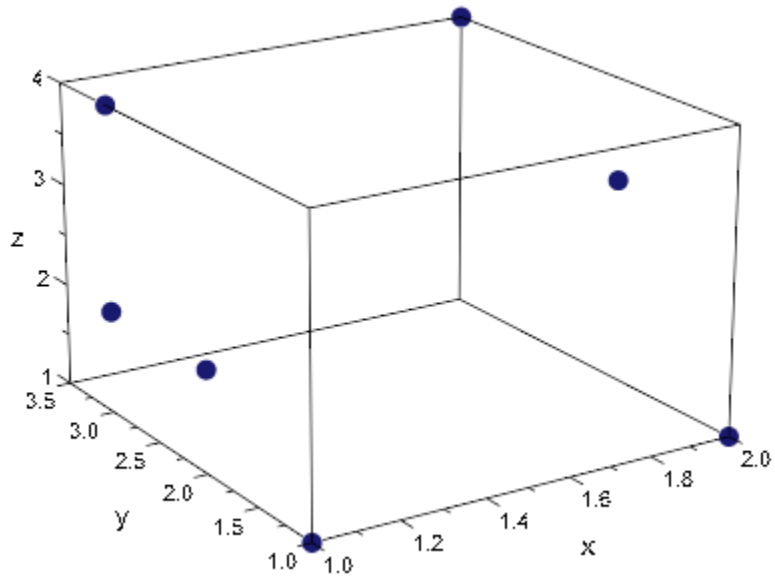
Attribute	Purpose	Default Value
`TitleAlignment`	horizontal alignment of titles w.r.t. their coordinates	`Center`
`TitlePositionX`	position of object titles, x component	
`TitlePositionY`	position of object titles, y component	
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

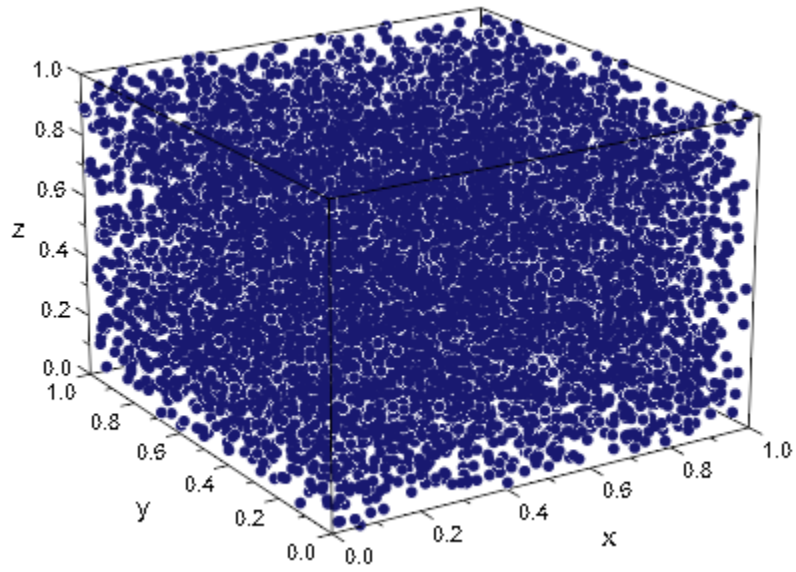
`plot::PointList3d` provides a basic form of scatter plot:

```
plot(plot::PointList3d([[1,1,1], [1,2,2], [1,3,2], [1,3,4],
                        [2,1,1], [2,2,3], [2,3.5, 4]],
                       PointSize=5))
```

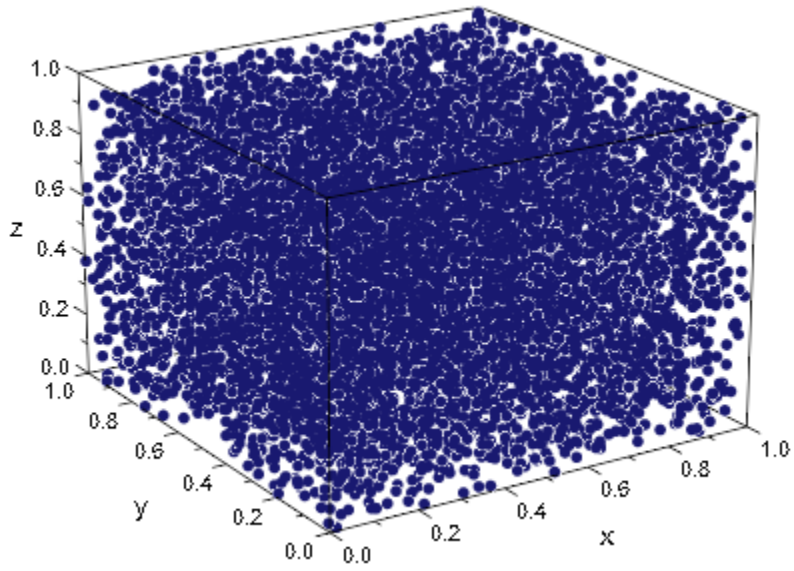


We can use this, for example, to get a visual test of random number generators:

```
r := frandom(0):
plot(plot::PointList3d([[r(), r(), r()] $ i=1..10000])):
```

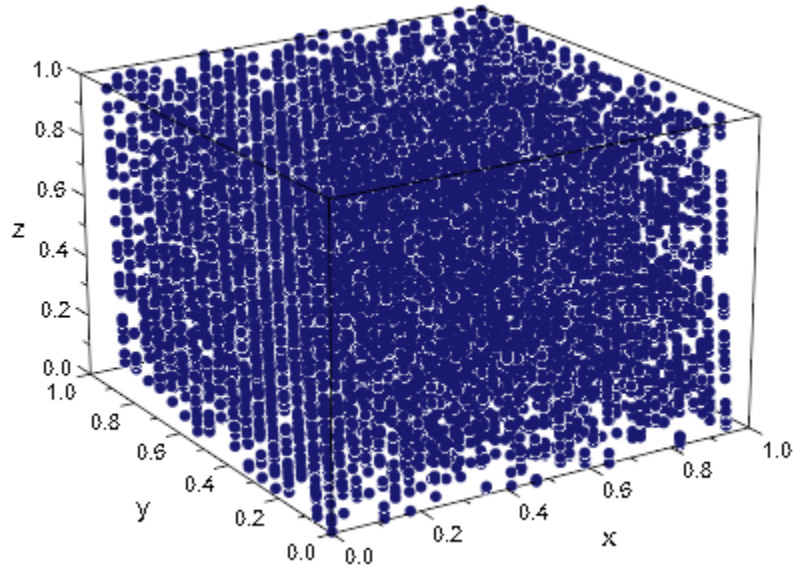



```
r := random(10^10)*1e-10:
plot(plot::PointList3d([[r(), r(), r()] $ i=1..10000])):
```



`frandom` and `random` fill the cube nicely, without noticeable patterns. The following generator, however, should probably not be used:

```
randseed := 12345:
r := proc()
     begin
       randseed := (randseed * 17 + 8) mod 10^10:
       1e-10 * randseed;
     end:
plot(plot::PointList3d([[r(), r(), r()] $ i=1..10000])):
```



Example 2

The following iteration leads to the so-called Hénon attractor (from chaos theory):

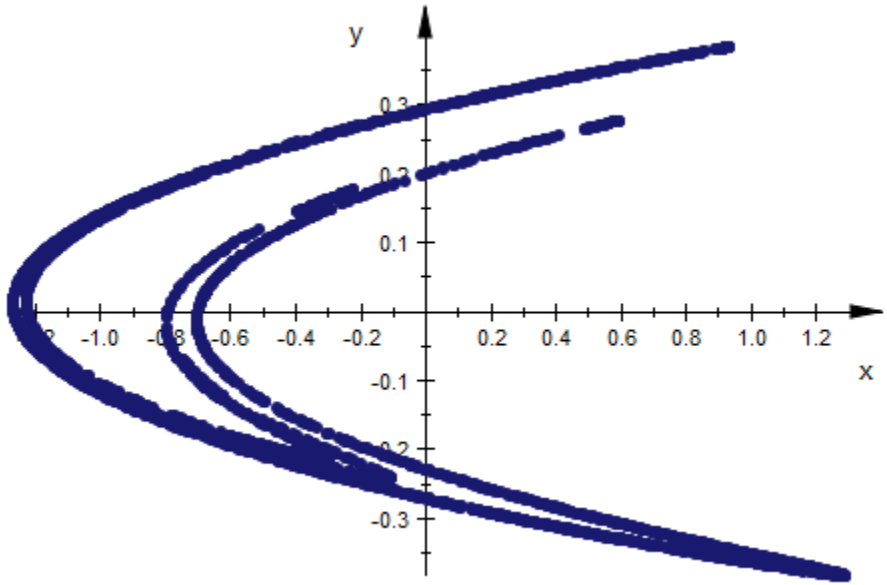
```
c1 := 1.4:
c2 := 0.3:
henon_iter := (x, y) -> [c1*x^2+y-1, c2*x]:
```

We start at (0, 0), let hundred iteration cycles pass by (to only plot the attractor) and then collect the next three thousand points:

```
[x, y] := [0, 0]:
for i from 1 to 100 do
  [x, y] := henon_iter(x, y);
end_for:
data := {}:
for i from 1 to 3000 do
  [x, y] := henon_iter(x, y);
  data := data union {[x, y]};
end_for:
```

Note that we collected the data in a set, because adding elements to a set is a fast operation, unlike changing the length of a list, and we don't care for the order in which points were reached. To plot the data, we must convert it to a list first:

```
data := coerce(data, DOM_LIST):
plot(plot::PointList2d(data))
```



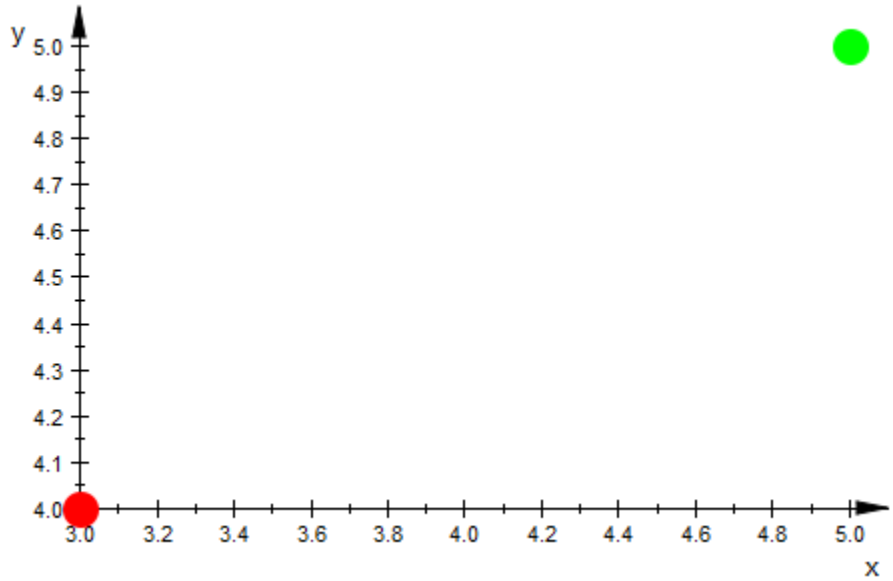
We'd like to invite you to experiment with different values of `c1` and `c2` and see how they change the resulting image.

Example 3

`plot::PointList2d` and `plot::PointList3d` allow you to specify the colors of the points. For example, the following list contains two points. When you plot this list, the first point appears in red, and the second point appears in green:

```
Coords := [[3, 4, RGB::Red], [5, 5, RGB::Green]];
plotCoords := plot::PointList2d(Coords):
plot(plotCoords,  PointSize=5)
```

```
[[3, 4, [1.0, 0.0, 0.0]], [5, 5, [0.0, 1.0, 0.0]]]
```



If you specify the color of one point, you must also specify the colors of all other points in the list:

```
Coords := [[3, 4, RGB::Red], [5, 5]];
plotCoords := plot::PointList2d(Coords)
```

```
[[3, 4, [1.0, 0.0, 0.0]], [5, 5]]
```

```
Error: The attribute 'Points2d' in the 'PointList2d' object must be a list of lists of
```

Example 4

(Feigenbaum's period doubling route to chaos)

We consider the iteration $x_{n+1} = f_p(x_n)$ where $f_p : x \to p\,x\,(1-x)$ is the "logistic map" with a parameter p. The iteration map f_p maps the interval [0, 1] to itself for $0 \le p \le 4$. For small values of p, the sequence (x_n) has a finite number of accumulation

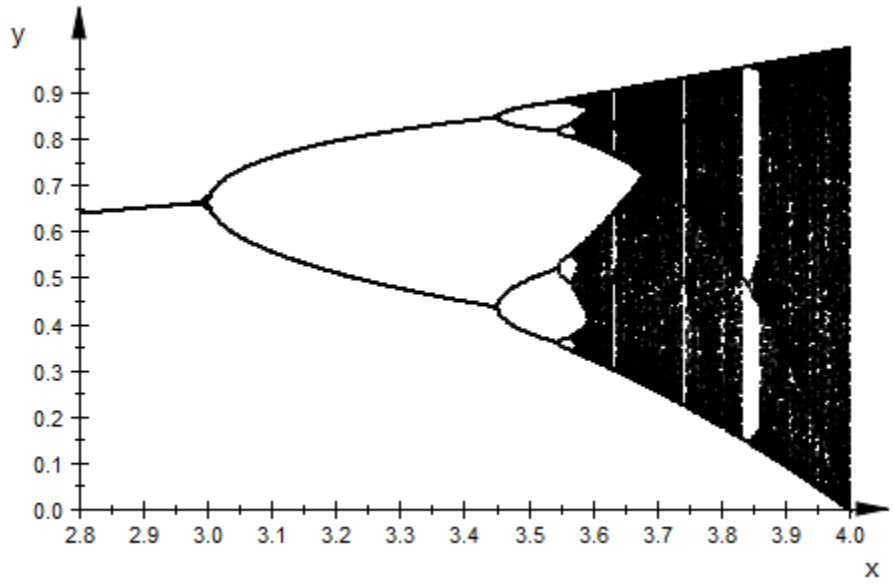
points that are visited cyclically. Increasing p, the accumulation points split into 2 separate accumulation points for certain critical values of p (“period doubling”). For $p \approx 3.569945672\ldots$, there are infinitely many accumulation points and the sequence (x_n) behaves chaotically.

We wish to visualize the accumulation points as functions of p (“Feigenbaum diagram”).

For P closely spaced values of p, we construct the sequence (x_n) starting with $x_0 = 0.5$. We ignore the first N values, expecting that the next M values cycle over the accumulation points. These points are added to a list `plotdata` that is finally fed into a `PointList2d` for plotting:

```
f:= (p, x) -> p*x*(1-x):

 P:= 500: // number of steps in p direction
N:= 200: // transitional steps before we are close to the cycle
M:= 300: // maximal number of points on the cycle

 pmin:= 2.8: // Consider p between
pmax:= 4.0: // pmin and pmax
plotdata:= [ ]:
for p in [pmin + i*(pmax - pmin)/P $ i = 0..P] do
    // First, do N iterations to drive the
    // point x towards the limit cycle
    x:= 0.5:
    for i from 1 to N do
      x:= f(p, x):
    end_for:

     // consider the next M iterates and use them as plot data:
    xSequence:= table():
    xSequence[1]:= x;
    for i from 2 to M do
        x:= f(p, x):
        if abs(x - xSequence[1]) < 10^(-5) then
           // We are back at the beginning of the cycle;
           // the points will repeat. Go to the next p.
           break;
        else
           xSequence[i]:= x;
        end_if;
    end_for:
    plotdata:= plotdata . [[p, rhs(x)] $ x in xSequence];
end_for:
```

```
plot(plot::PointList2d(plotdata,
                       PointColor = RGB::Black,
                       PointSize = 0.5*unit::mm)):
```




```
delete f, P, N, M, pmin, pmax, plotdata, x, xSequence, i;
```

Example 5

Create the following number spiral by plotting only prime numbers. This plot shows that primes cluster along particular curves called prime-generating curves.

```
plot(
     plot::PointList2d([[sqrt(n)*cos(2*PI*sqrt(n)),
                        sqrt(n)*sin(2*PI*sqrt(n))]
                $ n in [ithprime(j) $ j = 1..2345]],
                                     PointSize = 1
     ),
      Axes = None, Scaling = Constrained,
      Height = 100, Width = 100)
```

Parameters

`pts`

A list of points. A point must not be of type `plot::Point2d` or `plot::Point3d`, respectively. In 2D, each point must be a list of two real-valued expressions (the coordinates) and an optional RGB color. In 3D, each point mut be a list of three expressions (the coordinates) and an optional RGB or RGBa color. The lists specifying the points and the colors must all have the same length.

pts is equivalent to the attributes `Points2d`, `Points3d`.

M_{2d}

An array or a matrix with 2 columns. Each row provides the coordinates of one point.

M_{2d} is equivalent to the attribute `Points2d`.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Listplot` | `plot::Point2d` | `plot::Point3d` | `plot::PointList3d` | `plot::Polygon2d` | `plot::Polygon3d` | `plot::Scatterplot`

plot::PointList3d

Finite lists of 3D points

Syntax

```
plot::PointList3d(pts, <a = a_min .. a_max>, options)
plot::PointList3d(M_3d, <a = a_min .. a_max>, options)
```

Description

`plot::PointList3d` holds lists of points in 3D.

These types are containers for a (large) finite number of points. They let you avoid constructing large numbers of objects of type `plot::Point3d`, for two reasons. First, the point types have non-negligible overhead and constructing and plotting a large number of them (say, five thousand) takes more time than plotting the same number of points in a single container object. Second, and this may be even more important, having five thousand points in the object browser takes a significant amount of memory and is not as lucid as having a single point list displayed there.

The attribute `Points3d` is displayed in the inspector in the user interface only for short lists.

`plot::PointList2d, PointList3d` internally use lists for storing the points. It is therefore not advisable to add a large number of points one-by-one. See “Example 2” on page 24-686 for a better method of collecting data.

If you specify the color of one point, you must specify the colors of all other points in the list. See “Example 3” on page 24-687.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Color`	the main color	`RGB::MidnightBlue`

Attribute	Purpose	Default Value
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`Points3d`	list of 3D points	
`PointSize`	the size of points	`1.5`
`PointColor`	the color of points	`RGB::MidnightBlue`
`PointStyle`	the presentation style of points	`FilledCircles`
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`
`Title`	object title	
`TitleFont`	font of object titles	`[" sans-serif ", 11]`
`TitlePosition`	position of object titles	
`TitleAlignment`	horizontal alignment of titles w.r.t. their coordinates	`Center`

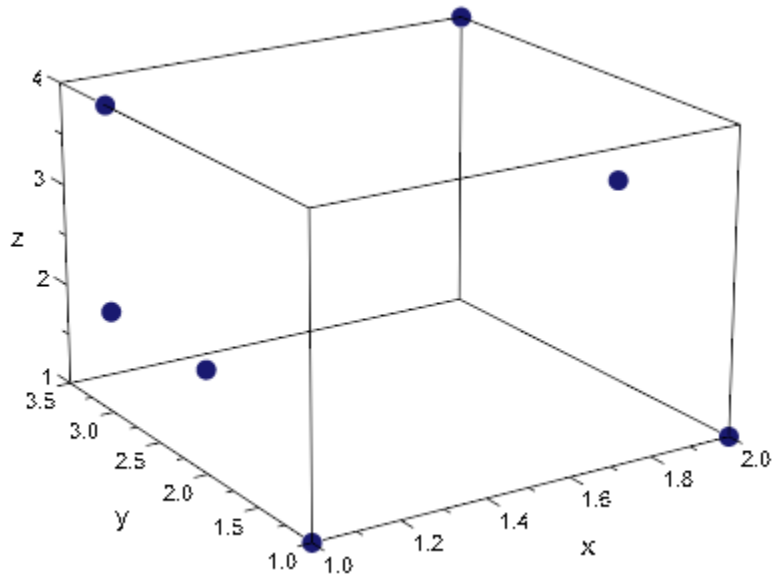
Attribute	Purpose	Default Value
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE

Examples

Example 1

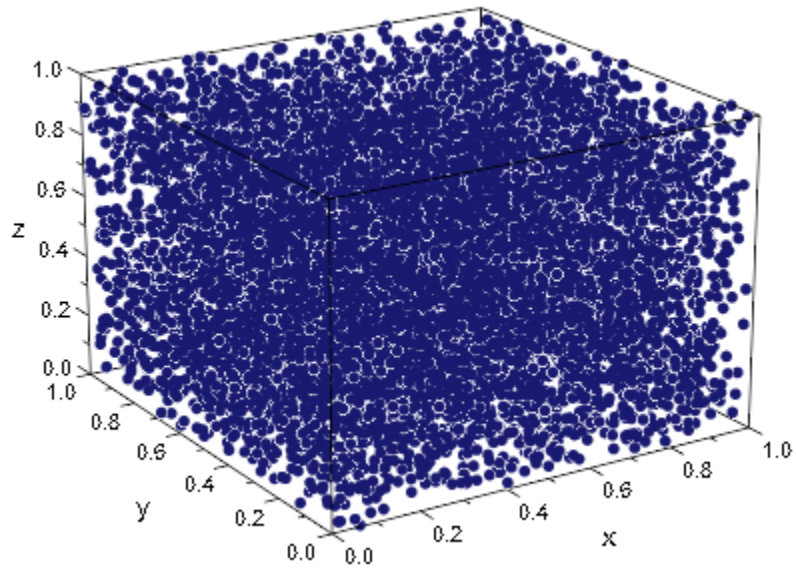
`plot::PointList3d` provides a basic form of scatter plot:

```
plot(plot::PointList3d([[1,1,1], [1,2,2], [1,3,2], [1,3,4],
                        [2,1,1], [2,2,3], [2,3.5, 4]],
                       PointSize=5))
```

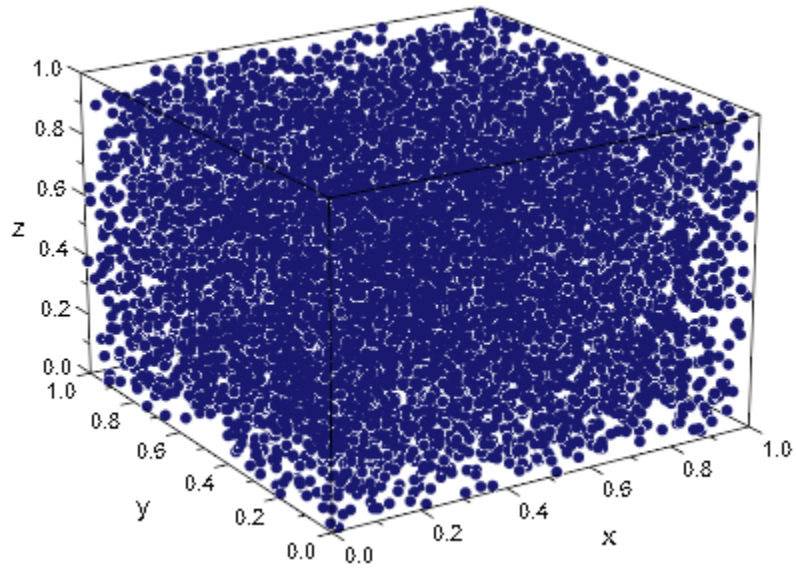


We can use this, for example, to get a visual test of random number generators:

```
r := frandom(0):
plot(plot::PointList3d([[r(), r(), r()] $ i=1..10000])):
```

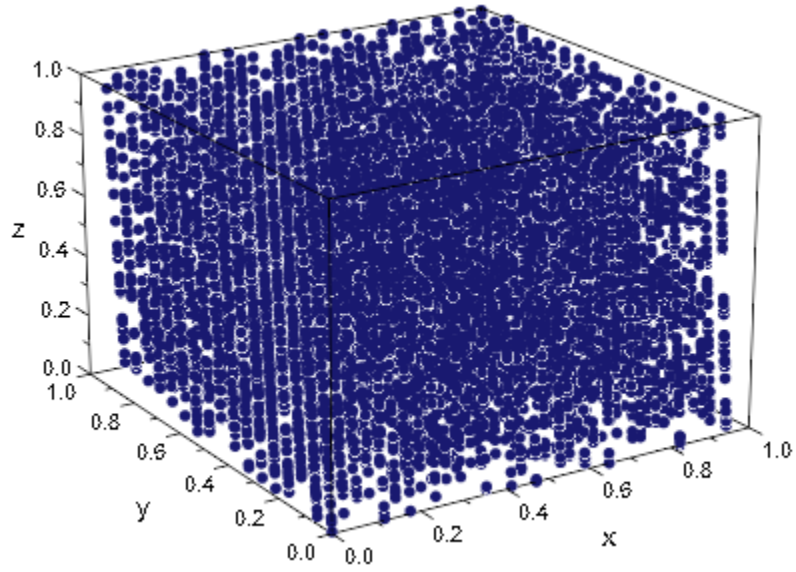



```
r := random(10^10)*1e-10:
plot(plot::PointList3d([[r(), r(), r()] $ i=1..10000])):
```



`frandom` and `random` fill the cube nicely, without noticeable patterns. The following generator, however, should probably not be used:

```
randseed := 12345:
r := proc()
     begin
       randseed := (randseed * 17 + 8) mod 10^10:
       1e-10 * randseed;
     end:
plot(plot::PointList3d([[r(), r(), r()] $ i=1..10000])):
```



Example 2

The following iteration leads to the so-called Hénon attractor (from chaos theory):

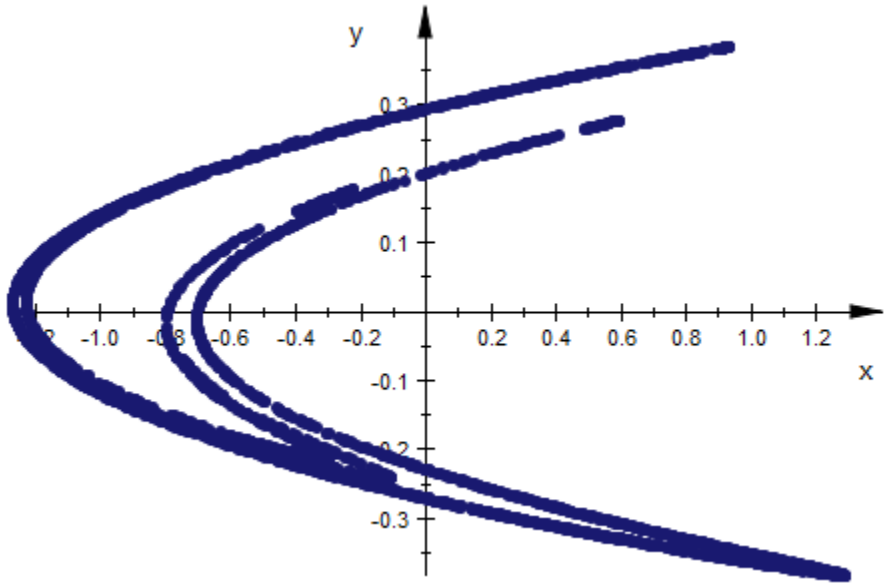
```
c1 := 1.4:
c2 := 0.3:
henon_iter := (x, y) -> [c1*x^2+y-1, c2*x]:
```

We start at (0, 0), let hundred iteration cycles pass by (to only plot the attractor) and then collect the next three thousand points:

```
[x, y] := [0, 0]:
for i from 1 to 100 do
  [x, y] := henon_iter(x, y);
end_for:
data := {}:
for i from 1 to 3000 do
  [x, y] := henon_iter(x, y);
  data := data union {[x, y]};
end_for:
```

Note that we collected the data in a set, because adding elements to a set is a fast operation, unlike changing the length of a list, and we don't care for the order in which points were reached. To plot the data, we must convert it to a list first:

```
data := coerce(data, DOM_LIST):
plot(plot::PointList2d(data))
```



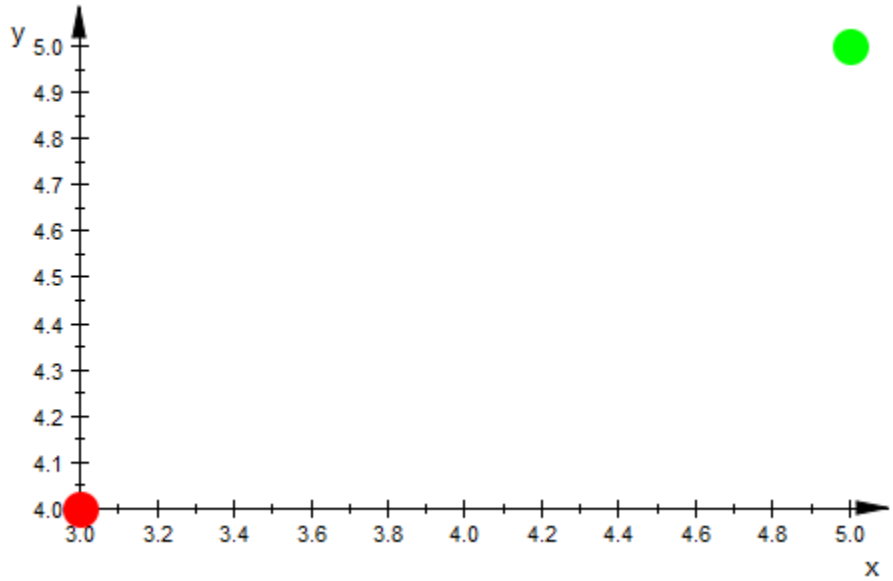
We'd like to invite you to experiment with different values of `c1` and `c2` and see how they change the resulting image.

Example 3

`plot::PointList2d` and `plot::PointList3d` allow you to specify the colors of the points. For example, the following list contains two points. When you plot this list, the first point appears in red, and the second point appears in green:

```
Coords := [[3, 4, RGB::Red], [5, 5, RGB::Green]];
plotCoords := plot::PointList2d(Coords):
plot(plotCoords,  PointSize=5)
```

```
[[3, 4, [1.0, 0.0, 0.0]], [5, 5, [0.0, 1.0, 0.0]]]
```



If you specify the color of one point, you must also specify the colors of all other points in the list:

```
Coords := [[3, 4, RGB::Red], [5, 5]];
plotCoords := plot::PointList2d(Coords)
```

```
[[3, 4, [1.0, 0.0, 0.0]], [5, 5]]
```

```
Error: The attribute 'Points2d' in the 'PointList2d' object must be a list of lists of
```

Example 4

(Feigenbaum's period doubling route to chaos)

We consider the iteration $x_{n+1} = f_p(x_n)$ where $f_p : x \to p\,x\,(1-x)$ is the "logistic map" with a parameter p. The iteration map f_p maps the interval [0, 1] to itself for $0 \le p \le 4$. For small values of p, the sequence (x_n) has a finite number of accumulation points that are visited cyclically. Increasing p, the accumulation points split into 2 separate accumulation points for certain critical values of p ("period doubling"). For

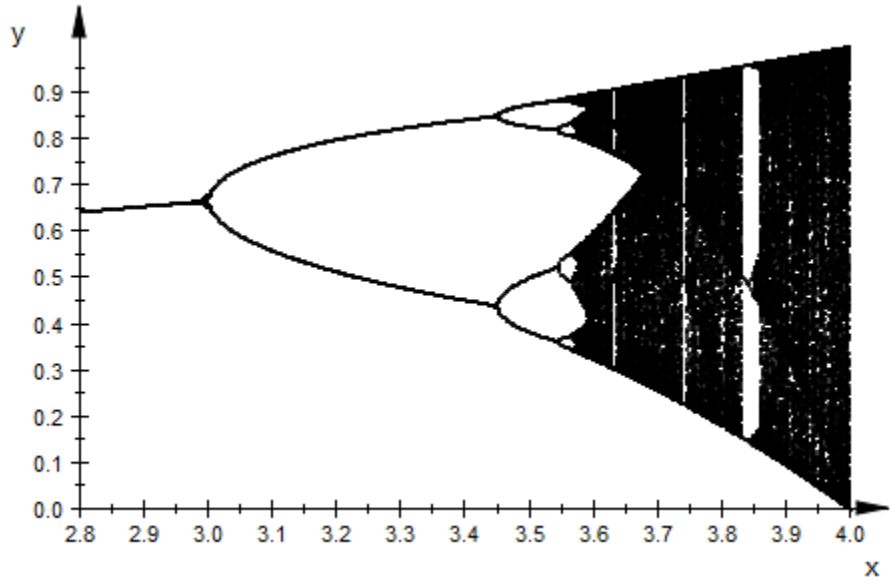
$p \approx 3.569945672\ldots$, there are infinitely many accumulation points and the sequence (x_n) behaves chaotically.

We wish to visualize the accumulation points as functions of p ("Feigenbaum diagram").

For P closely spaced values of p, we construct the sequence (x_n) starting with $x_0 = 0.5$. We ignore the first N values, expecting that the next M values cycle over the accumulation points. These points are added to a list `plotdata` that is finally fed into a `PointList2d` for plotting:

```
f:= (p, x) -> p*x*(1-x):

 P:= 500: // number of steps in p direction
N:= 200: // transitional steps before we are close to the cycle
M:= 300: // maximal number of points on the cycle

 pmin:= 2.8: // Consider p between
pmax:= 4.0: // pmin and pmax
plotdata:= [ ]:
for p in [pmin + i*(pmax - pmin)/P $ i = 0..P] do
    // First, do N iterations to drive the
    // point x towards the limit cycle
    x:= 0.5:
    for i from 1 to N do
      x:= f(p, x):
    end_for:

     // consider the next M iterates and use them as plot data:
    xSequence:= table():
    xSequence[1]:= x;
    for i from 2 to M do
        x:= f(p, x):
        if abs(x - xSequence[1]) < 10^(-5) then
           // We are back at the beginning of the cycle;
           // the points will repeat. Go to the next p.
           break;
        else
           xSequence[i]:= x;
        end_if;
    end_for:
    plotdata:= plotdata . [[p, rhs(x)] $ x in xSequence];
end_for:

 plot(plot::PointList2d(plotdata,
```

```
                    PointColor = RGB::Black,
                    PointSize = 0.5*unit::mm)):
```




```
delete f, P, N, M, pmin, pmax, plotdata, x, xSequence, i;
```

Parameters

`pts`

A list of points. A point must not be of type `plot::Point2d` or `plot::Point3d`, respectively. In 2D, each point must be a list of two real-valued expressions (the coordinates) and an optional RGB color. In 3D, each point mut be a list of three expressions (the coordinates) and an optional RGB or RGBa color. The lists specifying the points and the colors must all have the same length.

pts is equivalent to the attributes `Points2d`, `Points3d`.

M_{3d}

An array or a matrix with 3 columns. Each row provides the coordinates of one point.

M_{3d} is equivalent to the attribute `Points3d`.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Listplot` | `plot::Point2d` | `plot::Point3d` | `plot::PointList2d` | `plot::Polygon2d` | `plot::Polygon3d` | `plot::Scatterplot`

plot::Polar

Curves in 2D parameterized in polar coordinates

Syntax

```
plot::Polar([r, ϕ], u = u_min .. u_max, <a = a_min .. a_max>, options)
```

Description

`plot::Polar` creates parameterized curves in 2D, with parametrization in polar coordinates.

`plot::Polar` creates curves in one parameter, with parametrization in polar coordinates and possibly animated (see “Example 1” on page 24-695 and “Example 2” on page 24-699). The curves may contain poles, in which case automatic clipping is used by default, see “Example 4” on page 24-703.

Polar coordinates consist of a radius and an angle. The radius of a point is its distance from the origin (0, 0), while the angle is the angle between the positive “*x*”-axis (the ordinate) and the connection between the point and the origin, measured in radians and counter-clockwise.

By default, curves are sampled at equidistant values of the parameter `t`. The attribute `AdaptiveMesh` can be used to change this behavior, such that a denser sampling rate is used in areas of higher curvature. Cf. “Example 3” on page 24-700.

Curves are graphical objects that can be manipulated, see the examples and the documentation of the parameters listed below for details.

Attributes

Attribute	Purpose	Default Value
`AdaptiveMesh`	adaptive sampling	`0`
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`

Attribute	Purpose	Default Value
`Color`	the main color	`RGB::Blue`
`DiscontinuitySearch`	semi-symbolic search for discontinuities	`TRUE`
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`TRUE`
`LineColor`	color of lines	`RGB::Blue`
`LineWidth`	width of lines	`0.35`
`LineColor2`	color of lines	`RGB::DeepPink`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LinesVisible`	visibility of lines	`TRUE`
`LineColorType`	line coloring types	`Flat`
`LineColorFunction`	functional line coloring	
`LineColorDirection`	the direction of color transitions on lines	[0, 1]
`LineColorDirectionX`	x-component of the direction of color transitions on lines	`0`
`LineColorDirectionY`	y-component of the direction of color transitions on lines	`1`
`Mesh`	number of sample points	`121`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	

Attribute	Purpose	Default Value
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE
Submesh	density of submesh (additional sample points)	0
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
UMax	final value of parameter “u”	
UMesh	number of sample points for parameter “u”	121
UMin	initial value of parameter “u”	
UName	name of parameter “u”	
URange	range of parameter “u”	

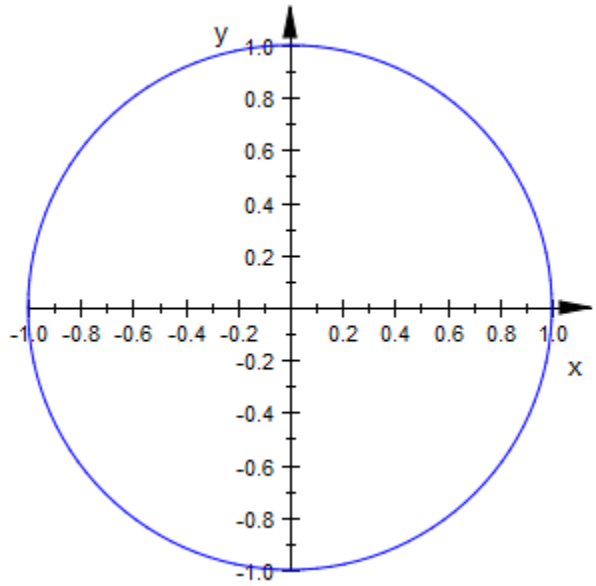
Attribute	Purpose	Default Value
USubmesh	density of additional sample points for parameter “u”	0
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XFunction	function for x values	
YFunction	function for y values	

Examples

Example 1

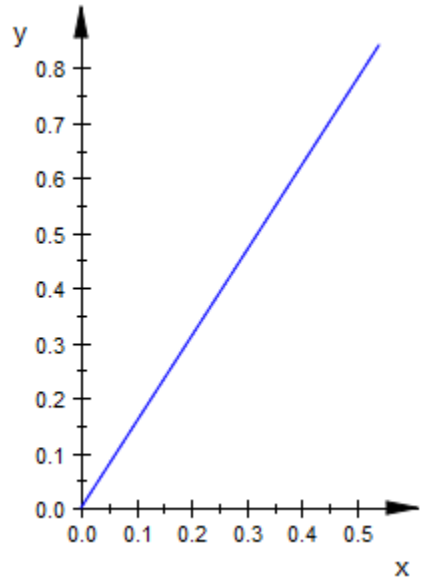
The most basic example of a curve in polar coordinates is a circle: Using a constant radius, the angle goes from 0 to 2 π:

```
plot(plot::Polar([1, u], u = 0..2*PI))
```



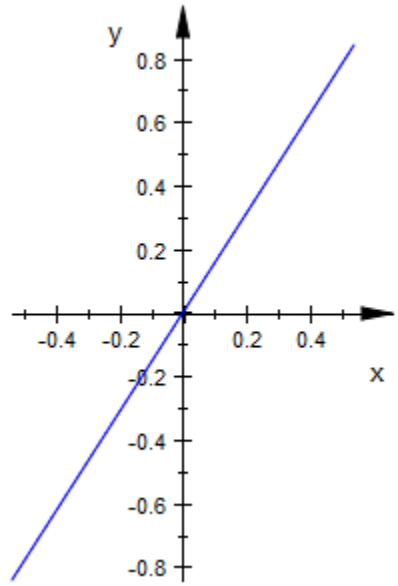
A constant angle, on the other hand, means a straight line through the origin:

```
plot(plot::Polar([r, 1], r = 0..1))
```



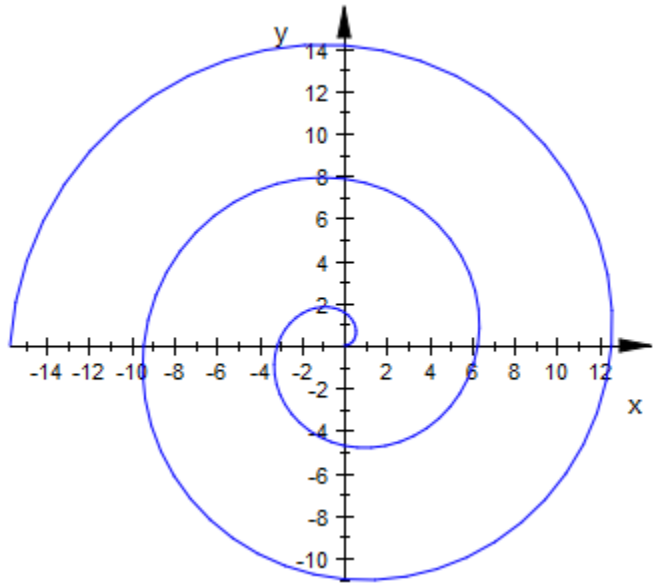
`plot::Polar` accepts negative radii:

```
plot(plot::Polar([r, 1], r = -1..1))
```



The most simple “interesting” example is probably Archimedes' spiral:

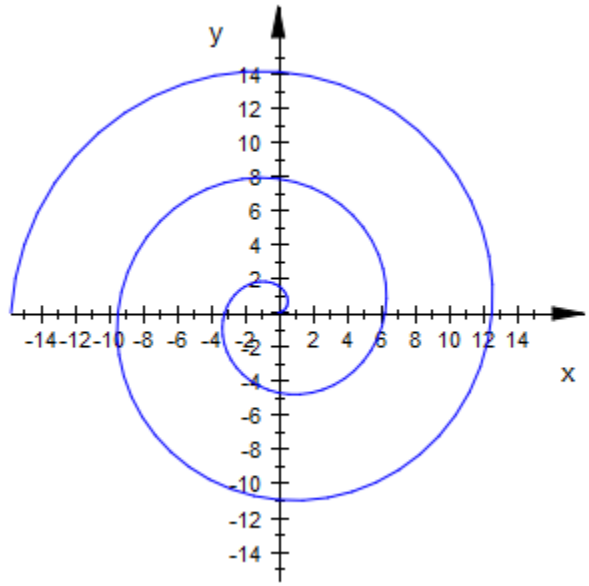
```
plot(plot::Polar([r, r], r = 0..5*PI))
```



Example 2

Polar curves can be animated just like almost anything else:

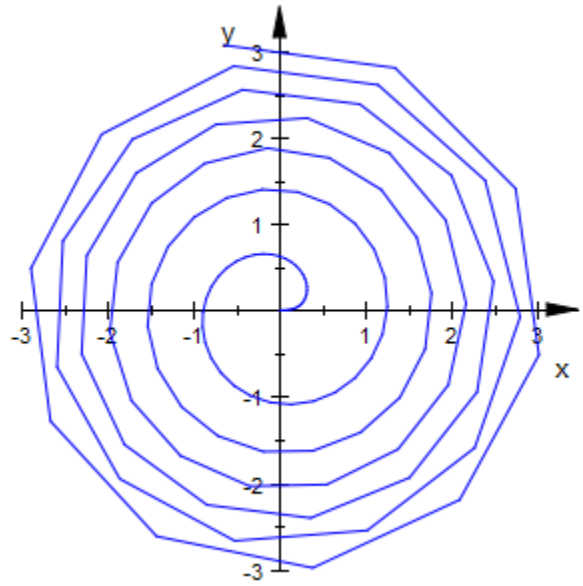
```
plot(plot::Polar([r, a*r], r = 0..5*PI, a = -1..1))
```



Example 3

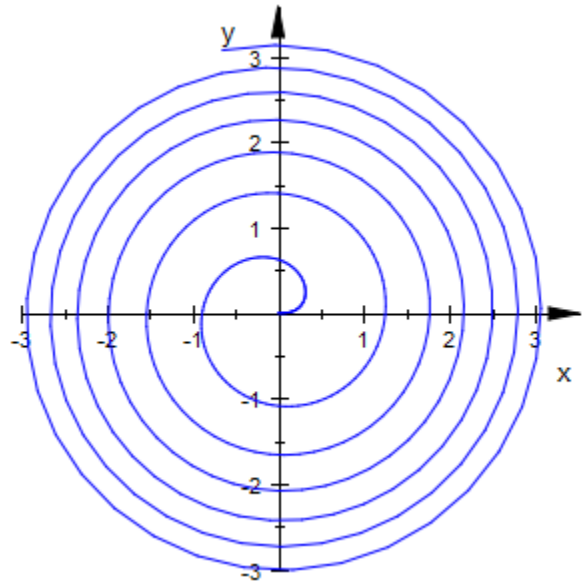
In some cases, the default of 121 evaluations on the curve is not sufficient and causes visible artifacts:

```
plot(plot::Polar([r, 4*r^2], r = 0..PI))
```



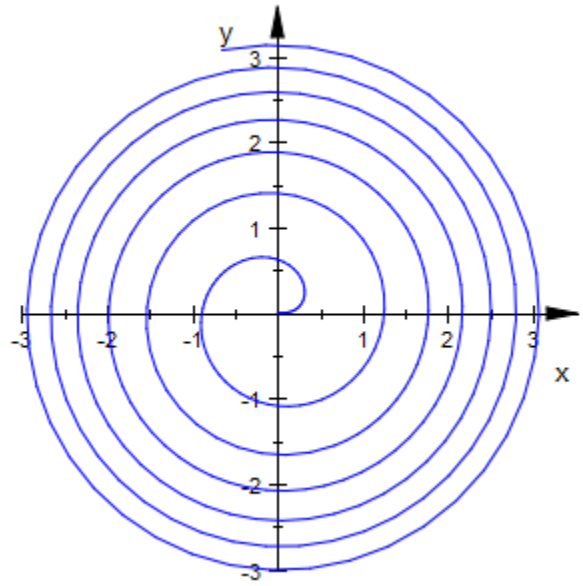
One remedy for this problem is to increase the number of evaluation points:

```
plot(plot::Polar([r, 4*r^2], r = 0..PI, Mesh = 400))
```



This method is, however, wasteful: Near the center, the initial density was perfectly sufficient, while on the outer edge still more points would be desirable. `plot::Polar` offers adaptive mesh refinement for exactly these situations. In the following example, we switch on adaptive mesh refinement with up to $2^4 = 16$ points introduced between each two consecutive points of the initial mesh:

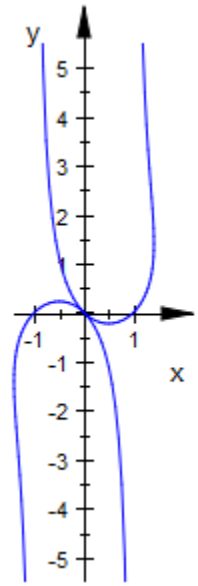
```
plot(plot::Polar([r, 4*r^2], r = 0..PI, AdaptiveMesh=4))
```



Example 4

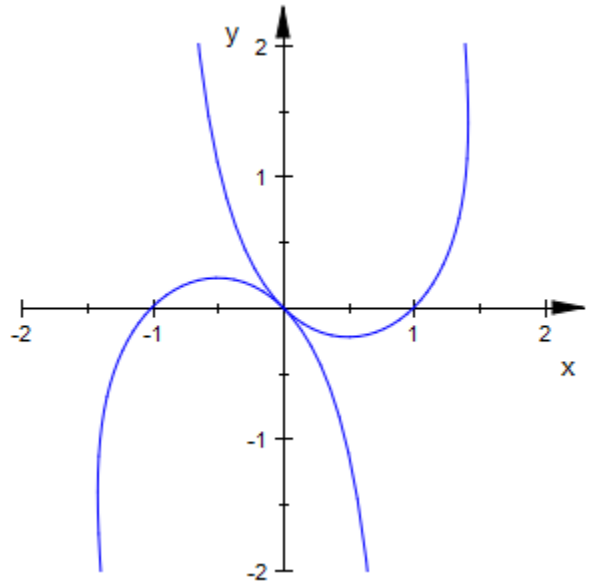
If the curve (i.e., the radius expression/function) contains poles, `plot::Polar` will use heuristics to clip the viewing box:

```
plot(plot::Polar([tan(t)+1, t], t = 0..2*PI))
```



To select a different area, use the attribute `ViewingBox`:

```
plot(plot::Polar([tan(t)+1, t], t = 0..2*PI,
                 ViewingBox = [-2..2, -2..2]))
```



Example 5

`plot::Polar` creates objects that can be manipulated interactively and/or programmatically:

```
p := plot::Polar([tan(t)+1, t], t = 0..PI)
```

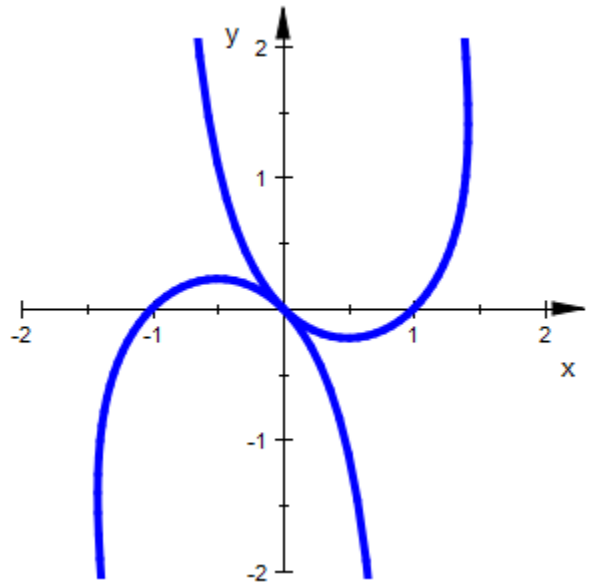
$$\text{plot::Polar}([\tan(t)+1,\ t],\ t=0..\pi)$$

```
p::UMax := 2*PI:
p
```

$$\text{plot::Polar}([\tan(t)+1,\ t],\ t=0..2\,\pi)$$

```
p::ViewingBox := [-2..2, -2..2]:
p::LineColor  := RGB::Blue:
```

```
p::LineWidth  := 1*unit::mm:
plot(p)
```



Parameters

r, ϕ

The coordinate functions: arithmetical expressions or `piecewise` objects depending on the curve parameter *u* and the animation parameter `a`. Alternatively, procedures that accept 1 input parameter *u* or 2 input parameters *u*, *a* and return a real numerical value when the input parameters are numerical.

r, ϕ are equivalent to the attributes `XFunction`, `YFunction`.

u

The curve parameter: an identifier or an indexed identifier.

u is equivalent to the attribute `UName`.

u_{min} .. u_{max}

The plot range for the parameter u: u_{min}, u_{max} must be numerical real values or expressions of the animation parameter a.

u_{min} .. u_{max} is equivalent to the attributes `URange`, `UMin`, `UMax`.

a

Animation parameter, specified as a = a_{min}..a_{max}, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions

`plot` | `plot::copy`

MuPAD Graphical Primitives

`plot::Curve2d` | `plot::Cylindrical` | `plot::Spherical`

plot::Polygon2d

2D polygons

Syntax

```
plot::Polygon2d([pt2d1, pt2d2, …], <a = amin .. amax>, options)

plot::Polygon2d(M2d, <a = amin .. amax>, options)
```

Description

`plot::Polygon2d` define polygons in 2D, by a given list of vertex points.

A polygon consists of points and edges. The edges are made up of the lines traversing from the first to the second point, the second to the third point, and so on. The last point is automatically connected with the first point if the attribute `Closed` = `TRUE` is specified.

Points and lines can be hidden via `PointsVisible` = `FALSE` and `LinesVisible` = `FALSE`. Per default the vertex points are hidden while the edges are visible.

All points as a whole can be manipulated via `PointStyle` and `PointSize`. The attribute `LineColor` sets the color for all points and all lines. Likewise all lines can be manipulated via `LineStyle` and `LineWidth`.

It is possible to vary the color of all lines and points via `LineColorType`. The default value is `Flat`. Specifying the values `Dichromatic` or `Rainbow`, a second color `LineColor2` can be set. With `Functional`, the colors are taken from a user defined `LineColorFunction`. Cf. “Example 2” on page 24-713.

The area of any closed 2D polygon can be filled by specifying `Filled` = `TRUE`. The filled area is defined by connecting the last and the first vertex. This additional edge itself, however, is only displayed if `Closed` = `TRUE` is set.

A fill color and a fill pattern can be chosen by `FillColor` and `FillPattern`.

In case of a self-intersecting polygon, a `FillStyle` can be selected. Cf. “Example 3” on page 24-715.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`
`Closed`	open or closed polygons	`FALSE`
`Color`	the main color	`RGB::Blue`
`Filled`	filled or transparent areas and surfaces	`FALSE`
`FillColor`	color of areas and surfaces	`RGB::Red`
`FillStyle`	definition of inside/outside	`EvenOdd`
`FillPattern`	type of area filling	`DiagonalLines`
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`LineColor`	color of lines	`RGB::Blue`
`LineWidth`	width of lines	`0.35`
`LineColor2`	color of lines	`RGB::DeepPink`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LinesVisible`	visibility of lines	`TRUE`
`LineColorType`	line coloring types	`Flat`
`LineColorFunction`	functional line coloring	

Attribute	Purpose	Default Value
LineColorDirection	the direction of color transitions on lines	[0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	1
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
Points2d	list of 2D points	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center

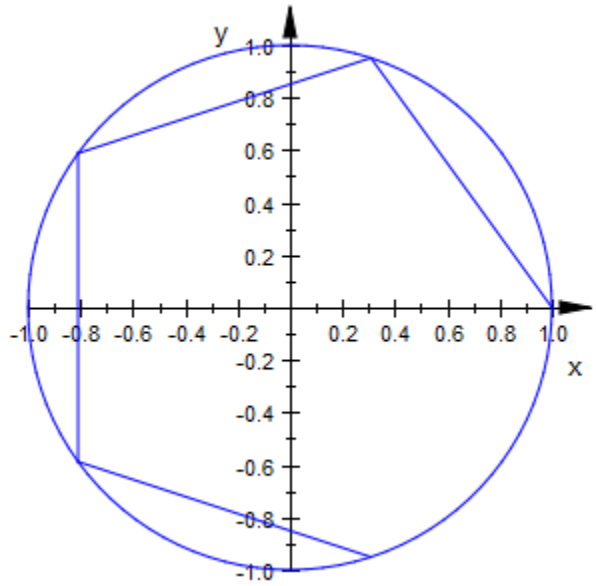
Attribute	Purpose	Default Value
`TitlePositionX`	position of object titles, x component	
`TitlePositionY`	position of object titles, y component	
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

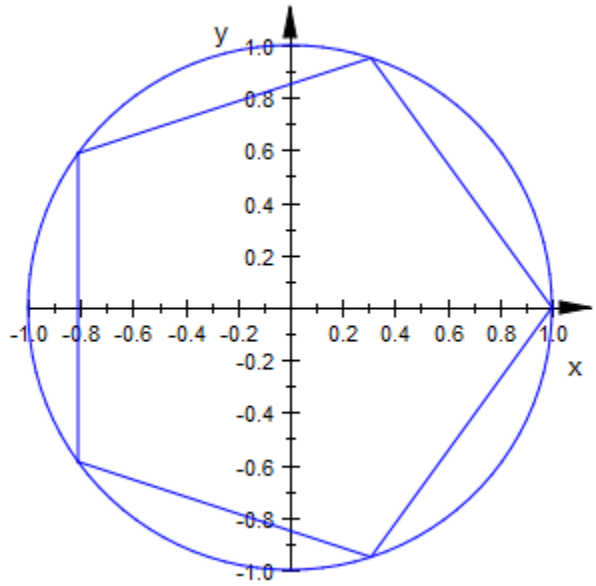
We create a polygon with vertices located at the 5 complex 5^{th} roots of 1. The polygon consists of the 4 lines joining the 5 points in the order given:

```
p := plot::Polygon2d(
       [[cos(2*PI*k/5), sin(2*PI*k/5)] $ k = 0..4]):
plot(p, plot::Circle2d(1, [0, 0])):
```



In order to include the line connecting the last with the first point, pass the attribute `Closed` to the polygon:

```
p::Closed := TRUE:
plot(p, plot::Circle2d(1, [0, 0])):
```

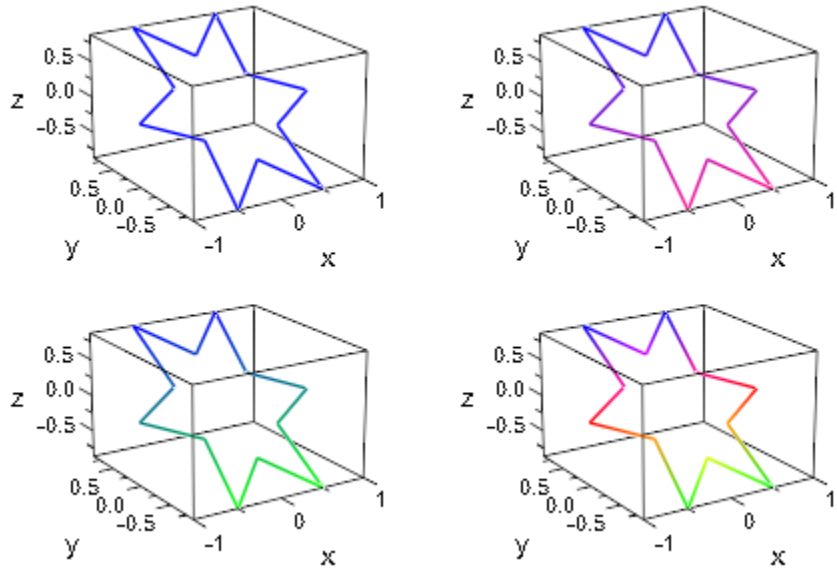



```
delete p
```

Example 2

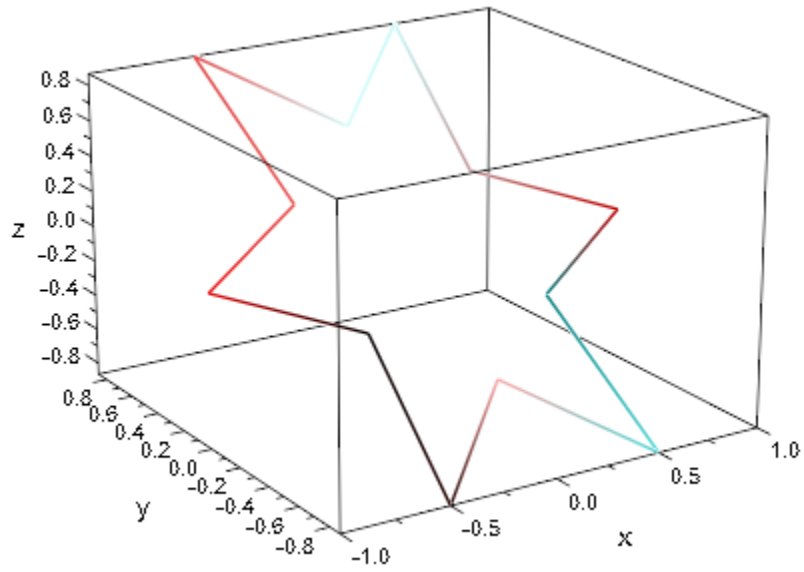
We plot a closed star-shaped 3D polygon with various color attributes:

```
p := plot::Polygon3d(
       [([cos(PI*k/3), sin(PI*k/3), sin(PI*k/3)],
         [cos(PI*k/3 + PI/6)/2,
          sin(PI*k/3 + PI/6)/2,
          sin(PI*k/3 + PI/6)/2]) $k = 1..6
       ], Closed = TRUE):
S1 := plot::Scene3d(p, LineColorType = Flat):
S2 := plot::Scene3d(p, LineColorType = Dichromatic):
S3 := plot::Scene3d(p, LineColorType = Dichromatic,
                    LineColor = RGB::Blue,
                    LineColor2 = RGB::Green):
S4 := plot::Scene3d(p, LineColorType = Rainbow,
                    LineColor = RGB::Blue,
                    LineColor2 = RGB::Green):
plot(S1, S2, S3, S4)
```



We plot the same polygon while animating its line color using a color function. The result is a dazzling star:

```
p := plot::Polygon3d(
       [([cos(PI*k/3), sin(PI*k/3), sin(PI*k/3)],
         [cos(PI*k/3 + PI/6)/2,
          sin(PI*k/3 + PI/6)/2,
          sin(PI*k/3 + PI/6)/2]) $k = 1..6
       ], Closed = TRUE,
       LineColorFunction =
           proc(x, y, z, i, a) begin
             [sin(x + a*i)^2, sin(y + a*i)^2, sin(z + a*i)^2]:
           end_proc,
       a = 0..10):
plot(p)
```

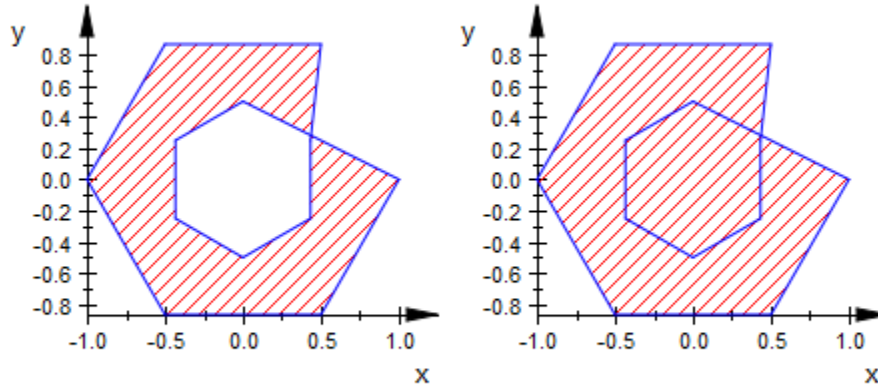



```
delete p, S1, S2, S3, S4
```

Example 3

We plot a closed 2D polygon and fill the area inside. In fact, there are two possible interpretations of what "inside" really means. In the first plot, the complement of the unbound component of the complement of the polygon is filled. In the second plot only that area is filled that contains points with non-zero winding number with respect to the polygon. Cf. `FillStyle` for a detailed discussion.

```
p := plot::Polygon2d(
      [[cos(PI*k/3), sin(PI*k/3)] $k = 1..6,
       [cos(PI*k/3 + PI/6)/2, sin(PI*k/3 + PI/6)/2] $k = 1..6
      ], Closed = TRUE):
S1 := plot::Scene2d(p, Filled = TRUE):
S2 := plot::Scene2d(p, Filled = TRUE, FillStyle = Winding):
plot(S1, S2, Layout = Horizontal, Axes = Frame,
     Scaling = Constrained)
```




```
delete p, S1, S2:
```

Parameters

$pt2d_1$, $pt2d_2$, …

The 2D vertices. These must not be of type `plot::Point2d`, but lists of two numerical real values or arithmetical expressions of the animation parameter `a` (the coordinates).

$pt2d_1$, $pt2d_2$, … is equivalent to the attribute `Points2d`.

M_{2d}

An array or a matrix with 2 columns. Each row provides the coordinates of one point.

M_{2d} is equivalent to the attribute `Points2d`.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Line2d` | `plot::Line3d` | `plot::Listplot` | `plot::Polygon3d`

plot::Polygon3d

3D polygons

Syntax

```
plot::Polygon3d([pt3d1, pt3d2, …], <a = amin .. amax>, options)

plot::Polygon3d(M3d, <a = amin .. amax>, options)
```

Description

`plot::Polygon3d` define polygons in 3D, by a given list of vertex points.

A polygon consists of points and edges. The edges are made up of the lines traversing from the first to the second point, the second to the third point, and so on. The last point is automatically connected with the first point if the attribute `Closed` = `TRUE` is specified.

Points and lines can be hidden via `PointsVisible` = `FALSE` and `LinesVisible` = `FALSE`. Per default the vertex points are hidden while the edges are visible.

All points as a whole can be manipulated via `PointStyle` and `PointSize`. The attribute `LineColor` sets the color for all points and all lines. Likewise all lines can be manipulated via `LineStyle` and `LineWidth`.

It is possible to vary the color of all lines and points via `LineColorType`. The default value is `Flat`. Specifying the values `Dichromatic` or `Rainbow`, a second color `LineColor2` can be set. With `Functional`, the colors are taken from a user defined `LineColorFunction`. Cf. “Example 2” on page 24-723.

A 3D polygon can only be filled if it is defined by 3 vertices (a triangle). The attribute `Filled` = `TRUE` is ignored for other 3D polygons.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Closed`	open or closed polygons	`FALSE`
`Color`	the main color	`RGB::Blue`
`Filled`	filled or transparent areas and surfaces	`FALSE`
`FillColor`	color of areas and surfaces	`RGB::LightBlue`
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`LineColor`	color of lines	`RGB::Blue`
`LineWidth`	width of lines	`0.35`
`LineColor2`	color of lines	`RGB::DeepPink`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LinesVisible`	visibility of lines	`TRUE`
`LineColorType`	line coloring types	`Flat`
`LineColorFunction`	functional line coloring	
`LineColorDirection`	the direction of color transitions on lines	[0, 0, 1]
`LineColorDirectionX`	x-component of the direction of color transitions on lines	`0`
`LineColorDirectionY`	y-component of the direction of color transitions on lines	`0`

Attribute	Purpose	Default Value
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
Points3d	list of 3D points	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	

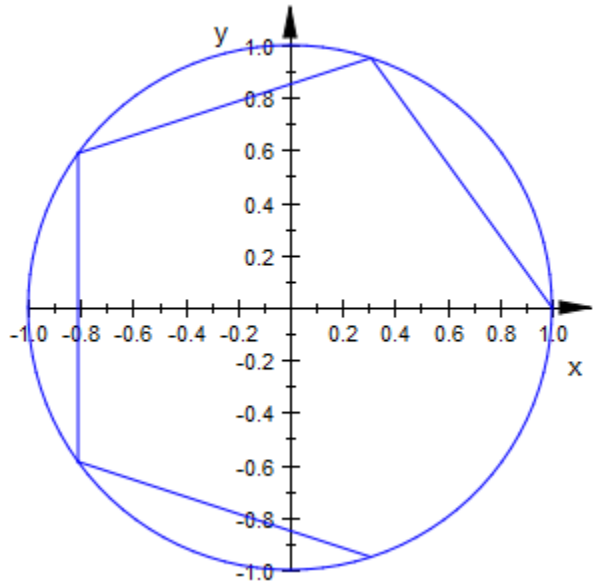
Attribute	Purpose	Default Value
`TitlePositionZ`	position of object titles, z component	
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

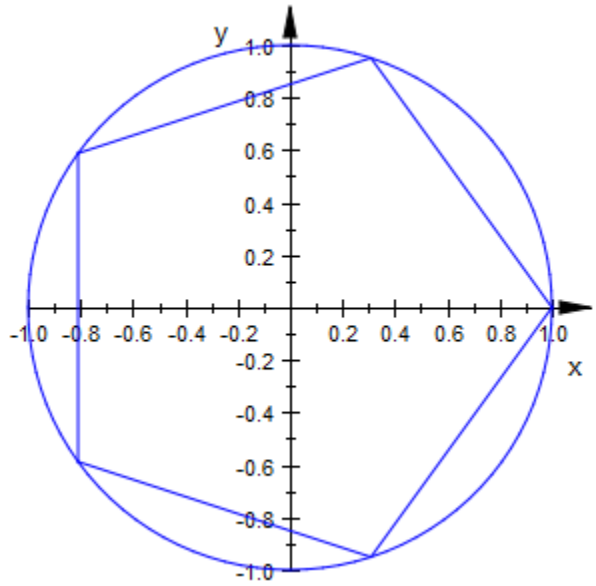
We create a polygon with vertices located at the 5 complex 5^{th} roots of 1. The polygon consists of the 4 lines joining the 5 points in the order given:

```
p := plot::Polygon2d(
       [[cos(2*PI*k/5), sin(2*PI*k/5)] $ k = 0..4]):
plot(p, plot::Circle2d(1, [0, 0])):
```



In order to include the line connecting the last with the first point, pass the attribute `Closed` to the polygon:

```
p::Closed := TRUE:
plot(p, plot::Circle2d(1, [0, 0])):
```

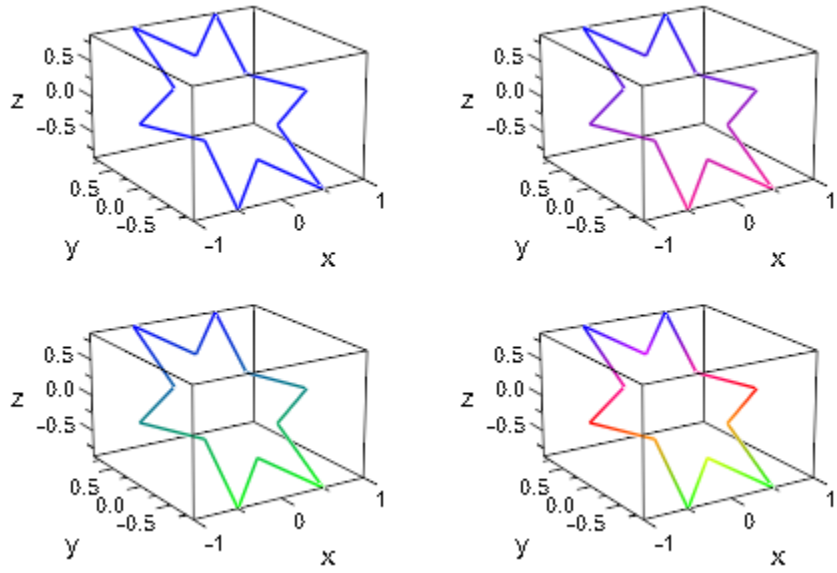



```
delete p
```

Example 2

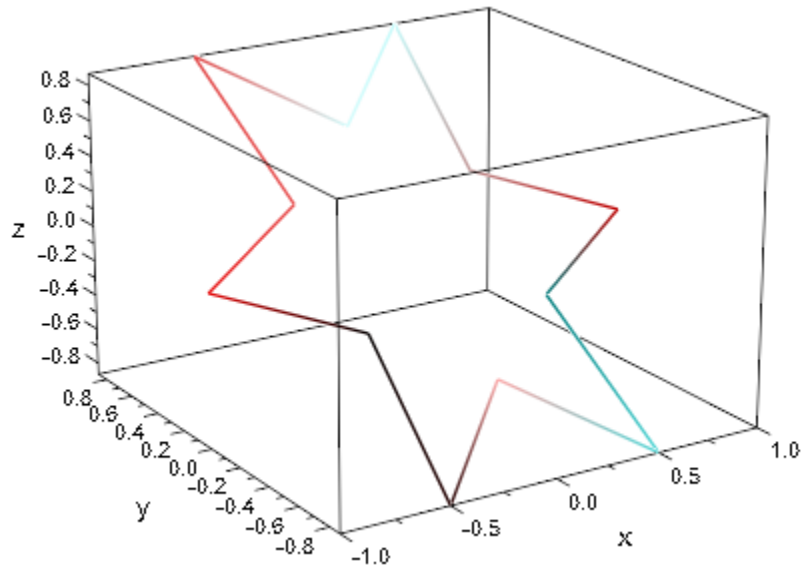
We plot a closed star-shaped 3D polygon with various color attributes:

```
p := plot::Polygon3d(
       [([cos(PI*k/3), sin(PI*k/3), sin(PI*k/3)],
         [cos(PI*k/3 + PI/6)/2,
          sin(PI*k/3 + PI/6)/2,
          sin(PI*k/3 + PI/6)/2]) $k = 1..6
       ], Closed = TRUE):
S1 := plot::Scene3d(p, LineColorType = Flat):
S2 := plot::Scene3d(p, LineColorType = Dichromatic):
S3 := plot::Scene3d(p, LineColorType = Dichromatic,
                    LineColor = RGB::Blue,
                    LineColor2 = RGB::Green):
S4 := plot::Scene3d(p, LineColorType = Rainbow,
                    LineColor = RGB::Blue,
                    LineColor2 = RGB::Green):
plot(S1, S2, S3, S4)
```



We plot the same polygon while animating its line color using a color function. The result is a dazzling star:

```
p := plot::Polygon3d(
       [([cos(PI*k/3), sin(PI*k/3), sin(PI*k/3)],
         [cos(PI*k/3 + PI/6)/2,
          sin(PI*k/3 + PI/6)/2,
          sin(PI*k/3 + PI/6)/2]) $k = 1..6
       ], Closed = TRUE,
       LineColorFunction =
           proc(x, y, z, i, a) begin
             [sin(x + a*i)^2, sin(y + a*i)^2, sin(z + a*i)^2]:
           end_proc,
       a = 0..10):
plot(p)
```

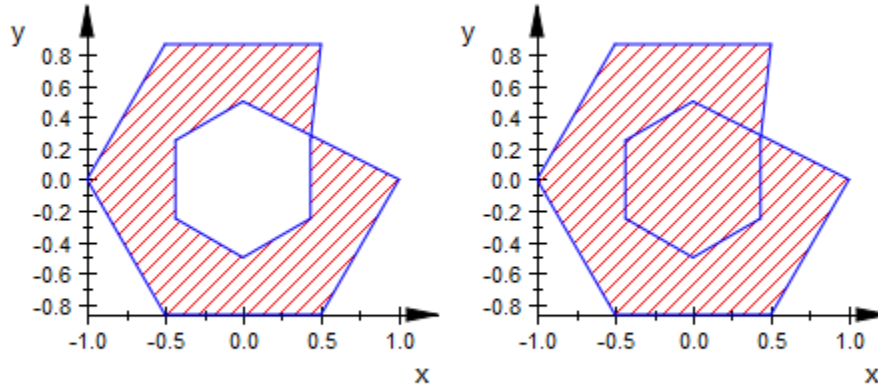



```
delete p, S1, S2, S3, S4
```

Example 3

We plot a closed 2D polygon and fill the area inside. In fact, there are two possible interpretations of what "inside" really means. In the first plot, the complement of the unbound component of the complement of the polygon is filled. In the second plot only that area is filled that contains points with non-zero winding number with respect to the polygon. Cf. `FillStyle` for a detailed discussion.

```
p := plot::Polygon2d(
      [[cos(PI*k/3), sin(PI*k/3)] $k = 1..6,
       [cos(PI*k/3 + PI/6)/2, sin(PI*k/3 + PI/6)/2] $k = 1..6
      ], Closed = TRUE):
S1 := plot::Scene2d(p, Filled = TRUE):
S2 := plot::Scene2d(p, Filled = TRUE, FillStyle = Winding):
plot(S1, S2, Layout = Horizontal, Axes = Frame,
     Scaling = Constrained)
```




```
delete p, S1, S2:
```

Parameters

$pt3d_1$, $pt3d_2$, …

The 3D vertices. These must not be of type `plot::Point3d`, but lists of three numerical real values or arithmetical expressions of the animation parameter `a` (the coordinates).

$pt3d_1$, $pt3d_2$, … is equivalent to the attribute `Points3d`.

M_{3d}

An array or a matrix with 3 columns. Each row provides the coordinates of one point.

M_{3d} is equivalent to the attribute `Points3d`.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Line2d` | `plot::Line3d` | `plot::Listplot` | `plot::Polygon2d`

plot::Prism

Prisms

Syntax

`plot::Prism(r, [x1, y1, z1], [x2, y2, z2], <a = amin .. amax>, options)`

Description

`plot::Prism(r, [x1, y1, z1] , [x2, y2, z2] )` creates a prism with a regular base plane with a circumcircle of radius *r* and an axis from the point `[x1, y1, z1]` to the point `[x2, y2, z2]`.

The base center and top center of the prism can also be passed as vectors.

Note that only prisms with a regular base can be created with `plot::Prism`. For other bases, use a `plot::SurfaceSet` primitive.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Angle`	rotation angle	`0`
`Base`	base center of cones, cylinders, pyramids and prisms	`[0, 0, 0]`
`BaseX`	x-coordinate of top center of cones, cylinders, pyramids and prisms	`0`
`BaseY`	y-coordinate of top center of cones, cylinders, pyramids and prisms	`0`

Attribute	Purpose	Default Value
BaseZ	z-coordinate of top center of cones, cylinders, pyramids and prisms	0
Color	the main color	RGB::Red
Edges	Number of Edges	3
Filled	filled or transparent areas and surfaces	TRUE
FillColor	color of areas and surfaces	RGB::Red
FillColor2	second color of areas and surfaces for color blends	RGB::CornflowerBlue
FillColorType	surface filling types	Dichromatic
FillColorFunction	functional area/surface coloring	
FillColorDirection	the direction of color transitions on surfaces	[0, 0, 1]
FillColorDirectionX	x-component of the direction of color transitions on surfaces	0
FillColorDirectionY	y-component of the direction of color transitions on surfaces	0
FillColorDirectionZ	z-component of the direction of color transitions on surfaces	1
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Black.[0.25]

Attribute	Purpose	Default Value
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	0
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Name	the name of a plot object (for browser and legend)	
Normal	normal vector of circles and discs, etc. in 3D	[0, 0, 0]
NormalX	normal vector of circles and discs, etc. in 3D, x-component	0
NormalY	normal vector of circles and discs, etc. in 3D, y-component	0
NormalZ	normal vector of circles and discs, etc. in 3D, z-component	0
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	

Attribute	Purpose	Default Value
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE
Radius	radius of circles, spheres etc.	1
Shading	smooth color blend of surfaces	Smooth
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Top	top center of cones, cylinders, pyramids and prisms	[0, 0, 1]

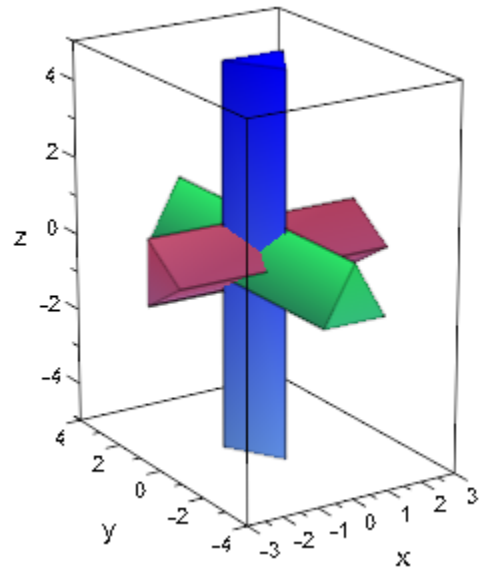
Attribute	Purpose	Default Value
TopX	base and top center of cones, cylinders, pyramids and prisms	0
TopY	base and top center of cones, cylinders, pyramids and prisms	0
TopZ	base and top center of cones, cylinders, pyramids and prisms	1
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE

Examples

Example 1

We plot three regular prisms with axes given by the coordinate axes:

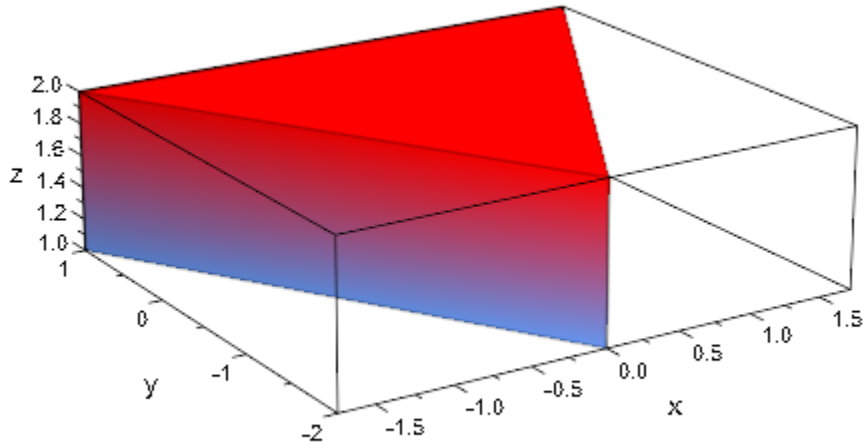
```
plot(plot::Prism(1, [-3, 0, 0], [3, 0, 0], Color = RGB::Red),
     plot::Prism(1, [0, -4, 0], [0, 4, 0], Color = RGB::Green),
     plot::Prism(1, [0, 0, -5], [0, 0, 5], Color = RGB::Blue)):
```



Example 2

All parameters of a prism can be animated:

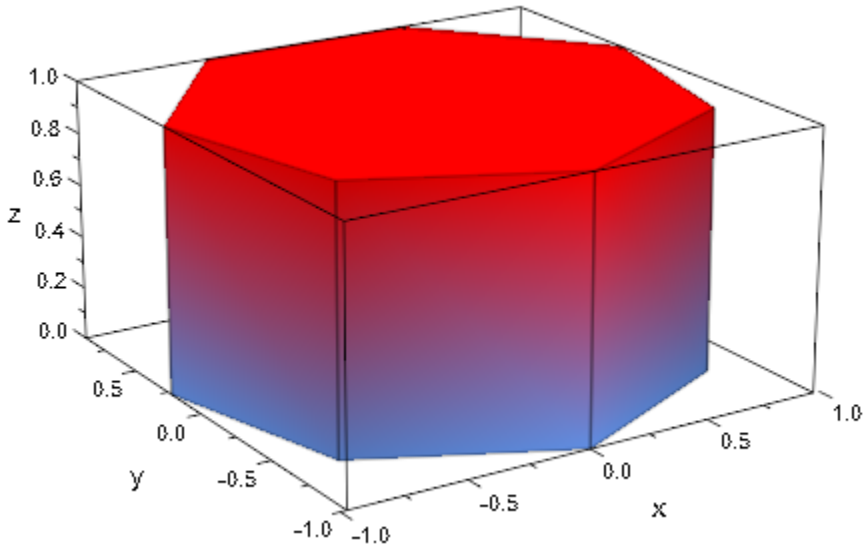
```
plot(plot::Prism(a, [0, 0, a], [0, 0, 3-a], a = 1..2)):
```



Example 3

The number of edges of the regular base plane of the prism are determined with the attribute `Edges`:

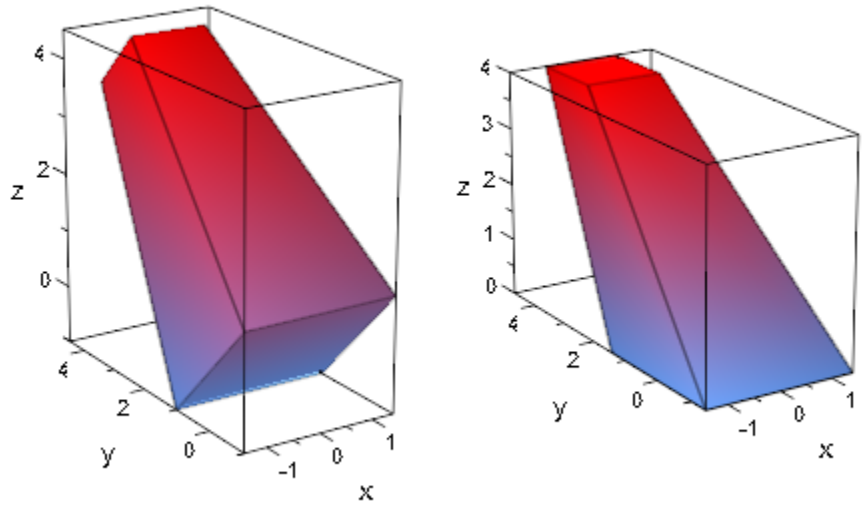
```
plot(plot::Prism(1, [0, 0, 0], [0, 0, 1], Edges = 7)):
```



Example 4

To create a crooked regular prism, the normal vector of its base plane is specified with the attribute `Normal`. If this attribute is set to [0, 0, 0], the axis between `Base` and `Top` ist used as normal vector:

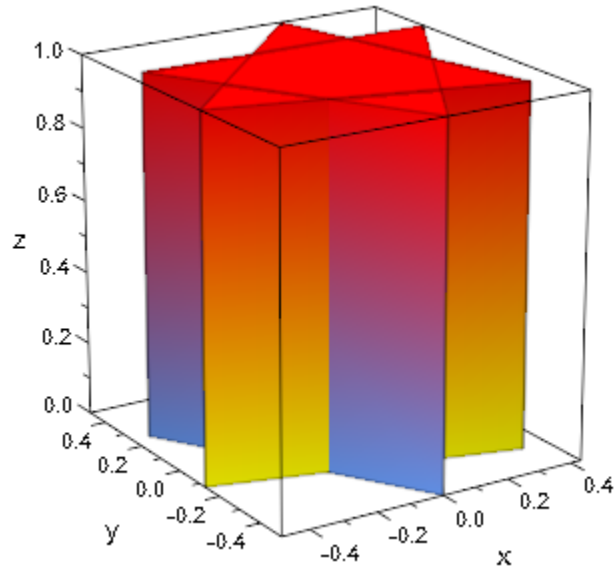
```
plot(plot::Scene3d(plot::Pyramid(2,[0,0,0],1,[0,4,4], Normal=[0,0,0])),
     plot::Scene3d(plot::Pyramid(2,[0,0,0],1,[0,4,4], Normal=[0,0,1]))):
```



Example 5

Additionally, the orientation of the edges of the base can be changed with the rotation angle `Angle`:

```
plot(plot::Prism(1/2, Angle=0),
     plot::Prism(1/2, Angle=PI/4, FillColor2=RGB::Yellow))
```



Parameters

r

The radius of the circumcircle of the regular base plane: a real numerical value or an arithmetical expression of the animation parameter `a`.

r is equivalent to the attribute `Radius`.

x_1, y_1, z_1

Components of the base center: real numerical values or expressions of the animation parameter `a`.

x_1, y_1, z_1 are equivalent to the attributes `BaseX`, `BaseY`, `BaseZ`.

x_2, y_2, z_2

Components of the top center: real numerical values or expressions of the animation parameter `a`.

x_2, y_2, z_2 are equivalent to the attributes `TopX`, `TopY`, `TopZ`.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Cone` | `plot::Cylinder` | `plot::Pyramid`

plot::Pyramid

Pyramids and frustums of pyramids

Syntax

```
plot::Pyramid(br, [bx, by, bz], <tr>, [tx, ty, tz], <a = amin .. amax>, options)
```

Description

`plot::Pyramid(br, [ bx, by, bz], [ tx, ty, tz])` creates a pyramid stretching from the regular base plane with a circumcircle of radius *br* and center `[bx, by, bz]` to the top `[tx, ty, tz]`.

`plot::Pyramid(br, [ bx, by, bz], tr, [ tx, ty, tz])` creates a frustum of pyramid from the base with center `[bx, by, bz]` to the top with center `[tx, ty, tz]`. The radius of the circumcircle of the regular base is `br`. The radius of the circumcircle of the regular top is `tr`.

The optional "top radius" `tr` for creating a frustum may also be specified as the attribute `TopRadius = tr`.

Note that only pyramids with a regular base can be created with `plot::Pyramid`. For other bases, use a `plot::SurfaceSet` primitive.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Angle`	rotation angle	`0`
`Base`	base center of cones, cylinders, pyramids and prisms	`[0, 0, 0]`

Attribute	Purpose	Default Value
`BaseX`	x-coordinate of top center of cones, cylinders, pyramids and prisms	`0`
`BaseY`	y-coordinate of top center of cones, cylinders, pyramids and prisms	`0`
`BaseZ`	z-coordinate of top center of cones, cylinders, pyramids and prisms	`0`
`BaseRadius`	base radius of cones/conical frustums and pyramids/ frustums of pyramids	`1`
`Color`	the main color	`RGB::Red`
`Edges`	Number of Edges	`4`
`Filled`	filled or transparent areas and surfaces	`TRUE`
`FillColor`	color of areas and surfaces	`RGB::Red`
`FillColor2`	second color of areas and surfaces for color blends	`RGB::CornflowerBlue`
`FillColorType`	surface filling types	`Dichromatic`
`FillColorFunction`	functional area/surface coloring	
`FillColorDirection`	the direction of color transitions on surfaces	`[0, 0, 1]`
`FillColorDirectionX`	x-component of the direction of color transitions on surfaces	`0`
`FillColorDirectionY`	y-component of the direction of color transitions on surfaces	`0`
`FillColorDirectionZ`	z-component of the direction of color transitions on surfaces	`1`

Attribute	Purpose	Default Value
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Black.[0.25]
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	0
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Name	the name of a plot object (for browser and legend)	
Normal	normal vector of circles and discs, etc. in 3D	[0, 0, 0]
NormalX	normal vector of circles and discs, etc. in 3D, x-component	0

Attribute	Purpose	Default Value
NormalY	normal vector of circles and discs, etc. in 3D, y-component	0
NormalZ	normal vector of circles and discs, etc. in 3D, z-component	0
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE
Shading	smooth color blend of surfaces	Smooth
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	

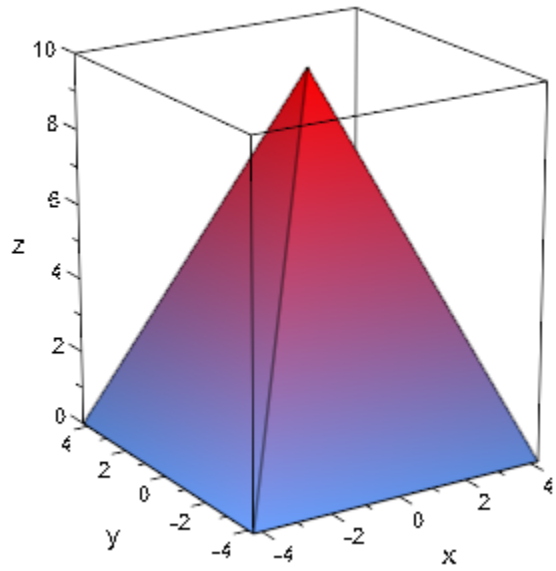
Attribute	Purpose	Default Value
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Top	top center of cones, cylinders, pyramids and prisms	[0, 0, 1]
TopX	base and top center of cones, cylinders, pyramids and prisms	0
TopY	base and top center of cones, cylinders, pyramids and prisms	0
TopZ	base and top center of cones, cylinders, pyramids and prisms	1
TopRadius	top radius of cones/conical frustums and pyramids/ frustums of pyramids	0
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE

Examples

Example 1

We draw a pyramid with base radius 6:

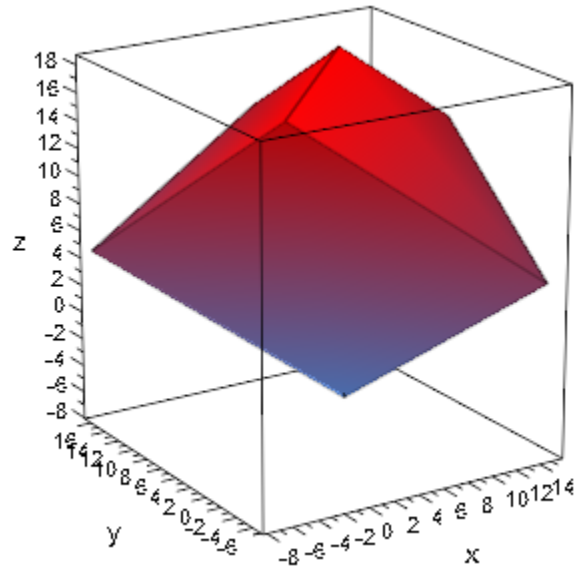
```
plot(plot::Pyramid(6, [0, 0, 0], [0, 0, 10])):
```



Example 2

We create a frustum of pyramid by specifying a non-zero top radius:

```
br := 16: base := [3, 4, 5]:
tr:= 7: top := [11, 12, 13]:
plot(plot::Pyramid(br, base, tr, top)):
```

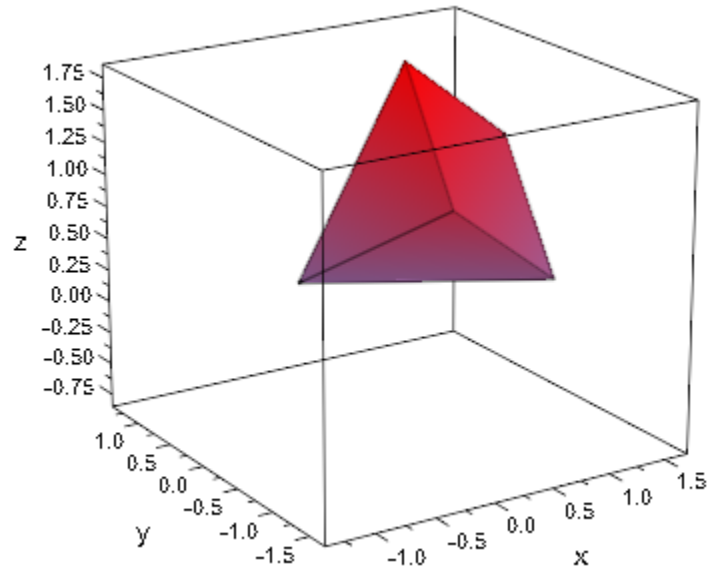



```
delete br, base, tr, top, n:
```

Example 3

Bottom and top radii and centers can be animated:

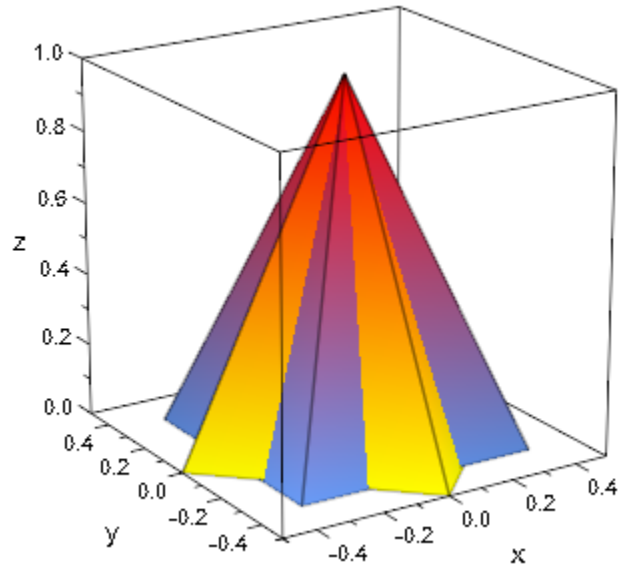
```
plot(plot::Pyramid(sin(a)^2, [sin(2*a), cos(2*a), 0],
                cos(a)^2, [cos(2*a), sin(2*a), 1],
                a = 0..PI)):
```



Example 4

Additionally, the orientation of the edges of the base can be changed with the rotation angle `Angle`:

```
plot(plot::Pyramid(1/2, Angle=0),
     plot::Pyramid(1/2, Angle=PI/4, FillColor2=RGB::Yellow))
```



Parameters

`br`

The radius of the circumcircle of the regular base. This must be a real numerical value or an arithmetical expression of the animation parameter `a`.

br is equivalent to the attribute `BaseRadius`.

b_x, b_y, b_z

The lower center point. The coordinates b_x, b_y, b_z must be real numerical values or arithmetical expressions of the animation parameter `a`.

b_x, b_y, b_z are equivalent to the attributes `BaseX`, `BaseY`, `BaseZ`.

`tr`

The radius of the circumcircle of the regular top of the frustum of pyramid. This must be a real numerical value or an arithmetical expression of the animation parameter `a`. If no top radius is specified, a pyramid with top radius $tr = 0$ is created.

tr is equivalent to the attribute `TopRadius`.

t_x, t_y, t_z

The upper center point. The coordinates t_x, t_y, t_z must be real numerical values or arithmetical expressions of the animation parameter `a`.

t_x, t_y, t_z are equivalent to the attributes `TopX`, `TopY`, `TopZ`.

a

Animation parameter, specified as $a = a_{min}..a_{max}$, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Cone` | `plot::Cylinder` | `plot::Prism`

plot::QQplot

Statistical quantile-quantile plots

Syntax

```
plot::QQplot([a1, a2, …], [b1, b2, …], <a = amin .. amax>, options)
plot::QQplot([[a1, a2, …], [b1, b2, …]], <a = amin .. amax>, options)
plot::QQplot(A, <a = amin .. amax>, options)
plot::QQplot(s, <c1, c2>, <a = amin .. amax>, options)
plot::QQplot(s, <[c1, c2]>, <a = amin .. amax>, options)
```

Description

`plot::QQplot(data1, data2)` plots the quantiles of the first data set against the quantiles of the second data set.

`plot::QQplot` creates a quantile-quantile plot of two discrete data samples `[a1, a2, …]` and `[b1, b2, …]`. A QQ plot displays the collection of points with coordinates `[x1, y1]`, `[x2, y2]` etc., where `x_i = stats::empiricialQuantile([a_1, a_2, Symbol::dots])(i/(n - 1))` and `y_i = stats::empiricialQuantile([b_1, b_2, Symbol::dots])(i/(n - 1))` with `i` running from 0 through n - 1. The number of plot points n is set by the attribute `Size` = n. If no value is specified by the user, n is chosen as the minimum of the lengths of the data lists [a_1, a_2, …] and [b_1, b_2, …].

In addition, the diagonal reference line $y = x$ is displayed in the plot. This line can be suppressed by the attribute `LinesVisible` = `FALSE`.

The samples [a_1, a_2, …] and [b_1, b_2, …] do not need to have the same length.

A QQ plot is a graphical technique for determining if two data sets come from populations with a common distribution.

If the two sets come from a population with the same distribution, the points of the QQ plot should fall approximately along the reference line $y = x$. The greater the departure

from this reference line, the greater the evidence for the conclusion that the two data sets have come from populations with different distributions.

A specialized version of the QQ plot is the "probability plot", where the quantiles of one of the data samples are replaced with the quantiles of a theoretical distribution. You can use `plot::QQplot` for this type of plot, too, by using a reference list such as

```
[stats::normalQuantile(0, 1)(i/n) $ i = 1 .. n-1]
```

as one of the data lists. In this particular case, data obeying a standard normal distribution should produce plot points close to the diagonal reference line $y = x$.

Cf. "Example 3" on page 24-755.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`
`Data`	the (statistical) data to plot	
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`LineColor`	color of lines	`RGB::Red`
`LineWidth`	width of lines	`0.35`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LinesVisible`	visibility of lines	`TRUE`
`Name`	the name of a plot object (for browser and legend)	

Attribute	Purpose	Default Value
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointColor	the color of points	RGB::Black
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	TRUE
Size	size of a point list	
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	

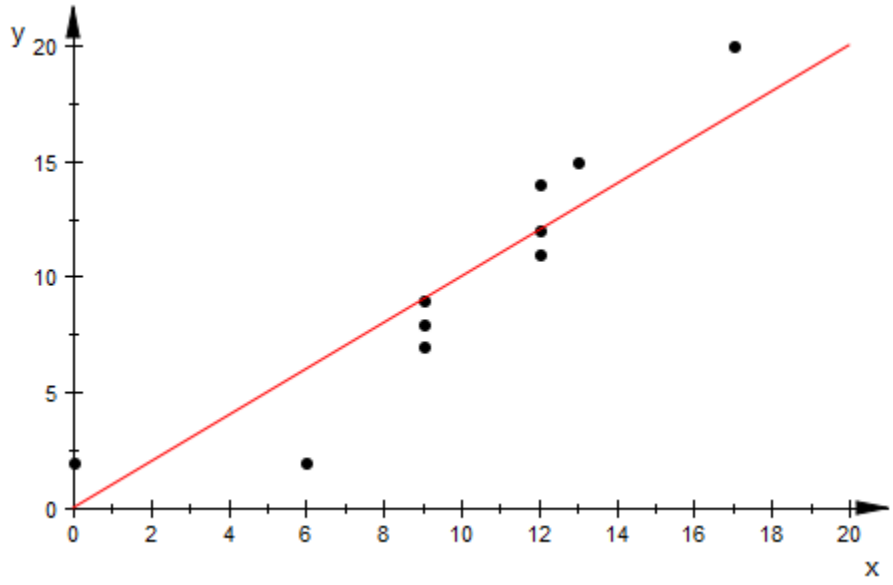
Attribute	Purpose	Default Value
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

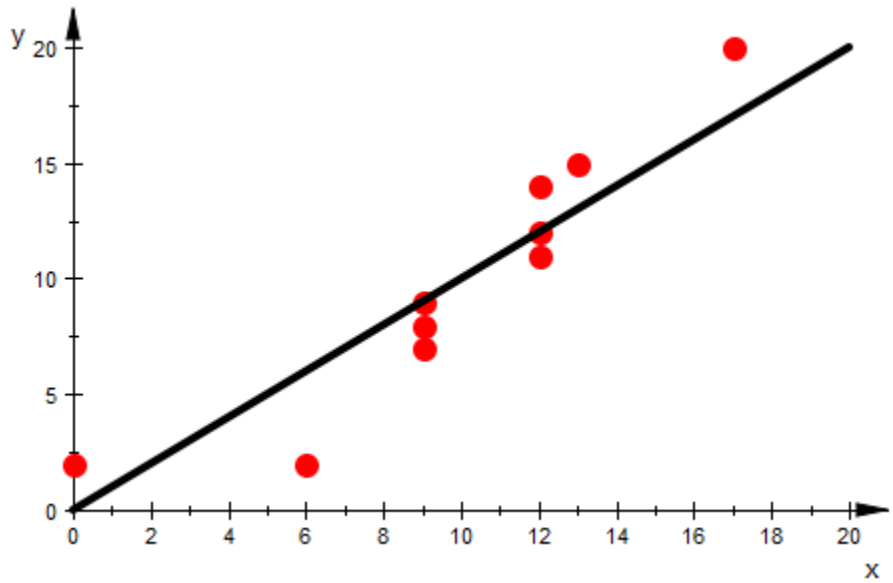
We create a QQ plot of some data samples:

```
a := [6, 9, 17,  0, 13, 9, 9, 12, 12, 12]:
b := [7, 8, 20,  2, 11, 8, 9, 12, 13, 15, 2, 14]:
q := plot::QQplot(a, b):
plot(q)
```



We can modify the appearance of the QQ plot in various ways:

```
q::PointColor := RGB::Red:
q::PointSize := 3*unit::mm:
q::LineColor := RGB::Black:
q::LineWidth := 1*unit::mm:

plot(q)
```




```
delete a, b, q:
```

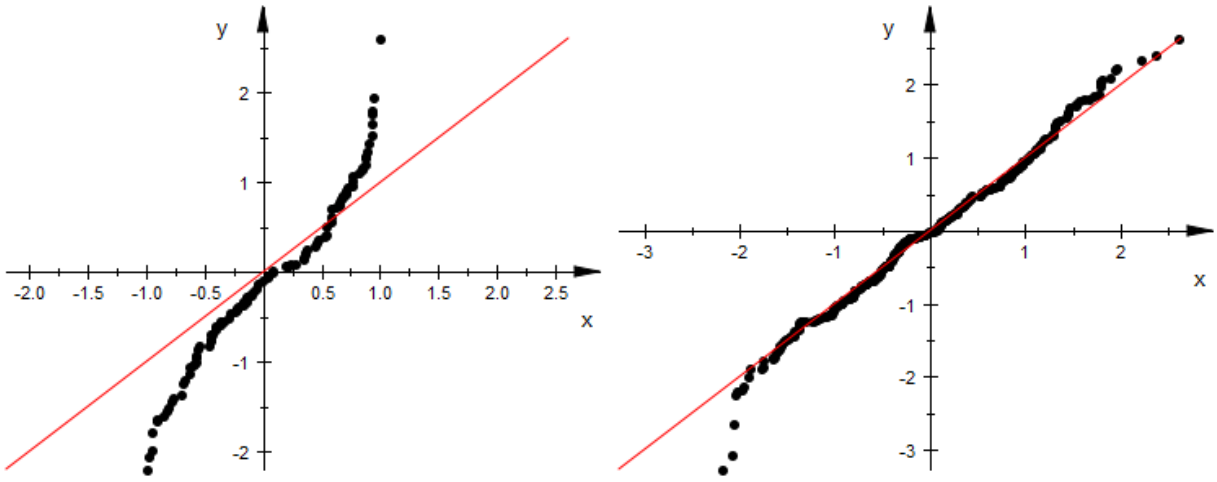
Example 2

We create some samples:

```
a := [stats::uniformRandom(-1, 1)() $ k = 1..100]:
b := [stats::normalRandom(0, 1)() $ k = 1..300]:
c := [stats::normalRandom(0, 1)() $ k = 1..500]:
```

The left QQ-plot shows a clear deviation from the reference line $y = x$. The samples a and b do not seem to be chosen from the same population. The QQ plot of the samples b and c (both normally distributed with mean 0 and variance 1), however, shows data points close to the reference line:

```
plot(plot::Scene2d(plot::QQplot(a, b)),
     plot::Scene2d(plot::QQplot(b, c)),
     Width = 20*unit::cm, Rows = 1)
```




```
delete a, b, c:
```

Example 3

We create a normally distributed sample:

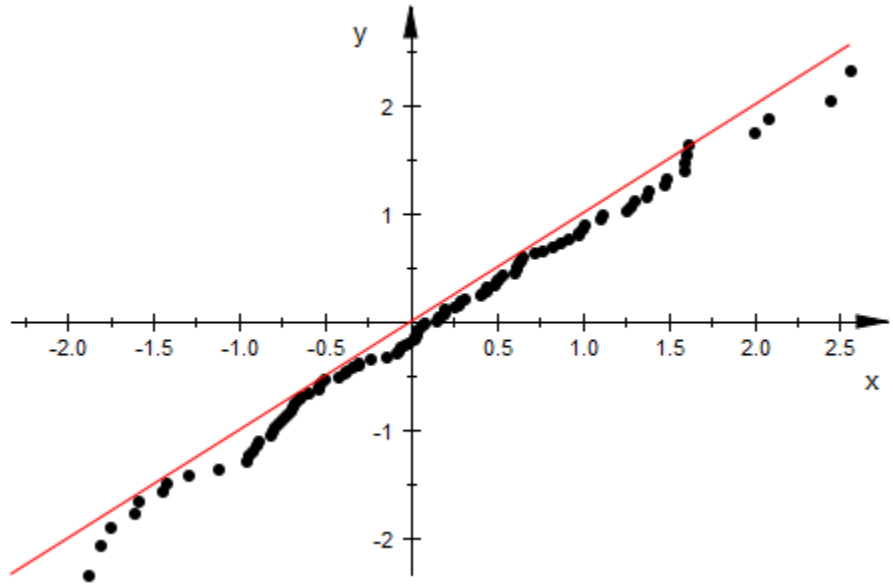
```
data1 := [stats::normalRandom(0, 1)() $ k = 1..100]:
```

We wish to investigate whether these data can indeed be regarded as normally distributed. We create a reference sample of data that are definitely normally distributed:

```
n:= nops(data1):
data2 := [stats::normalQuantile(0, 1)(i/n) $ i = 1 .. n-1]:
```

The QQ plot of the data shows plot points close to the reference line $y = x$:

```
plot(plot::QQplot(data1, data2))
```




```
delete data1, n, data2:
```

Parameters

a_1, a_2, …, b_1, b_2, …

The statistical data: numerical real values or arithmetical expressions of the animation parameter `a`.

a_1, a_2, …, b_1, b_2, … are equivalent to the attribute `Data`.

A

An array of domain type `DOM_ARRAY` or a matrix of category `Cat::Matrix` (e.g., of type `matrix` or `densematrix`) providing numerical real values or arithmetical expressions of the animation parameter `a`. The array/matrix must have 2 columns. The first column is regarded as the data set [a_1, a_2, …] the second column is regarded as the data set [b_1, b_2, …]. If more columns are provided, the superfluous columns are ignored.

A is equivalent to the attribute `Data`.

s

A data collection of domain type `stats::sample`. Two columns in `s` are regarded as the data lists [a_1, a_2, ...] and [b_1, b_2, ...] respectively.

s is equivalent to the attribute `Data`.

c_1, c_2

Column indices into `s`: positive integers. These indices, if given, indicate that only the specified columns in `s` should be used. If no column indices are specified, the first two columns in `s` are used as the data sets [a_1, a_2, ...] and [b_1, b_2, ...], respectively.

a

Animation parameter, specified as $a = a_{min}..a_{max}$, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Bars2d` | `plot::Bars3d` | `plot::Boxplot` | `plot::Histogram2d` | `plot::Listplot` | `plot::Scatterplot`

plot::Raster

Raster plot

Syntax

```
plot::Raster(A, options)
plot::Raster(A, x = x_min .. x_max, y = y_min .. y_max, <a = a_min .. a_max>, options)
plot::Raster(L, options)
plot::Raster(L, x = x_min .. x_max, y = y_min .. y_max, <a = a_min .. a_max>, options)
```

Description

`plot::Raster(A, x = x_min..x_max, y = y_min..y_max)` translates a matrix *A* of RGB values into a regular 2D mesh of rectangles extending from the lower left corner (x_{min}, y_{min}) to the upper right corner (x_{max}, y_{max}). The rectangles are colored according to the color entries in *A*.

`plot::Raster` serves for generating 2D raster objects such as bitmaps. External bitmap data can be imported to a MuPAD session via `import::readbitmap`. The resulting array of color values can be passed directly to `plot::Raster` to embed the imported bitmap in a 2D MuPAD scene.

When color values are specified by an array or a matrix *A*, the low indices correspond to the lower left corner of the graphics. The high indices correspond to the upper right corner.

Note: Note that the bitmap data of most standard graphical formats are stored in the usual Western reading order: the first pixels correspond to the upper left corner, the last pixels correspond to the lower right corner. The utility `import::readbitmap` produces an array in which the first element corresponds to the lower left corner. Bitmap data imported this way can be passed directly to `plot::Raster`.

Arrays/matrices do not need to be indexed from 1. E.g.,

`A = array( `i_{min}`..`i_{max}`, `j_{min}`..`j_{max}`, [..RGB values..])`

yields a graphical array with

`XMesh = `j_{max}` - `j_{min}` + 1`, `YMesh = `i_{max}` - `i_{min}` + 1`.

If no plot range $x_{min}..x_{max}$, $y_{min}..y_{max}$ is specified,

x_{min} = j_{min} - 1, x_{max} = j_{max}, y_{min} = i_{min} - 1, y_{max} = i_{max}

is used.

When color values are specified by a list of lists *L*, the first entries in the list correspond to the lower left corner of the graphics. The last entries correspond to the upper right corner.

If no plot range $x_{min}..x_{max}$, $y_{min}..y_{max}$ is specified,

x_{min} = 0, x_{max} = m, y_{min} = 0, y_{max} = n

is used, where *n* is the length of *L* and *m* is the (common) length of the sublists in *L*. All sublists ("rows") must have the same length.

Animations are triggered by specifying a range `a = amin .. amax` for a parameter `a` that is different from the variables `x`, `y`. Thus, in animations, both the ranges $x = x_{min}..x_{max}$, $y = y_{min}..y_{max}$ as well as the animation range $a = a_{min}..a_{max}$ must be specified.

The related plot routine `plot::Density` provides a similar functionality offering an automatic color scheme based on scalar density values.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`FALSE`
`Color`	the main color	`RGB::Blue`
`ColorData`	color values of a raster plot	

Attribute	Purpose	Default Value
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Blue
LineWidth	width of lines	0.35
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	FALSE
Mesh	number of sample points	[11, 11]
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	

Attribute	Purpose	Default Value
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XMax	final value of parameter “x”	
XMesh	number of sample points for parameter “x”	11
XMin	initial value of parameter “x”	
XName	name of parameter “x”	
XRange	range of parameter “x”	
YMax	final value of parameter “y”	
YMesh	number of sample points for parameter “y”	11
YMin	initial value of parameter “y”	
YName	name of parameter “y”	
YRange	range of parameter “y”	

Examples

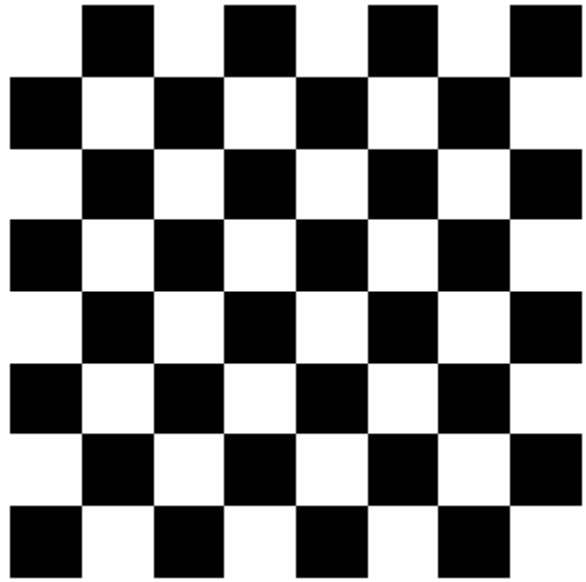
Example 1

We generate a raster plot:

```
checkerboard:= array(1..8, 1..8):
for i from 1 to 8 do
 for j from 1 to 8 do
    if i + j mod 2 = 0 then
      checkerboard[i,j] := RGB::Black;
    else
      checkerboard[i,j] := RGB::White;
    end_if;
 end_for:
end_for:
p := plot::Raster(checkerboard):
```

The plot object is rendered:

```
plot(p):
```



```
delete checkerboard, p:
```

Example 2

We import an external bitmap file:

```
[width, height, colordata] := import::readbitmap("Eva.jpeg"):
```

The array `colordata` can be passed directly to `plot::Raster`:

```
scenewidth:= 80*unit::mm:
sceneheight:= height/width*scenewidth:
plot(plot::Raster(colordata),
     Width  = scenewidth,
     Height = sceneheight,
     Footer = "This is Eva"):
```


```
delete width, height, colordata, scenewidth, sceneheight:
```

Example 3

This is Tom:

```
[widthT, heightT, Tom] :=
       import::readbitmap("Tom.jpeg", ReturnType = DOM_ARRAY):
plot(plot::Raster(Tom), Width = widthT/3, Height = heightT/3):
```

This is Jerry:

```
[widthJ, heightJ, Jerry] :=
       import::readbitmap("Jerry.jpeg", ReturnType = DOM_ARRAY):
plot(plot::Raster(Jerry), Width = widthT/3, Height = heightT/3):
```

Although they look different, they are topologically equivalent. We demonstrate this by deforming Tom to Jerry via a smooth map $(1 - a)\,T + a\,J$, $a \in [0, 1]$:

```
blend := (T, J, a) -> zip(T, J, (t,j) -> (1-a)*t + a*j):
Tom2Jerry:= array(1..heightT, 1..widthT):
for i from 1 to heightT do
  for j from 1 to widthT do
    Tom2Jerry[i, j]:= blend(Tom[i, j], Jerry[i, j], a):
  end_for:
end_for:
```

The following call produces an animated plot of the deformation. Note that x and y ranges must be specified for an animation:

```
plot(plot::Raster(Tom2Jerry,
                  x = 1..widthT,
                  y = 1..heightT,
                  a = 0..1, Frames = 10,
                  Footer = "Tom & Jerry"),
     Width = widthT/3, Height = heightT/3):
```

This is the arithmetical mean of Tom and Jerry:

```
plot(plot::Raster(map(subs(Tom2Jerry, a = 0.5), eval)),
     Footer = "(Tom + Jerry)/2", FooterFont = [12],
     Width = widthT/3, Height = heightT/3):
```

(Tom + Jerry)/2

Parameters

A

An array of domain type `DOM_ARRAY` or a matrix of category `Cat::Matrix` (e.g., of type `matrix` or `densematrix`) providing `RGB` values or color expressions of the animation parameter *a*. Rows/columns of the array, respectively matrix, correspond to rows/columns of the graphical array.

A is equivalent to the attribute `ColorData`.

L

A list of lists `RGB` values or color expressions of the animation parameter *a*. Each sublist of *L* represents a row of the graphical array.

L is equivalent to the attribute `ColorData`.

x

Name of the horizontal variable: an identifier or an indexed identifier. It is used as the title of the coordinate axis in *x* direction.

x is equivalent to the attribute `XName`.

x_{min} .. x_{max}

The range of the horizontal variable: x_{min}, x_{max} must be numerical real value or expressions of the animation parameter *a*.

x_{min} .. x_{max} is equivalent to the attributes `XRange`, `XMin`, `XMax`.

y

Name of the vertical variable: an identifier or an indexed identifier. It is used as the title of the coordinate axis in *y* direction.

y is equivalent to the attribute `YName`.

y_{min} .. y_{max}

The range of the vertical variable: y_{min}, y_{max} must be numerical real value or expressions of the animation parameter *a*.

y_{min} .. y_{max} is equivalent to the attributes `YRange`, `YMin`, `YMax`.

a

Animation parameter, specified as $a = a_{min}..a_{max}$, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`import::readbitmap` | `plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Density` | `plot::Inequality`

plot::Rectangle

Rectangles in 2D

Syntax

```
plot::Rectangle(x_min .. x_max, y_min .. y_max, <a = a_min .. a_max>, options)
```

Description

`plot::Rectangle( `x_{min}`..`x_{max}` , `y_{min}`..`y_{max}` )` generates the 2D rectangle with the corners (x_{min}, y_{min}), (x_{min}, y_{max}), (x_{max}, y_{min}), (x_{max}, y_{max}).

`plot::Rectangle` creates a 2D rectangle with edges parallel to the coordinate axes.

With `Filled` = `FALSE`, the rectangle consists only of its edges. With `Filled` = `TRUE`, it is a filled area.

The lines can be set as desired with `LineStyle`, `LineWidth`, and `LineColor`. Cf. “Example 1” on page 24-772.

With `LinesVisible = FALSE`, the edges are rendered invisible.

For filled rectangles, a `FillColor` and a `FillPattern` can be selected. Cf. “Example 2” on page 24-772.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`FALSE`
`Color`	the main color	`RGB::Blue`
`Filled`	filled or transparent areas and surfaces	`FALSE`

Attribute	Purpose	Default Value
FillColor	color of areas and surfaces	RGB::Red
FillPattern	type of area filling	DiagonalLines
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Blue
LineWidth	width of lines	0.35
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
LineColorDirection	the direction of color transitions on lines	[0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	1
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
TimeEnd	end time of the animation	10.0

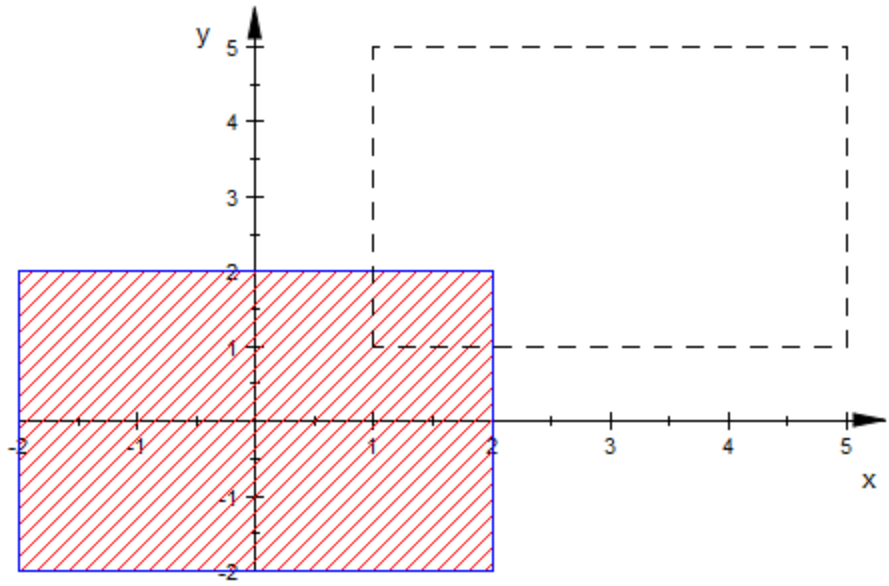
Attribute	Purpose	Default Value
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XMax	final value of parameter “x”	1
XMin	initial value of parameter “x”	-1
XRange	range of parameter “x”	-1 .. 1
YMax	final value of parameter “y”	1
YMin	initial value of parameter “y”	-1
YRange	range of parameter “y”	-1 .. 1

Examples

Example 1

We plot two rectangles:

```
plot(plot::Rectangle(-2..2, -2..2, Filled = TRUE,
                     FillColor = RGB::Red),
     plot::Rectangle(1..5, 1..5, Filled = FALSE,
                     LineColor = RGB::Black,
                     LineStyle = Dashed))
```

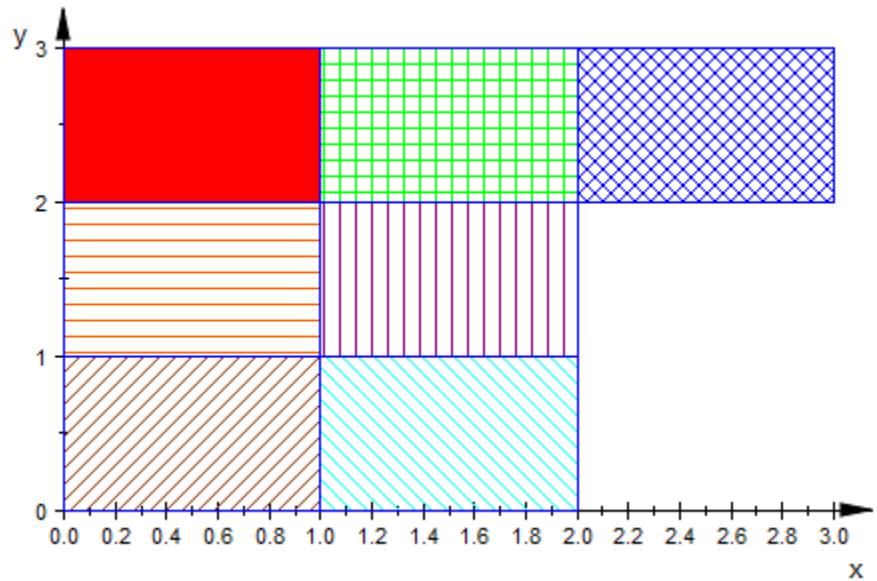


Example 2

We plot rectangles with different fill patterns (`FillPattern`):

```
plot(plot::Rectangle(0..1, 2..3, Filled = TRUE,
                     FillPattern = Solid,
```

```
                FillColor = RGB::Red),
plot::Rectangle(1..2, 2..3, Filled = TRUE,
                FillPattern = CrossedLines,
                FillColor = RGB::Green),
plot::Rectangle(2..3, 2..3, Filled = TRUE,
                FillPattern = XCrossedLines,
                FillColor = RGB::Blue),
plot::Rectangle(0..1, 1..2, Filled = TRUE,
                FillPattern = HorizontalLines,
                FillColor = RGB::Orange),
plot::Rectangle(1..2, 1..2, Filled = TRUE,
                FillPattern = VerticalLines,
                FillColor = RGB::Violet),
plot::Rectangle(0..1, 0..1, Filled = TRUE,
                FillPattern = DiagonalLines,
                FillColor = RGB::Brown),
plot::Rectangle(1..2, 0..1, Filled = TRUE,
                FillPattern = FDiagonalLines,
                FillColor = RGB::Cyan))
```



Parameters

x_{min} .. x_{max}

The left and right border of the rectangle: real numerical values or arithmetical expressions of the animation parameter `a`.

x_{min} .. x_{max} is equivalent to the attributes `XRange`, `XMin`, `XMax`.

y_{min} .. y_{max}

The lower and upper border of the rectangle: real numerical values or arithmetical expressions of the animation parameter `a`.

y_{min} .. y_{max} is equivalent to the attributes `YRange`, `YMin`, `YMax`.

a

Animation parameter, specified as a = a_{min}..a_{max}, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Box` | `plot::Line2d` | `plot::Line3d` | `plot::Polygon2d` | `plot::Polygon3d`

plot::Rootlocus

Curves of roots of rational expressions

Syntax

```
plot::Rootlocus(p(z, u), u = u_min .. u_max, <a = a_min .. a_max>, options)
```

Description

`plot::Rootlocus(p(z, u), u = u_min .. u_max)` creates a 2D plot of the curves in the complex plane given by the roots of $p(z, u) = 0$ (solved for z) as the parameter u varies between u_{min} and u_{max}.

For any given value of u, `plot::Rootlocus` solves the equation $p(z, u) = 0$ for z. The solutions define points with coordinates $x = \Re(z)$, $y = \Im(z)$ in the complex plane. As the parameter u varies, the solutions form continuous curves that a depicted by `plot::Rootlocus`.

The roots of the numerator of `p(z, u)` are considered. All complex solutions of this polynomial in z are computed numerically via `numeric::polyroots`.

The polynomial is initially solved for some values u from the range `u = `u_{min}` .. `u_{max}``. The optional argument `Mesh` = n can be used to specify the number n of these initial points (the default value is 51). These points are not equally spaced, but accumulate close to the end of the range.

The routine then tries to pair up the roots for adjacent values of u by choosing those closest to each other.

Finally, the routine tries to trace out the different curves by joining up adjacent points with line segments. If adjacent line segments exhibit angles that are not close to 180 degrees, additional roots are computed for parameter values u between the values of the initial mesh. Up to m such bisectioning steps are possible, where m is specified by the optional argument `AdaptiveMesh` = m (the default value is 4). With `AdaptiveMesh` = 0, this adaptive mechanism may be switched off.

Sometimes, the matching up of the roots to continuous curves can be fooled and the result is a messy plot. In such a case, the user can take the following measures to improve the plot:

- The parameter range `u = `u_{min}` .. `u_{max}`` may be unreasonably large. Reduce this range to a reasonable size!
- Increase the size n of the initial mesh using the option `Mesh` = n. Note that increasing n by some factor may increase the runtime of the plot by the same factor!
- Increase the number m of possible adaptive bisectioning steps using the option `AdaptiveMesh` = m. Note that increasing m by 1 may increase the runtime of the plot by a factor of 2!
- Using the options `LinesVisible = FALSE` in conjunction with `PointsVisible = TRUE`, the roots are displayed as separate points without joining line segments.

Cf. “Example 2” on page 24-783.

Animations are triggered by specifying a range `a = `a_{min}` .. `a_{max}`` for a parameter `a` that is different from the variables `z` and `u`. Cf. “Example 3” on page 24-787.

The curves can be colored by a user defined color scheme. Just pass the option `LineColorFunction` = `mycolor`, where `mycolor` is a user definied procedure that returns an `RGB` color value. The routine `plot::Rootlocus` calls `mycolor(u, x, y)`, where u is the parameter value and x, y are the real and imaginary parts of the root of $p(x + \mathrm{i}\,y, u) = 0$. Cf. “Example 4” on page 24-789.

Attributes

Attribute	Purpose	Default Value
`AdaptiveMesh`	adaptive sampling	`4`
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	

Attribute	Purpose	Default Value
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Blue
LineWidth	width of lines	0.35
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
Mesh	number of sample points	51
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.0
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE
RationalExpression	rational expression in a rootlocus plot	
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0

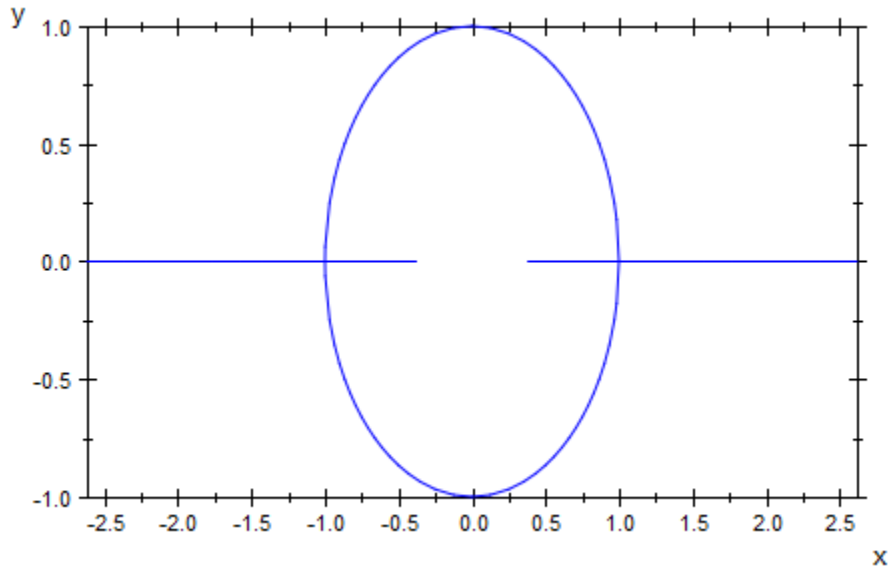
Attribute	Purpose	Default Value
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`
`Title`	object title	
`TitleFont`	font of object titles	`[" sans-serif ", 11]`
`TitlePosition`	position of object titles	
`TitleAlignment`	horizontal alignment of titles w.r.t. their coordinates	`Center`
`TitlePositionX`	position of object titles, x component	
`TitlePositionY`	position of object titles, y component	
`UMax`	final value of parameter “u”	
`UMesh`	number of sample points for parameter “u”	`51`
`UMin`	initial value of parameter “u”	
`UName`	name of parameter “u”	
`URange`	range of parameter “u”	
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

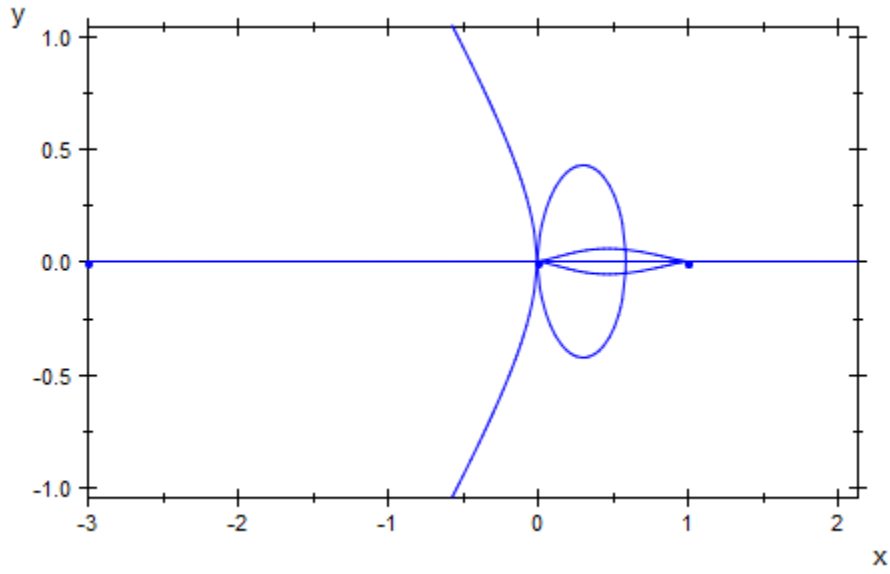
The roots of the polynomial z^2 - 2 u z + 1 are given by $z = u + \sqrt{u^2 - 1}$ and $z = u - \sqrt{u^2 - 1}$. We visualise these two curves via a rootlocus plot:

```
plot(plot::Rootlocus(z^2 - 2*u*z + 1, u = -1.5..1.5))
```



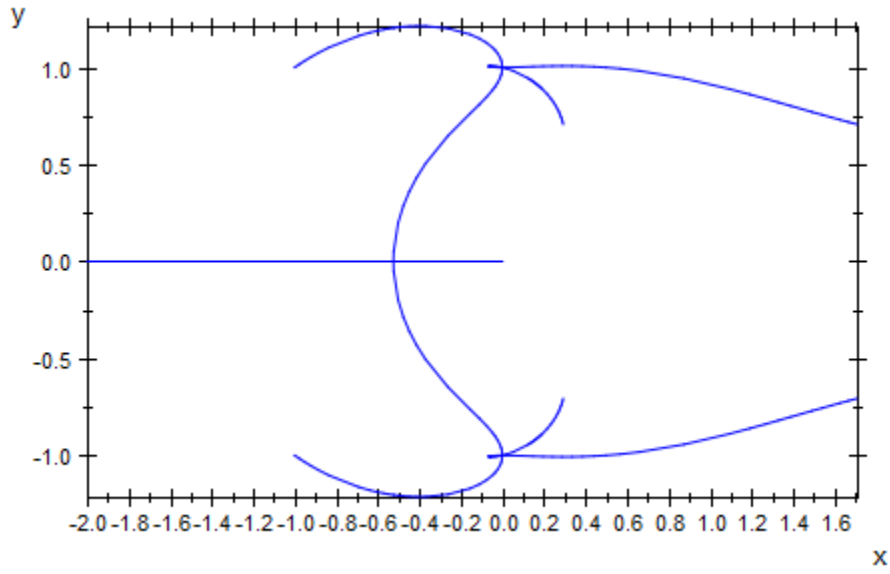
For rational expressions, the roots of the numerator are considered. The following plot displays the roots of the numerator polynomial (z^2 - u)^2 + u (z - u)^3:

```
plot(plot::Rootlocus(1 + u * (z - u)^3/(z^2 - u)^2, u = -1..1)):
```

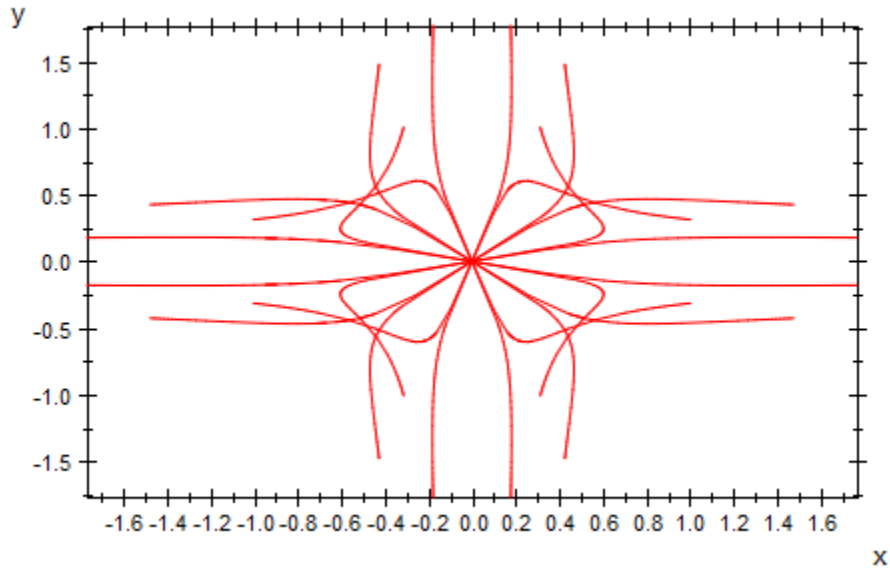


Here are various other examples:

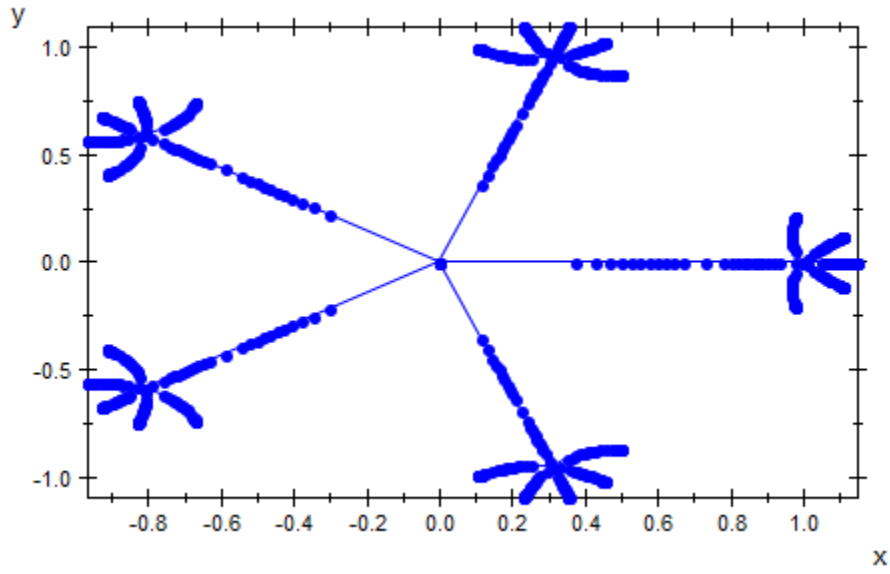
```
plot(plot::Rootlocus((z^2 - 2*u*z + 1)^2 + u, u = -1..1))
```




```
plot(plot::Rootlocus((z^2 - u)^6 + u^2, u = -2..2,
                                    Color = RGB::Red))
```



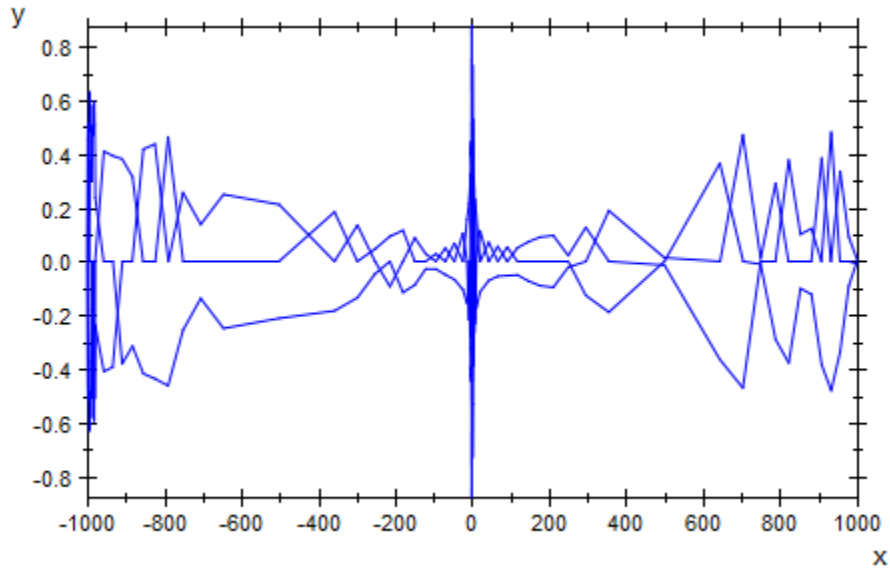

```
plot(plot::Rootlocus((z^5  - 1)^3 + u, u = -1..1, PointsVisible,
                                                   PointSize = 1.5))
```



Example 2

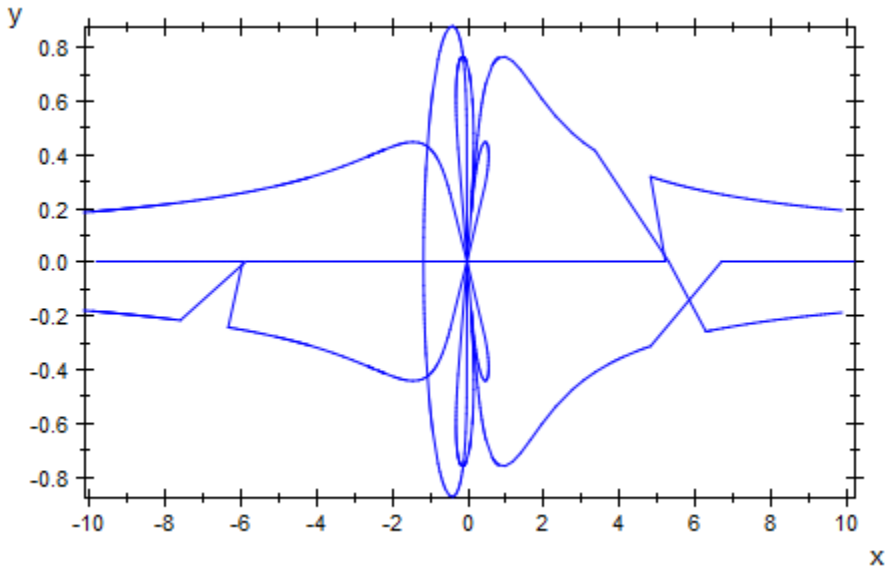
The following plot is rather messy, since the default mesh size of 51 initial points on each curve is not sufficient to obtain a good resolution:

```
plot(plot::Rootlocus((z-u)^3 - u/z^3, u = -10^3 .. 10^3)):
```



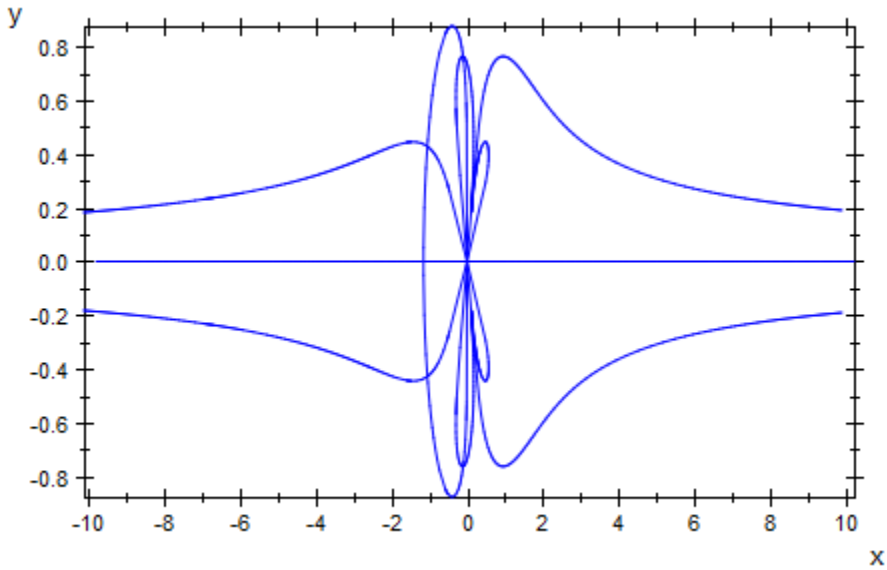
We obtain a better resolution by decreasing the range of the parameter *u* to a reasonable size. There are still a few points that are not properly matched up with the curves:

```
plot(plot::Rootlocus((z-u)^3 - u/z^3, u = -10 .. 10)):
```



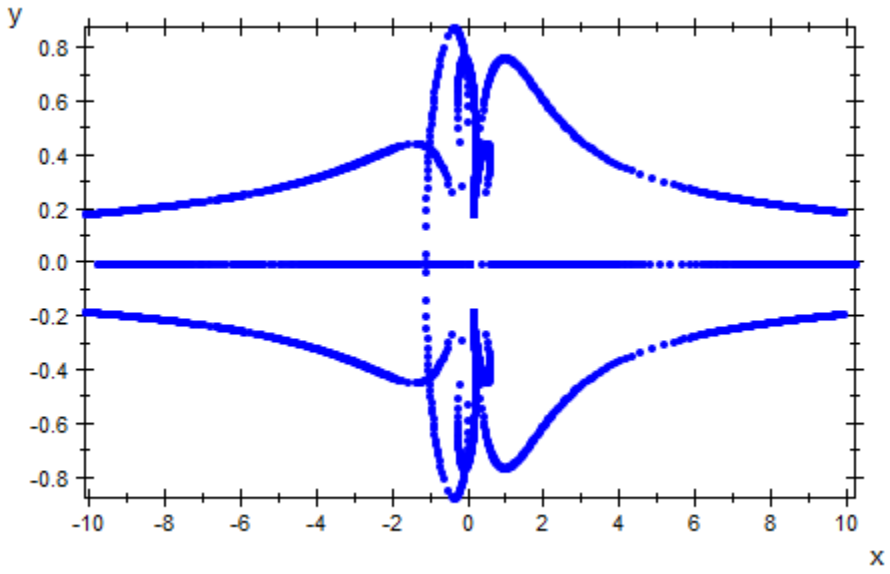
We increase the mesh size to cure this problem:

```
plot(plot::Rootlocus((z-u)^3 - u/z^3, u = -10 .. 10, Mesh = 251)):
```



We plot the roots as separate points without displaying connecting line segments:

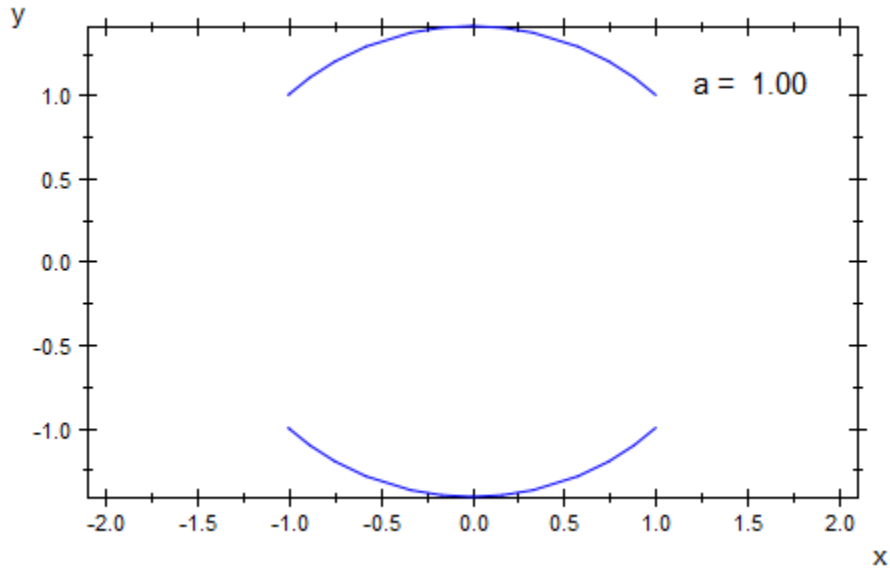
```
plot(plot::Rootlocus((z-u)^3 - u/z^3, u = -10 .. 10, Mesh = 501,
                     LinesVisible = FALSE, PointsVisible)):
```



Example 3

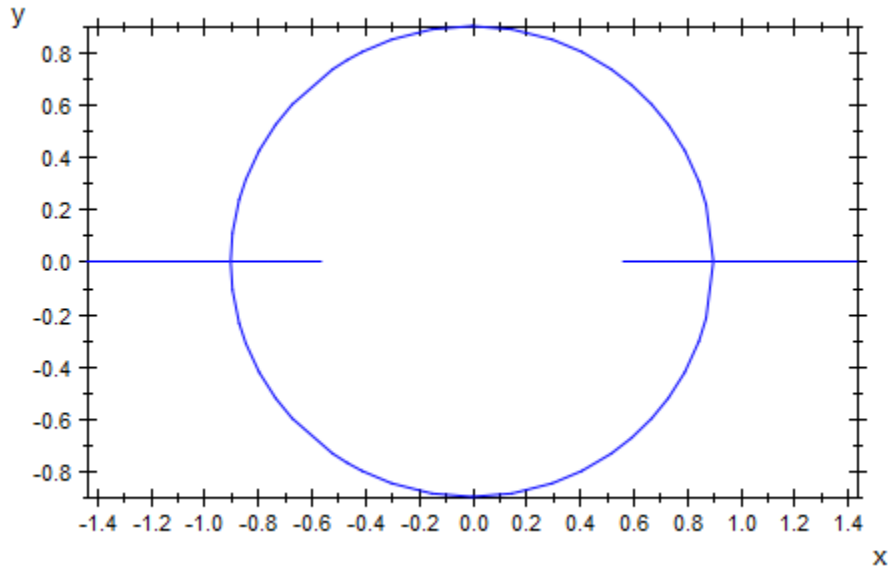
We animate the expression whose roots are to plotted:

```
plot(plot::Rootlocus(z^2 - 2*u*z + a, u = -1..1, a = -0.2 .. 2, Mesh = 10),
     plot::Text2d(a -> "a = ".stringlib::formatf(a, 2, 5), [1.2, 1.0], a = -0.2 .. 1));
```



We animate the parameter range:

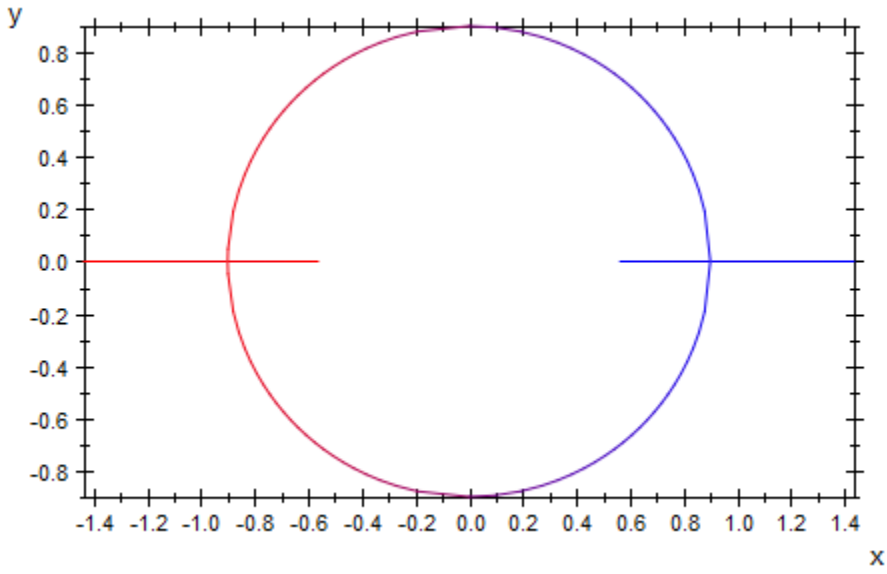
```
plot(plot::Rootlocus(z^2 - 2*u*z + 0.81, u = -1 .. a, a = -1 .. 1, Mesh = 10))
```



Example 4

We provide a color function: roots for small values of the parameter *u* are displayed in red, whereas roots for large parameter values are displayed in blue:

```
plot(plot::Rootlocus(z^2 - 2*u*z + 0.81, u = -1..1,
                     LineColorFunction = ((u, x, y) -> [(1 - u)/2, 0, (1 + u)/2])))
```



Parameters

`p(z, u)`

An arithmetical expression in two unknowns *z* and *u* and, possibly, the animation parameter `a`. It must be a rational expression in *z*.

p(z, u) is equivalent to the attribute `RationalExpression`.

`z`

Name of the unknown: an identifier or an indexed identifier.

`u`

Name of the curve parameter: an identifier or an indexed identifier.

u is equivalent to the attribute `UName`.

u_{min} .. u_{max}

The range of the curve parameter: u_{min}, u_{max} must be numerical real values or expressions of the animation parameter a.

u_{min} .. u_{max} is equivalent to the attributes `URange`, `UMin`, `UMax`.

a

Animation parameter, specified as a = a_{min}..a_{max}, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions

`numeric::polyroots` | `plot` | `plot::copy`

MuPAD Graphical Primitives

`plot::Curve2d`

plot::Scatterplot

Statistical scatter plots

Syntax

```
plot::Scatterplot([x1, x2, …], [y1, y2, …], <a = amin .. amax>, options)
plot::Scatterplot([[x1, x2, …], [y1, y2, …]], <a = amin .. amax>, options)
plot::Scatterplot([x1, y1], [x2, y2], …, <a = amin .. amax>, options)
plot::Scatterplot([[x1, y1], [x2, y2], …], <a = amin .. amax>, options)
plot::Scatterplot(A, <a = amin .. amax>, options)
plot::Scatterplot(s, <c1, c2>, <a = amin .. amax>, options)
```

Description

`plot::Scatterplot` creates a scatter plot of two discrete data samples `[x1, x2, …]` and `[y1, y2, …]`. A scatter plot displays the collection of points with coordinates `[x1, y1]`, `[x2, y2]` etc.

In addition, a regression line $y = a + b\,x$ through the given data pairs `[x1, y1]` etc. is computed and added to the plot. The estimators a, b of the regression are computed by `stats::linReg`.

The regression line can be suppressed by specifying the attribute `LinesVisible = FALSE`.

The samples $[x_1, x_2, \ldots]$ and $[y_1, y_2, \ldots]$ should have the same number of elements. Otherwise, superflous elements in the longer list are ignored.

There is an ambiguity between the various input formats if only 2 data points are provided:

Note: For two data points the calls `plot::Scatterplot([a, b], [c, d])` and `plot::Scatterplot([[a, b], [c, d]])` both yield plots of the two points `(x1, y1)`

= (a, b) and (x_2, y_2) = (c, d), not of the points (x_1, y_1) = (a, c) and (x_2, y_2) = (b, d)!

The routines `plot::Listplot` and `plot::PointList2d` have a similar functionality. The main additional feature of `plot::Scatterplot` is the regression line.

Scatter plots are useful to visualize the relationship between two variables x (the "predictor") and y (the "criterion").

The variable regarded as a predictor corresponds to the horizontal axis while the variable regarded as the criterion corresponds to the vertical axis. The criterion variable represents the behavior to be predicted. The predictor variable represents the activity which is believed to be associated with the criterion.

The scatter plot consists of points (x, y) where x is a predictor value and y is the corresponding value of the criterion.

If there is a linear relation $y = a + b\,x$ between x and y, the data points should form a line, potentially marred by statistical deviations. The regression line provided by the scatter plot allows a visual test of such a relation between x and y.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`
`Data`	the (statistical) data to plot	
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`LineColor`	color of lines	`RGB::Red`

Attribute	Purpose	Default Value
LineWidth	width of lines	0.35
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointColor	the color of points	RGB::Black
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	TRUE
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	

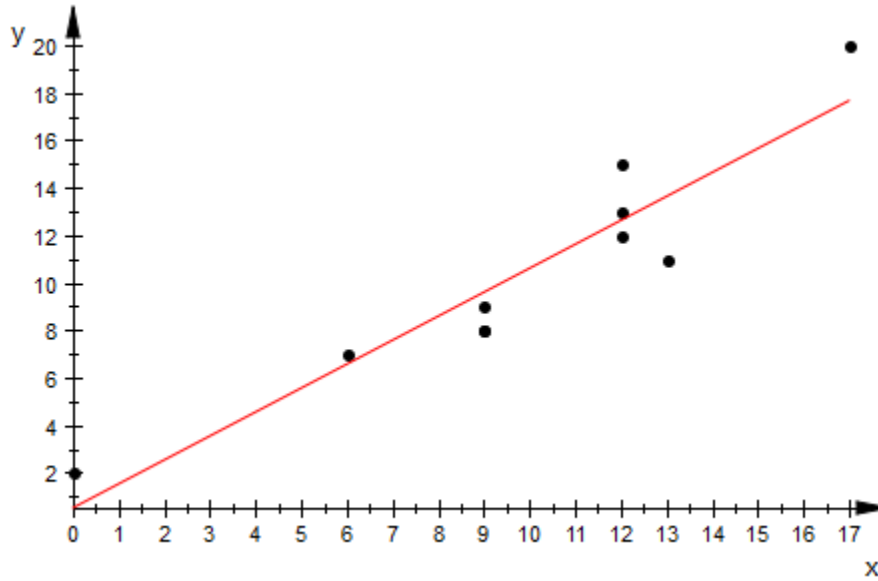
Attribute	Purpose	Default Value
`TitlePositionY`	position of object titles, y component	
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

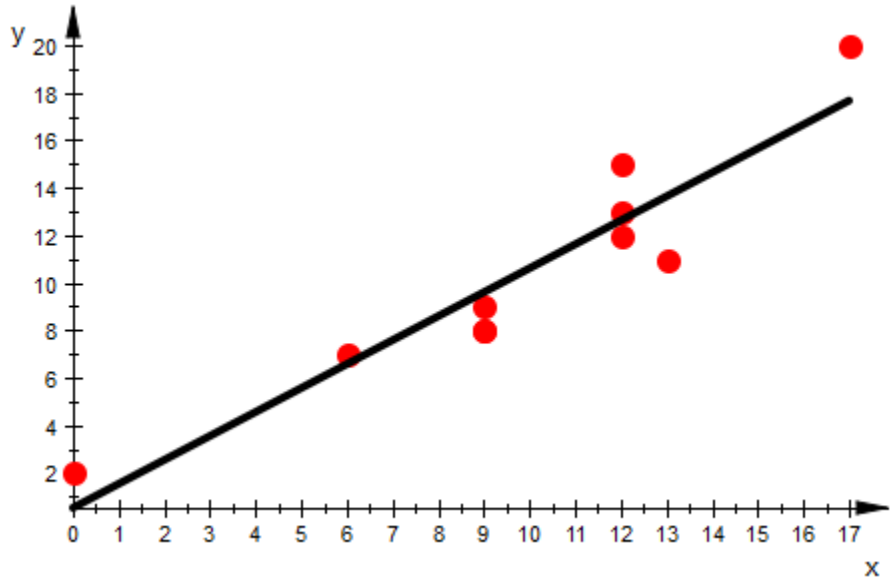
We plot some data samples:

```
xdata := [6, 9, 17,  0, 13, 9, 9, 12, 12, 12]:
ydata := [7, 8, 20,  2, 11, 8, 9, 12, 13, 15]:
b := plot::Scatterplot(xdata, ydata):
plot(b)
```



We can modify the appearance of the scatter plot in various ways:

```
b::PointColor := RGB::Red:
b::PointSize := 3*unit::mm:
b::LineColor := RGB::Black:
b::LineWidth := 1*unit::mm:

plot(b)
```




```
delete xdata, ydata, b:
```

Example 2

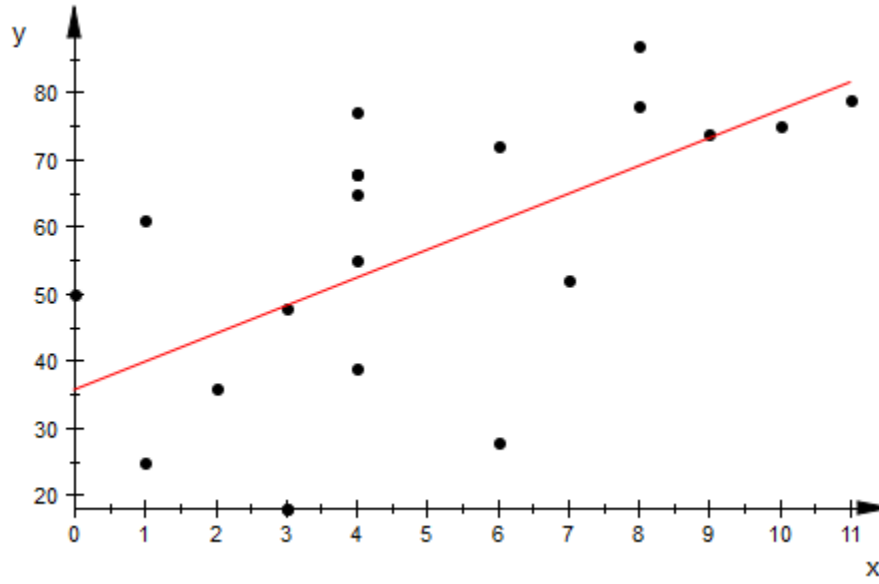
We analyze the relationship between the time students spent on preparing for a test and the result of the test. We collect the data in a matrix. Each row corresponds to a student. The first column describes the numbers of hours spent for the preparation, the second column contains the corresponding test score (points out of 100):

```
TimesAndScores := matrix([[ 1,  61],
                          [10,  75],
                          [4,   55],
                          [3,   18],
                          [4,   77],
                          [6,   72],
                          [3,   18],
                          [1,   25],
                          [0,   50],
                          [4,   68],
                          [4,   68],
```

```
  [8,   87],
  [9,   74],
  [11,  79],
  [6,   28],
  [4,   65],
  [7,   52],
  [8,   78],
  [2,   36],
  [3,   48],
  [4,   39]
]):
```

We draw a scatter plot to identify a possible relationship between the two variables:

```
plot(plot::Scatterplot(TimesAndScores))
```



There seems to be a relationship, indeed.

```
delete TimesAndScores:
```

Parameters

x_1, y_1, x_2, y_2, …

The statistical data: numerical real values or arithmetical expressions of the animation parameter `a`.

x_1, y_1, x_2, y_2, … is equivalent to the attribute `Data`.

A

An array of domain type `DOM_ARRAY` or a matrix of category `Cat::Matrix` (e.g., of type `matrix` or `densematrix`) providing numerical real values or arithmetical expressions of the animation parameter `a`. The i-th row is regarded as the data point (x_i, y_i). The array/matrix must have 2 columns. If more columns are provided, the superfluous columns are ignored.

A is equivalent to the attribute `Data`.

s

A data collection of domain type `stats::sample`. The columns in `s` are regarded as x- and y-values, respectively.

s is equivalent to the attribute `Data`.

c_1, c_2

Column indices into `s`: positive integers. These indices, if given, indicate that only the specified columns in `s` should be used. If no column indices are specified, the first two columns in `s` are used as x and y-values, respectively.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions

`plot` | `plot::copy` | `stats::correlation` | `stats::linReg`

MuPAD Graphical Primitives
`plot::Bars2d` | `plot::Bars3d` | `plot::Boxplot` | `plot::Histogram2d` | `plot::Listplot`

plot::Sequence

Sequences

Syntax

`plot::Sequence(y, n = n_1 .. n_2, <a = a_min .. a_max>, options)`

`plot::Sequence(x, y, n = n_1 .. n_2, <a = a_min .. a_max>, options)`

Description

`plot::Sequence(y(n), n = n_1 .. n_2 )` creates the points

$$(n_1, y(n_1)), (n_1+1, y(n_1+1)), \ldots, (n_2, y(n_2))$$

.

`plot::Sequence(x(n), y(n), n = n_1 .. n_2 )` creates the sequence of points

$$(x(n_1), y(n_1)), (x(n_1+1), y(n_1+1)), \ldots, (x(n_2), y(n_2))$$

.

`plot::Sequence` creates graphs of sequences, i.e., functions and curves defined over (some subset of) the integers.

`plot::Sequence(y(n), n = n_1..n_2 )` is functionally equivalent to the call `plot::PointList2d([[n, y(n)] $ n = n_1..n_2 )`, and `plot::Sequence(x(n), y(n), n = n_1..n_2 )` creates the same image as `plot::PointList2d([[x(n), y(n)] $ n = n_1..n_2 )`. See “Example 2” on page 24-805 for some extra functionality.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`

Attribute	Purpose	Default Value
`Color`	the main color	`RGB::Blue`
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`LineColor`	color of lines	`RGB::Blue`
`LineWidth`	width of lines	`0.35`
`LineColor2`	color of lines	`RGB::DeepPink`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LinesVisible`	visibility of lines	`FALSE`
`LineColorType`	line coloring types	`Flat`
`LineColorFunction`	functional line coloring	
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`PointSize`	the size of points	`2`
`PointStyle`	the presentation style of points	`FilledCircles`
`PointsVisible`	visibility of mesh points	`TRUE`

Attribute	Purpose	Default Value
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
UMax	final value of parameter “u”	
UMin	initial value of parameter “u”	
UName	name of parameter “u”	
URange	range of parameter “u”	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XFunction	function for x values	

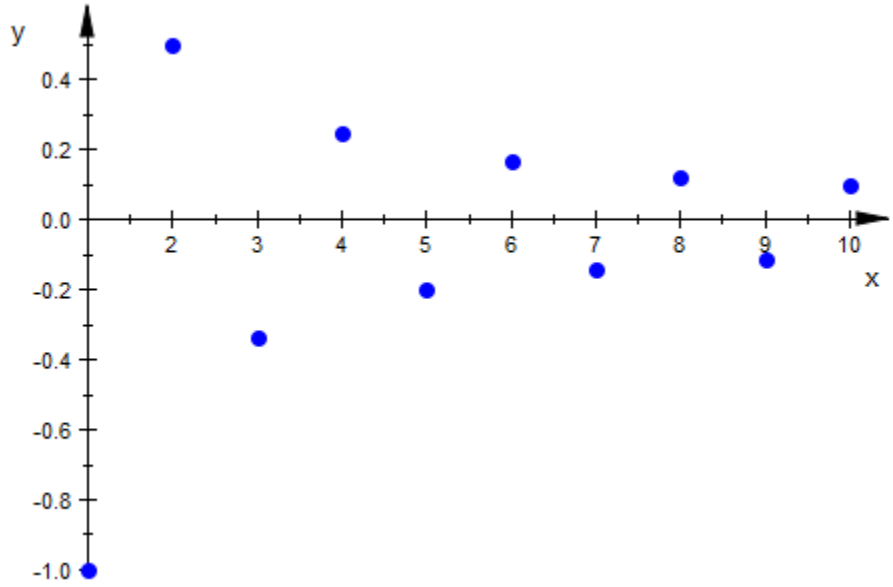
Attribute	Purpose	Default Value
YFunction	function for y values	

Examples

Example 1

When given one expression and a range, `plot::Sequence` plots the sequence in function style:

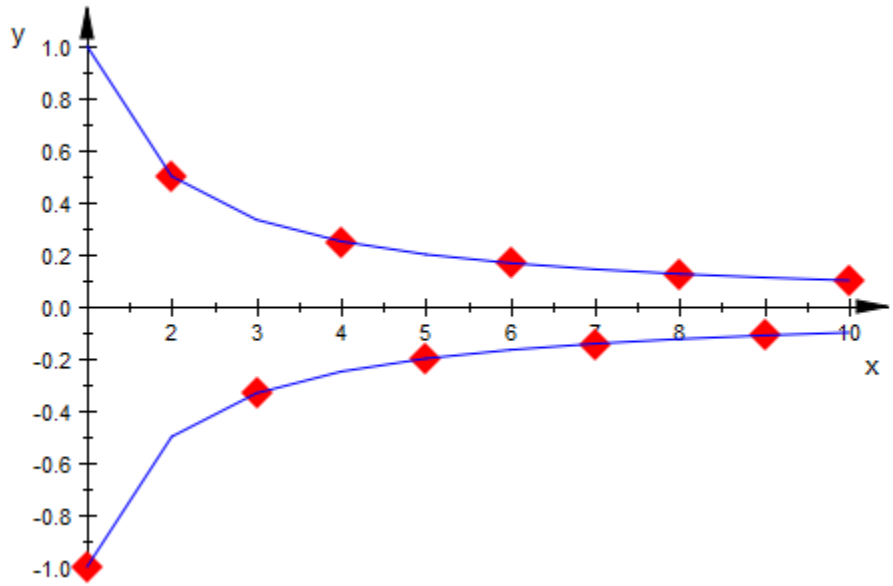
```
plot(plot::Sequence((-1)^n/n, n=1..10))
```



`plot::Sequence` accepts a variety of attributes to influence the appearance of the plot:

```
plot(plot::Sequence((-1)^n/n, n=1..10,
                    PointStyle = FilledDiamonds,
                    PointSize = 4*unit::mm,
```

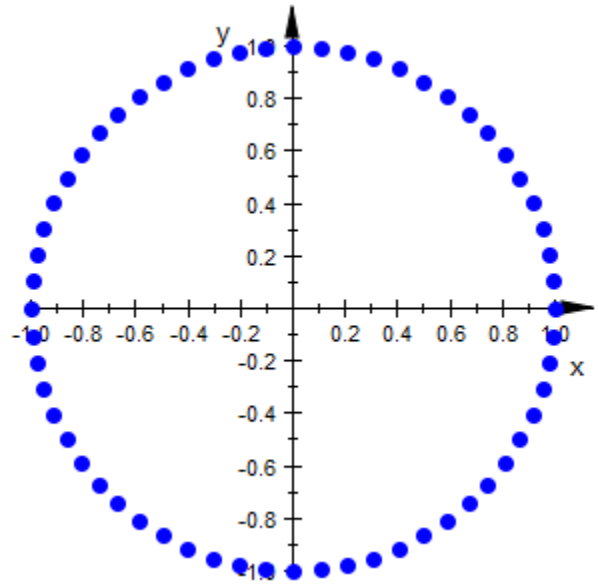
```
                Color = RGB::Red),
plot::Sequence(1/n, n=1..10,
                PointsVisible = FALSE,
                LinesVisible = TRUE),
plot::Sequence(-1/n, n=1..10,
                PointsVisible = FALSE,
                LinesVisible = TRUE))
```



Example 2

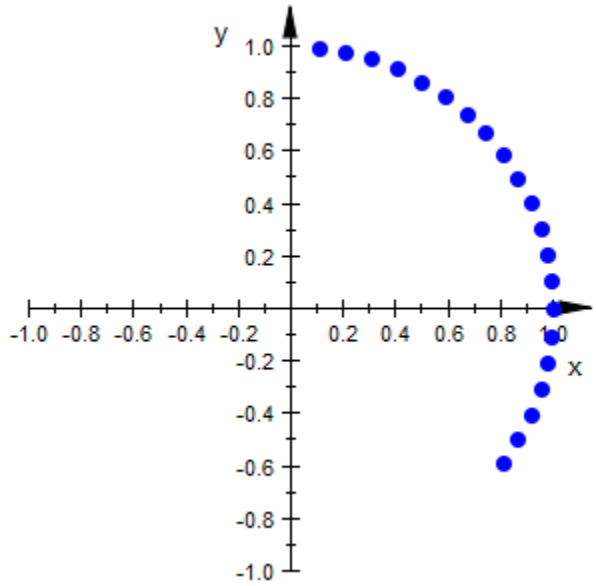
By giving two expressions, we can make `plot::Sequence` plot a sequence of points given by two expressions, for the *x*- and *y*-coordinate:

```
plot(plot::Sequence(sin(2*PI*n/60), cos(2*PI*n/60),
                n = 1..60), Scaling=Constrained)
```



In contrast to the `plot::PointList2d` call listed above as equivalent, `plot::Sequence` allows to easily animate the number of points:

```
plot(plot::Sequence(sin(2*PI*n/60), cos(2*PI*n/60),
                    n = 1..nmax, nmax = 1..60),
     Scaling=Constrained, Frames = 60, TimeRange = 1..60)
```



Example 3

By including the animation parameter in the expressions `x` and `y`, more complex animations are possible. As an example, we animate Newton iteration for different starting values. First of all, we define the iteration step which maps an approximation to its refinement:

```
newton := x -> x - f(x)/f'(x):
```

For concrete calculations, we will need to use a specific function `f`:

```
f := x -> sin(2*x) + x^2:
```

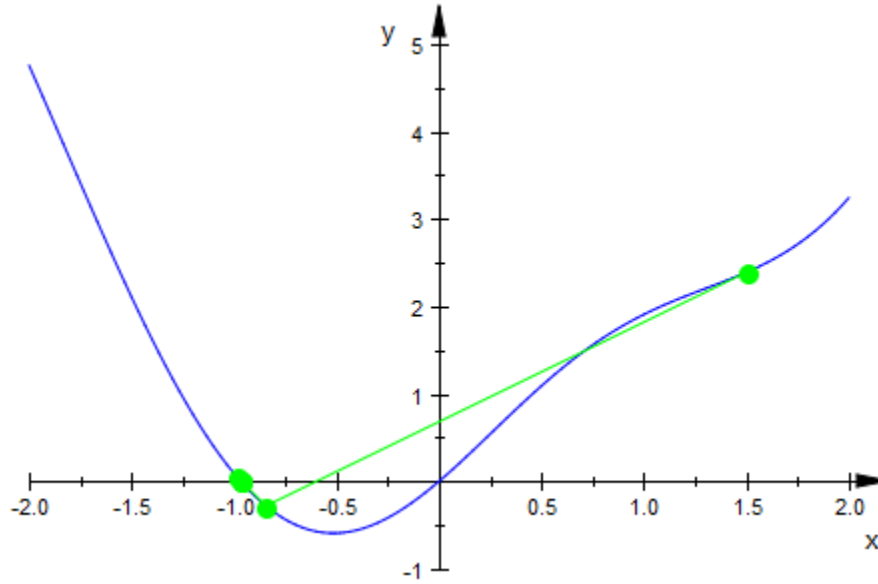
To get successive iteration steps, we will employ the function iteration operator `@@`. For example, the third improvement of the starting value `1.0` is calculated as follows:

```
(newton@@3)(1.0)
```

$$-1.064963748$$

For our animation, we want to show the approximations, the corresponding function values, and the order in which the approximations are found. Additionally, we display the function itself:

```
function := plot::Function2d(f, x = -2..2):
steps := plot::Sequence((newton@@n)(x0), f((newton@@n)(x0)),
                        n = 0..5, x0 = -1.25..1.5,
                        Color = RGB::Green,
                        LinesVisible = TRUE):
plot(function, steps,
     ViewingBox = [-2..2, -1..5], PointSize = 2.5)
```



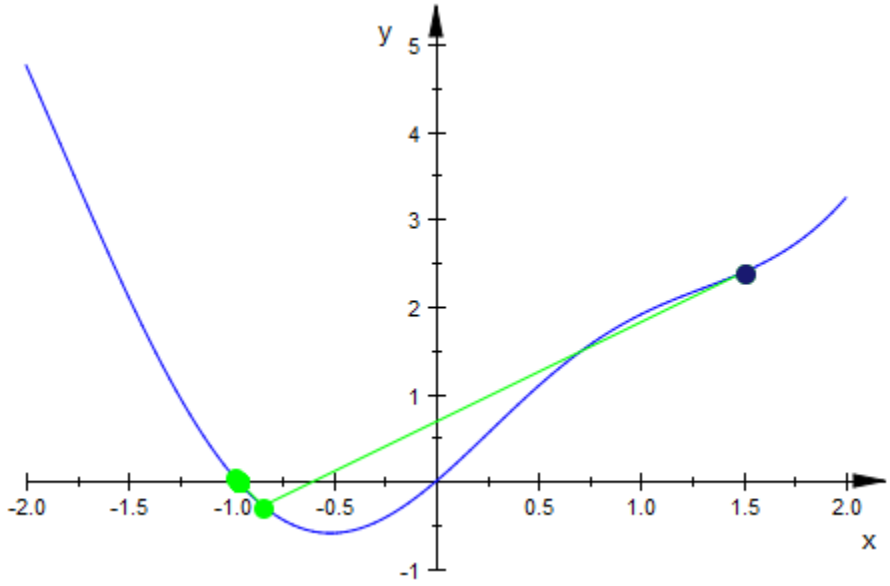
To further increase the number of iteration steps, we should reuse previously computed approximations. To this end, we use a function with `option remember`:

```
newtonIter := proc(x0, n)
                option remember;
              begin
                if domtype(n) <> DOM_INT then
                  return(procname(args()));
                end_if;
                if iszero(n) then x0
                else newton(newtonIter(x0, n-1));
```

```
                end_if;
              end_proc:
```

Additionally, we use `plot::Point2d` to display the initial point in a different color.

```
steps := plot::Sequence(newtonIter(x0, n), f(newtonIter(x0, n)),
                        n = 0..10, x0 = -1.25..1.5,
                        Color = RGB::Green,
                        LinesVisible = TRUE):
start := plot::Point2d(x0, f(x0), x0 = -1.25..1.5):
plot(function, steps, start,
     ViewingBox = [-2..2, -1..5], PointSize = 2.5)
```



Since `f` was evaluated in our object definitions, we will need to reissue the corresponding commands when changing `f`.

Parameters

x, y

Real-valued arithmetical expressions in `n` and possibly the animation parameter `a`.

x, y are equivalent to the attributes `XFunction`, `YFunction`.

`n`

The index of the sequence: an identifier or an indexed identifier.

n is equivalent to the attribute `UName`.

n_1 `..` n_2

The range of the index `n`: real-valued expressions, possibly of the animation parameter `a`.

n_1 .. n_2 is equivalent to the attributes `URange`, `UMin`, `UMax`.

`a`

Animation parameter, specified as `a = a`$_{min}$`..a`$_{max}$, where `a`$_{min}$ is the initial parameter value, and `a`$_{max}$ is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Curve2d` | `plot::Function2d` | `plot::PointList2d`

plot::SparseMatrixplot

Sparsity pattern of a matrix

Syntax

```
plot::SparseMatrixplot(A, options)
plot::SparseMatrixplot(A, x = xmin .. xmax, y = ymin .. ymax, <a = amin .. amax>, options)
plot::SparseMatrixplot([row1, row2, …], options)
plot::SparseMatrixplot([row1, row2, …], x = xmin .. xmax, y = ymin .. ymax, <a = amin .. amax
```

Description

`plot::SparseMatrixplot(A)` creates a 2D plot with the axes representing the rows and columns of the matrix A. For each nonzero entry of A a point is plotted, thus displaying sparsity patterns in the matrix.

`plot::SparseMatrixplot` interprets the indices of a matrix as x and y coordinates, respectively. The indices are ordered according to the standard orientation of the axes, i.e., low matrix indices are found in the lower left corner of the plot.

If `x = xmin .. xmax` is specified, the j-th column of an $m \times n$ matrix A corresponds to the coordinate $x = x_{\min} + \frac{(j-1)}{(n-1)} \left(x_{\max} - x_{\min}\right)$.

If `y = ymin .. ymax` is specified, the i-th row corresponds to the coordinate $y = y_{\min} + \frac{(i-1)}{(m-1)} \left(y_{\max} - y_{\min}\right)$.

If no coordinate range is specified, `xmin = 1`, `xmax = n`, and `ymin = 1`, `ymax = m` is used, i.e., the coordinate $x = j$ corresponds to the j-th column, the coordinate $y = i$ corresponds to the i-th row.

A point is plotted for each non-zero matrix entry A_{ij}.

By default, the attribute `PointColorType = Flat` is used. The color of all points is given by `PointColor`.

With PointColorType = Dichromatic, a color blend from PointColor to PointColor2 is used to indicate the size of the non-zero matrix entries. The color of points corresponding to small entries A_{ij} is PointColor. Large entries are colored with PointColor2.

Animations are triggered by specifying a range a = a_{min} .. a_{max} for a parameter a that is different from the variables x, y. Thus, in animations, both the ranges x = x_{min} .. x_{max}, y = y_{min} .. y_{max} as well as the animation range a = a_{min} .. a_{max} must be specified.

Attributes

Attribute	Purpose	Default Value
AffectViewingBox	influence of objects on the ViewingBox of a scene	TRUE
AntiAliased	antialiased lines and points?	TRUE
Color	the main color	RGB::MidnightBlue
Data	the (statistical) data to plot	
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	

Attribute	Purpose	Default Value
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.0
PointColor	the color of points	RGB::MidnightBlue
PointColor2	secondary point color for color blends	RGB::Red
PointStyle	the presentation style of points	Diamonds
PointsVisible	visibility of mesh points	TRUE
PointColorType	point coloring types	Flat
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	

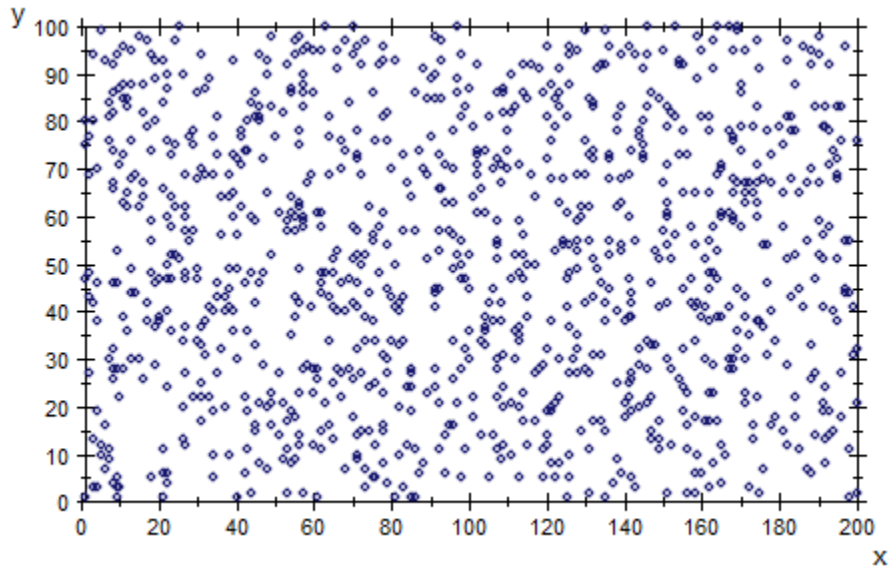
Attribute	Purpose	Default Value
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`
`XMax`	final value of parameter “x”	
`XMin`	initial value of parameter “x”	
`XName`	name of parameter “x”	
`XRange`	range of parameter “x”	
`YMax`	final value of parameter “y”	
`YMin`	initial value of parameter “y”	
`YName`	name of parameter “y”	
`YRange`	range of parameter “y”	

Examples

Example 1

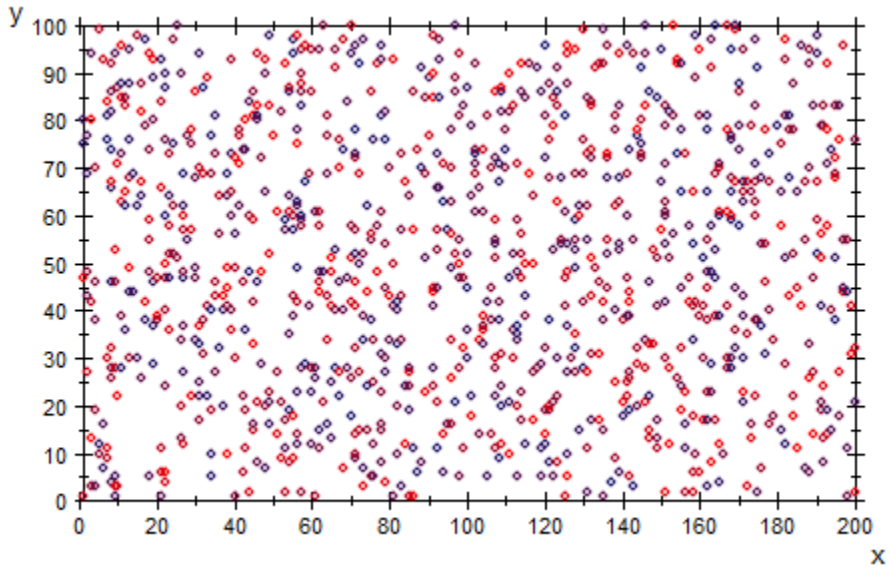
We create a random matrix of dimension 100 ×200 with 1000 nonzero entries:

```
A := matrix::random(100, 200, 1000, frandom):
plot(plot::SparseMatrixplot(A))
```



With `PointColorType` = `Dichromatic`, the color of the points indicates the size of the matrix entries:

```
plot(plot::SparseMatrixplot(A, PointColorType = Dichromatic)):
```

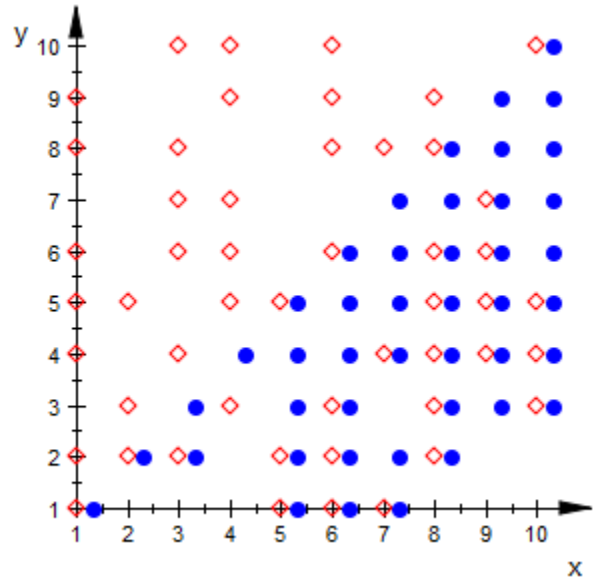



```
delete A:
```

Example 2

Choosing appropriate coordinate ranges, we let two sparse matrix plots overlap each other. The red points correspond to a sparse 10 ×10 matrix with 50 random entries. The blue points indicate the corresponding upper triangular form obtained by Gaussian elimination:

```
A := matrix::random(10, 10, 50, random(1..5)):
B := A::dom::gaussElim(A)[1]:
plot(plot::SparseMatrixplot(A, x = 1..10, y = 1..10,
                            Color = RGB::Red),
     plot::SparseMatrixplot(B, x = 1.3..10.3, y = 1..10,
                            Color = RGB::Blue,
                            PointStyle = FilledCircles),
     PointSize = 2*unit::mm, Scaling = Constrained,
     Axes = Frame)
```




```
delete A, B:
```

Parameters

A

A matrix of category `Cat::Matrix` or an array containing real numerical values or expressions of the animation parameter `a`.

A is equivalent to the attribute `Data`.

$\mathtt{row}_1$, $\mathtt{row}_2$, …

The matrix rows: each row must be a list of real numerical values or expressions of the animation parameter `a`. All rows must have the same length.

row_1, row_2, … is equivalent to the attribute `Data`.

x

Name of the horizontal coordinate: an identifier or an indexed identifier. It is used as the title of the coordinate axis in *x* direction.

x is equivalent to the attribute `XName`.

x_{min} .. x_{max}

The range of the horizontal coordinate: x_{min}, x_{max} must be numerical real value or expressions of the animation parameter *a*.

x_{min} .. x_{max} is equivalent to the attributes `XRange`, `XMin`, `XMax`.

y

Name of the vertical coordinate: an identifier or an indexed identifier. It is used as the title of the coordinate axis in *y* direction.

y is equivalent to the attribute `YName`.

y_{min} .. y_{max}

The range of the vertical coordinate: y_{min}, y_{max} must be numerical real value or expressions of the animation parameter *a*.

y_{min} .. y_{max} is equivalent to the attributes `YRange`, `YMin`, `YMax`.

a

Animation parameter, specified as $a = a_{min}..a_{max}$, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Density` | `plot::Matrixplot` | `plot::Raster`

plot::Sphere

Graphical primitive for spheres

Syntax

`plot::Sphere(r, <[`c_x`, `c_y`, `c_z`]>, <a = `a_{min}` .. `a_{max}`>, options)`

Description

`plot::Sphere(r, c)` creates a sphere of radius `r` and center `c`.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Center`	center of objects, rotation center	`[0, 0, 0]`
`CenterX`	center of objects, rotation center, x-component	`0`
`CenterY`	center of objects, rotation center, y-component	`0`
`CenterZ`	center of objects, rotation center, z-component	`0`
`Color`	the main color	`RGB::LightBlue`
`FillColor`	color of areas and surfaces	`RGB::LightBlue`
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	

Attribute	Purpose	Default Value
`LegendEntry`	add this object to the legend?	`FALSE`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`Radius`	radius of circles, spheres etc.	`1`
`Shading`	smooth color blend of surfaces	`Smooth`
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`
`Title`	object title	
`TitleFont`	font of object titles	`[" sans-serif ", 11]`
`TitlePosition`	position of object titles	
`TitleAlignment`	horizontal alignment of titles w.r.t. their coordinates	`Center`
`TitlePositionX`	position of object titles, x component	
`TitlePositionY`	position of object titles, y component	
`TitlePositionZ`	position of object titles, z component	

Attribute	Purpose	Default Value
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

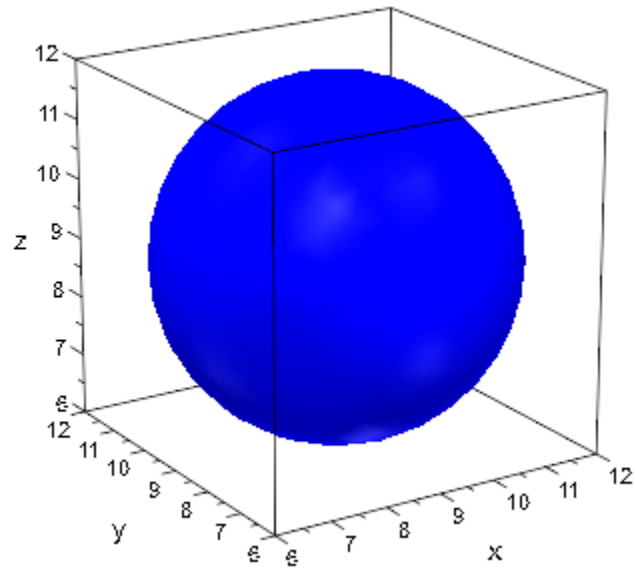
We create a blue sphere with center (9, 9, 9) and radius 3:

```
s := plot::Sphere(3, [9, 9, 9], Color = RGB::Blue)
```

```
plot::Sphere(3, [9, 9, 9])
```

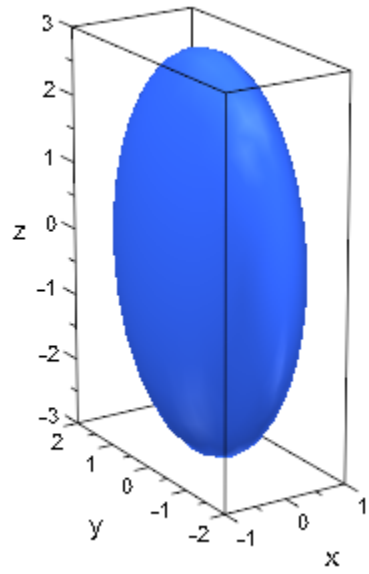
Call `plot` to plot the sphere:

```
plot(s)
```



Here is an ellipsoid around the origin with semi axes of lengths 1, 2, 3:

```
plot(plot::Ellipsoid(1, 2, 3, [0, 0, 0]))
```




```
delete s:
```

Example 2

We create a sphere with center (- 1, - 1, 5) and radius 3. At two points on the sphere, we wish to add 3D discs indicating the tangent planes.

```
c := [-1, -1, 3]:
s := plot::Sphere(3, c):
p1 := [ 1, -3, 4]:
p2 := [-3, -2, 1]:
```

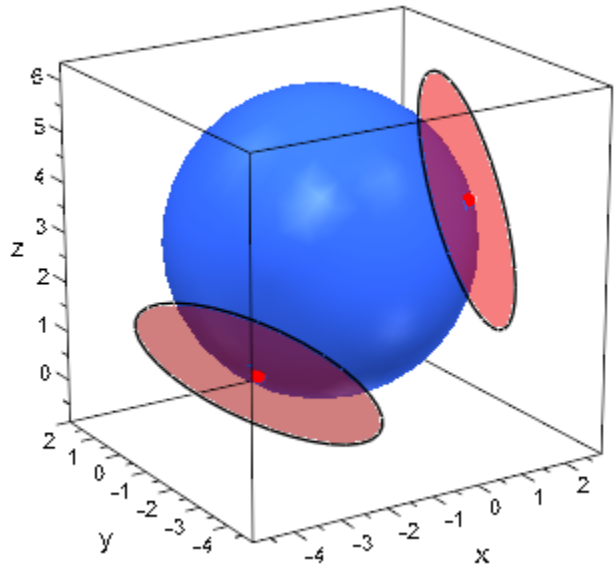
The discs are created via `plot::Circle3d` as filled 3D circles of radius 2.5, centered at the points p_1 and p_2, respectively. The normals n_i are given by p_i - c. We compute them by subracting the center c from the points p_i via `zip`:

```
n1 := zip(p1, c, _subtract):
n2 := zip(p2, c, _subtract):
t1 := plot::Circle3d(2.5, p1, n1, Filled = TRUE,
                     LineColor = RGB::Black,
                     FillColor = RGB::Red.[0.5]):
t2 := plot::Circle3d(2.5, p2, n2, Filled = TRUE,
```

```
                    LineColor = RGB::Black,
                    FillColor = RGB::Red.[0.5]):
```

Finally, we convert the points p_i to graphical points and add them to the plot:

```
p1 := plot::Point3d(p1, PointColor = RGB::Red,
                    PointSize = 2*unit::mm):
p2 := plot::Point3d(p2, PointColor = RGB::Red,
                    PointSize = 2*unit::mm):
plot(s, p1, p2, t1, t2)
```




```
delete c, s, p1, p2, n1, n2, t1, t2:
```

Example 3

We consider the same sphere as in the previous example:

```
radius := 3: center := [-1, -1, 5]:
s := plot::Sphere(radius, center):
```

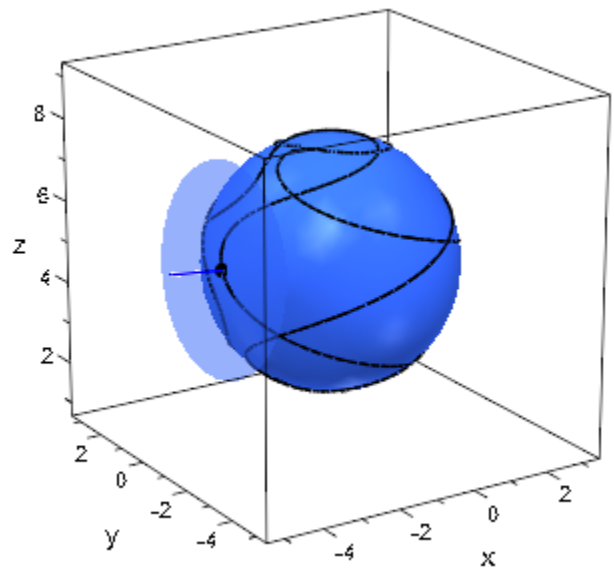
Using spherical coordinates, we define a curve on the sphere:

```
phi := a -> PI*sin(7*a):
```

```
thet := a -> PI/2 + 1.3*sin(5*a):
x := a -> center[1] + radius*cos(phi(a))*sin(thet(a)):
y := a -> center[2] + radius*sin(phi(a))*sin(thet(a)):
z := a -> center[3] + radius*cos(thet(a)):
```

The curve c is defined as an object of type `plot::Curve3d`. Further, we define an animated point p that will run along the curve. An animated filled disc of type `plot::Circle3d` indicating the tangent plane at the point p as well as the corresponding normal are added to the plot:

```
c := plot::Curve3d([x(t), y(t), z(t)], t = 0..2*PI,
                   Mesh = 1000, Color = RGB::Black):
p :=  a -> [x(a), y(a), z(a)]:
n :=  a -> zip([x(a), y(a), z(a)], center, _subtract):
d := plot::Circle3d(2.5, p(a), n(a), a = 0..2*PI, Filled = TRUE,
                    FillColor = RGB::BlueLight.[0.5],
                    LinesVisible = FALSE):
n := plot::Arrow3d(p(a), [p(a)[i] + n(a)[i]/2 $ i=1..3],
                   a = 0..2*PI, TipLength = 0.8*unit::mm):
p := plot::Point3d(p(a), a = 0..2*PI, PointColor = RGB::Black,
                   PointSize = 2*unit::mm):

plot(s, c, p, n, d, Frames = 200, TimeEnd = 50):
```



```
delete radius, center, s, phi, thet,
       x, y, z, c, p, n, d:
```

Parameters

r

The radius of the sphere: a real numerical value or an arithmetical expression of the animation parameter `a`.

r is equivalent to the attribute `Radius`.

c_x, c_y, c_z

The coordinates of the center: real numerical values or arithmetical expressions of the animation parameter `a`. If no center is specified, a sphere/ellipsoid centered at the orign is created.

c_x, c_y, c_z are equivalent to the attributes `Center`, `CenterX`, `CenterY`, `CenterZ`.

a

Animation parameter, specified as a = a_{min}..a_{max}, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Circle3d` | `plot::Ellipsoid` | `plot::Surface`

plot::Ellipsoid

Graphical primitive for ellipsoids

Syntax

```
plot::Ellipsoid(r_x, r_y, r_z, <[c_x, c_y, c_z]>, <a = a_min .. a_max>, options)
```

Description

`plot::Ellipsoid(`r_x`, `r_y`, `r_z`, c)` creates an ellipsoid with the center `c` and symmetry axes parallel to the coordinate axes. The semi axes have the lengths r_x, r_y, r_z.

Ellipsoids with arbitrary orientations of the symmetry axes can be generated via `plot::Rotate3d`.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Center`	center of objects, rotation center	`[0, 0, 0]`
`CenterX`	center of objects, rotation center, x-component	`0`
`CenterY`	center of objects, rotation center, y-component	`0`
`CenterZ`	center of objects, rotation center, z-component	`0`
`Color`	the main color	`RGB::LightBlue`
`FillColor`	color of areas and surfaces	`RGB::LightBlue`
`Frames`	the number of frames in an animation	`50`

Attribute	Purpose	Default Value
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
SemiAxes	semi axes of ellipses and ellipsods	[1, 2, 3]
SemiAxisX	first semi axis of ellipses and ellipsods	1
SemiAxisY	second semi axis of ellipses and ellipsods	2
SemiAxisZ	third semi axis of ellipsods	3
Shading	smooth color blend of surfaces	Smooth
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	

Attribute	Purpose	Default Value
`TitleAlignment`	horizontal alignment of titles w.r.t. their coordinates	`Center`
`TitlePositionX`	position of object titles, x component	
`TitlePositionY`	position of object titles, y component	
`TitlePositionZ`	position of object titles, z component	
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

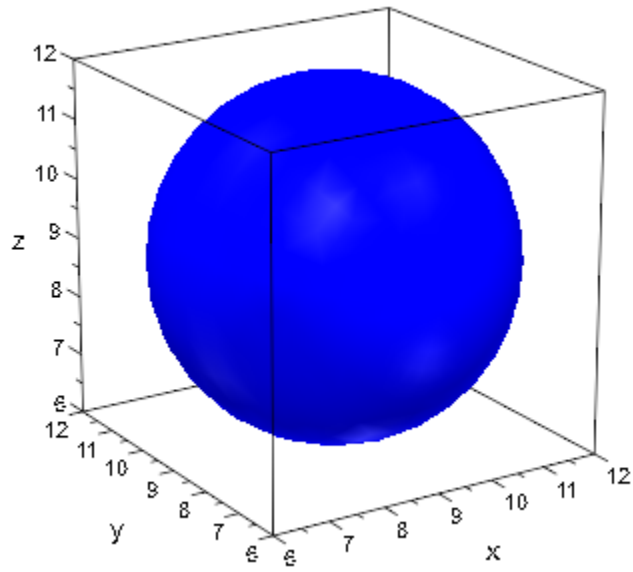
We create a blue sphere with center (9, 9, 9) and radius 3:

```
s := plot::Sphere(3, [9, 9, 9], Color = RGB::Blue)
```

```
plot::Sphere(3, [9, 9, 9])
```

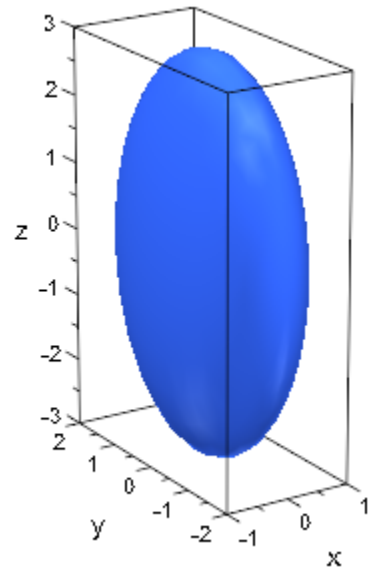
Call `plot` to plot the sphere:

```
plot(s)
```



Here is an ellipsoid around the origin with semi axes of lengths 1, 2, 3:

```
plot(plot::Ellipsoid(1, 2, 3, [0, 0, 0]))
```




```
delete s:
```

Example 2

We create a sphere with center (- 1, - 1, 5) and radius 3. At two points on the sphere, we wish to add 3D discs indicating the tangent planes.

```
c := [-1, -1, 3]:
s := plot::Sphere(3, c):
p1 := [ 1, -3, 4]:
p2 := [-3, -2, 1]:
```

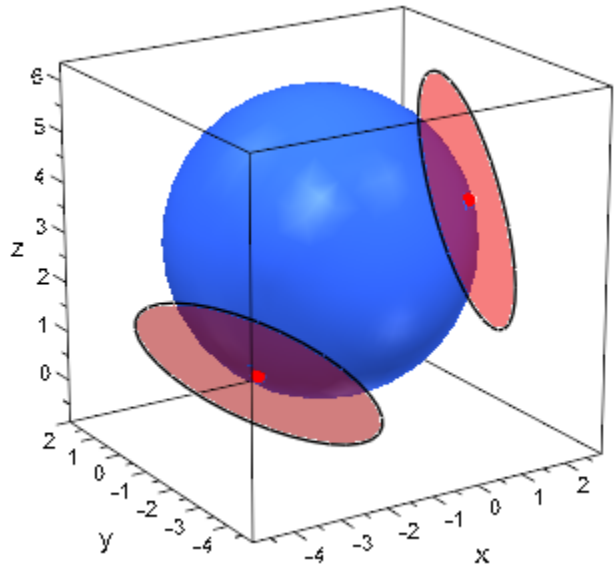
The discs are created via `plot::Circle3d` as filled 3D circles of radius 2.5, centered at the points p_1 and p_2, respectively. The normals n_i are given by p_i - c. We compute them by subracting the center c from the points p_i via `zip`:

```
n1 := zip(p1, c, _subtract):
n2 := zip(p2, c, _subtract):
t1 := plot::Circle3d(2.5, p1, n1, Filled = TRUE,
                     LineColor = RGB::Black,
                     FillColor = RGB::Red.[0.5]):
t2 := plot::Circle3d(2.5, p2, n2, Filled = TRUE,
```

```
                    LineColor = RGB::Black,
                    FillColor = RGB::Red.[0.5]):
```

Finally, we convert the points p_i to graphical points and add them to the plot:

```
p1 := plot::Point3d(p1, PointColor = RGB::Red,
                    PointSize = 2*unit::mm):
p2 := plot::Point3d(p2, PointColor = RGB::Red,
                    PointSize = 2*unit::mm):
plot(s, p1, p2, t1, t2)
```




```
delete c, s, p1, p2, n1, n2, t1, t2:
```

Example 3

We consider the same sphere as in the previous example:

```
radius := 3: center := [-1, -1, 5]:
s := plot::Sphere(radius, center):
```

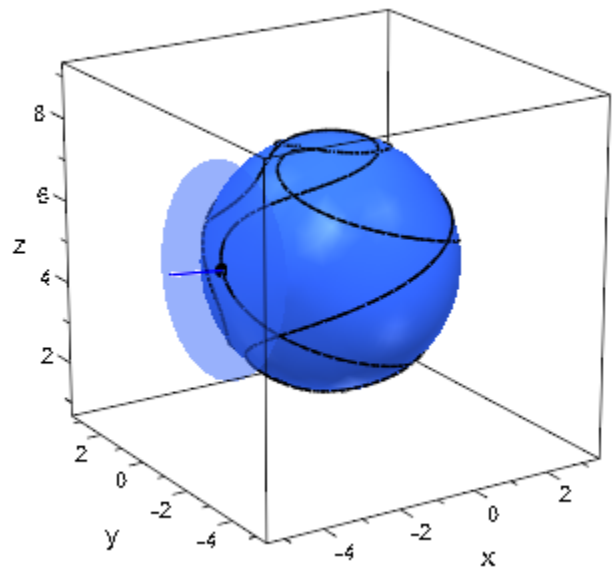
Using spherical coordinates, we define a curve on the sphere:

```
phi := a -> PI*sin(7*a):
```

```
thet := a -> PI/2 + 1.3*sin(5*a):
x := a -> center[1] + radius*cos(phi(a))*sin(thet(a)):
y := a -> center[2] + radius*sin(phi(a))*sin(thet(a)):
z := a -> center[3] + radius*cos(thet(a)):
```

The curve c is defined as an object of type `plot::Curve3d`. Further, we define an animated point p that will run along the curve. An animated filled disc of type `plot::Circle3d` indicating the tangent plane at the point p as well as the corresponding normal are added to the plot:

```
c := plot::Curve3d([x(t), y(t), z(t)], t = 0..2*PI,
                   Mesh = 1000, Color = RGB::Black):
p :=  a -> [x(a), y(a), z(a)]:
n :=  a -> zip([x(a), y(a), z(a)], center, _subtract):
d := plot::Circle3d(2.5, p(a), n(a), a = 0..2*PI, Filled = TRUE,
                    FillColor = RGB::BlueLight.[0.5],
                    LinesVisible = FALSE):
n := plot::Arrow3d(p(a), [p(a)[i] + n(a)[i]/2 $ i=1..3],
                   a = 0..2*PI, TipLength = 0.8*unit::mm):
p := plot::Point3d(p(a), a = 0..2*PI, PointColor = RGB::Black,
                   PointSize = 2*unit::mm):

plot(s, c, p, n, d, Frames = 200, TimeEnd = 50):
```



```
delete radius, center, s, phi, thet,
       x, y, z, c, p, n, d:
```

Parameters

r_x, r_y, r_z

The length of the semi axes of the ellipsoid: real numerical values or arithmetical expressions of the animation parameter `a`.

r_x, r_y, r_z are equivalent to the attributes `SemiAxes`, `SemiAxisX`, `SemiAxisY`, `SemiAxisZ`.

c_x, c_y, c_z

The coordinates of the center: real numerical values or arithmetical expressions of the animation parameter `a`. If no center is specified, a sphere/ellipsoid centered at the orign is created.

c_x, c_y, c_z are equivalent to the attributes `Center`, `CenterX`, `CenterY`, `CenterZ`.

a

Animation parameter, specified as a `= a`$_{min}$`..a`$_{max}$, where `a`$_{min}$ is the initial parameter value, and `a`$_{max}$ is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Circle3d` | `plot::Sphere` | `plot::Surface`

plot::Spherical

Surfaces in 3D parameterized in spherical coordinates

Syntax

```
plot::Spherical([r, ϕ, θ], u = u_min .. u_max, v = v_min .. v_max, <a = a_min .. a_max>, options)
```

Description

`plot::Spherical` creates surfaces parametrized in spherical coordinates.

The surface given by a mapping ("parametrization") $(u, v) \to (r(u, v), \phi(u, v), \theta(u, v))$ is the set of all image points

$$\left\{ \begin{pmatrix} r(u, v) \\ \phi(u, v) \\ \theta(u, v) \end{pmatrix} \;\middle|\; u \in [u_{min}, u_{max}], v \in [v_{min}, v_{max}] \right\}$$

in spherical coordinates, which translate to the usual "Cartesian" coordinates as

$$\begin{aligned} x &= r \cos(\phi) \sin(\theta) \\ y &= r \sin(\phi) \sin(\theta) \\ z &= r \cos(\theta) \end{aligned}$$

.

r is referred to as "radius", ϕ as "azimuthal angle", and θ is known as "polar angle."

The functions r, ϕ, θ are evaluated on a regular equidistant mesh of sample points in the u-v plane. This mesh is determined by the attributes `UMesh`, `VMesh`. By default, the attribute `AdaptiveMesh = 0` is set, i.e., no adaptive refinement of the equidistant mesh is used.

If the standard mesh does not suffice to produce a sufficiently detailed plot, one may either increase the value of `UMesh`, `VMesh` or `USubmesh`, `VSubmesh`, or set `AdaptiveMesh = n` with some (small) positive integer `n`. If necessary, up to 2^n - 1

additional points are placed in each direction of the u-v plane between adjacent points of the initial equidistant mesh. Cf. "Example 3" on page 24-843.

"Coordinate lines" ("parameter lines") are curves on the surface.

The phrase "ULines" refers to the curves $(r(u, v_0), \phi(u, v_0), \theta(u, v_0))$ with the parameter u running from u_{min} to u_{max}, while v_0 is some fixed value from the interval $[v_{min}, v_{max}]$.

The phrase "VLines" refers to the curves $(r(u_0, v), \phi(u_0, v), \theta(u_0, v))$ with the parameter v running from v_{min} to v_{max}, while u_0 is some fixed value from the interval $[u_{min}, u_{max}]$.

By default, the parameter curves are visible. They may be switched off by specifying `ULinesVisible = FALSE` and `VLinesVisible = FALSE`, respectively.

The coordinate lines controlled by `ULinesVisible = TRUE/FALSE` and `VLinesVisible = TRUE/FALSE` indicate the equidistant mesh in the u-v plane set via the `UMesh`, `VMesh` attributes. If the mesh is refined by the `USubmesh`, `VSubmesh` attributes, or by the adaptive mechanism controlled by `AdaptiveMesh = n`, no additional parameter lines are drawn.

As far as the numerical approximation of the surface is concerned, the settings

`UMesh =` n_u, `VMesh =` n_v, `USubmesh =` m_u, `VSubmesh =` m_v

and

`UMesh = (`n_u `- 1) (`m_u `+ 1) + 1`, `VMesh = (`n_v `- 1) (`m_v `+ 1) + 1`,

`USubmesh = 0`, `VSubmesh = 0`

are equivalent. However, in the first setting, `nu` parameter lines are visible in the u direction, while in the latter setting `(nu - 1)*(mu + 1) + 1` parameter lines are visible. Cf. "Example 3" on page 24-843.

Use `Filled = FALSE` to obtain a wireframe representation of the surface.

If the expression/function r contains singularities, it is recommended (but not strictly necessary) to use the attribute `ViewingBox` to set a suitable viewing box. No such precautions are necessary for ϕ and θ, although singularities in these functions may result in poorly rendered surfaces – in many cases setting the attributes `Mesh` and/or `AdaptiveMesh` to higher values will help. Cf. "Example 6" on page 24-849.

Attributes

Attribute	Purpose	Default Value
AdaptiveMesh	adaptive sampling	0
AffectViewingBox	influence of objects on the ViewingBox of a scene	TRUE
Color	the main color	RGB::Red
Filled	filled or transparent areas and surfaces	TRUE
FillColor	color of areas and surfaces	RGB::Red
FillColor2	second color of areas and surfaces for color blends	RGB::CornflowerBlue
FillColorType	surface filling types	Dichromatic
FillColorFunction	functional area/surface coloring	
FillColorDirection	the direction of color transitions on surfaces	[0, 0, 1]
FillColorDirectionX	x-component of the direction of color transitions on surfaces	0
FillColorDirectionY	y-component of the direction of color transitions on surfaces	0
FillColorDirectionZ	z-component of the direction of color transitions on surfaces	1
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	TRUE

Attribute	Purpose	Default Value
LineColor	color of lines	RGB::Black.[0.25]
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	0
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Mesh	number of sample points	[25, 25]
MeshVisible	visibility of irregular mesh lines in 3D	FALSE
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles

Attribute	Purpose	Default Value
PointsVisible	visibility of mesh points	FALSE
Submesh	density of submesh (additional sample points)	[0, 0]
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
ULinesVisible	visibility of parameter lines (u lines)	TRUE
UMax	final value of parameter “u”	
UMesh	number of sample points for parameter “u”	25
UMin	initial value of parameter “u”	
UName	name of parameter “u”	
URange	range of parameter “u”	
USubmesh	density of additional sample points for parameter “u”	0
VLinesVisible	visibility of parameter lines (v lines)	TRUE

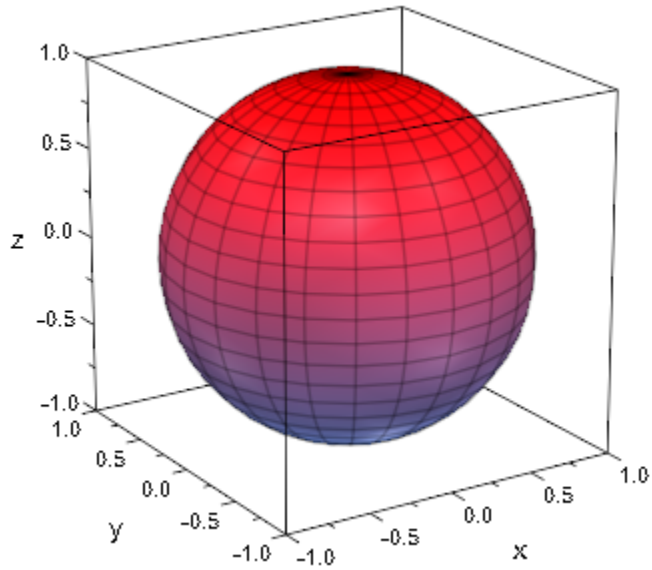
Attribute	Purpose	Default Value
VMax	final value of parameter “v”	
VMesh	number of sample points for parameter “v”	25
VMin	initial value of parameter “v”	
VName	name of parameter “v”	
VRange	range of parameter “v”	
VSubmesh	density of additional sample points for parameter “v”	0
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XContours	contour lines at constant x values	[]
XFunction	function for x values	
YContours	contour lines at constant y values	[]
YFunction	function for y values	
ZContours	contour lines at constant z values	[]
ZFunction	function for z values	

Examples

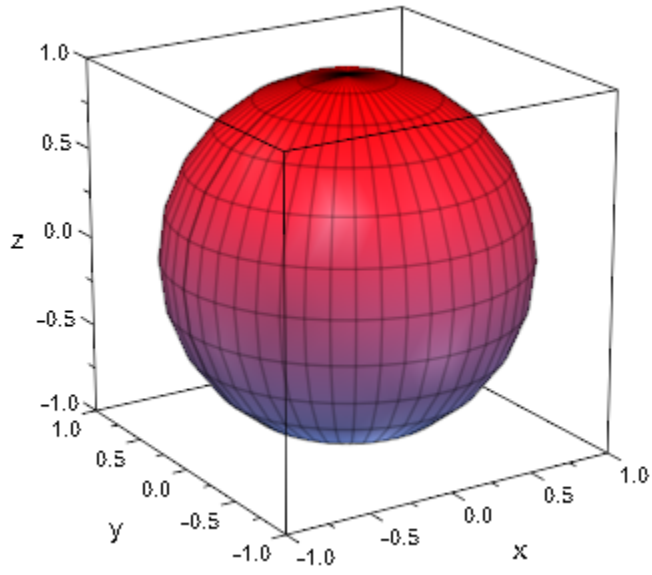
Example 1

Spherical coordinates get their name from the fact that, with a constant radius, the parameterize a sphere:

```
plot(plot::Spherical([1, u, v], u = 0..2*PI, v = 0..PI))
```



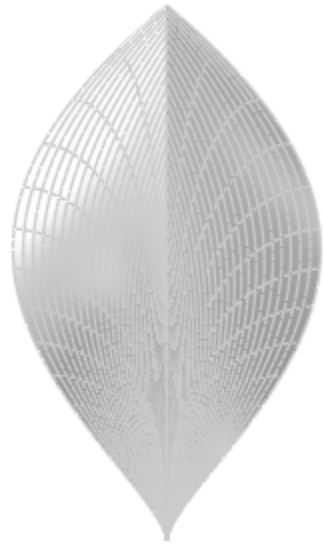

```
plot(plot::Spherical([1, u, v], u = 0..PI, v = 0..2*PI))
```



Example 2

The following plot demonstrates that spherical plots can exhibit singular surface features even with differentiable parameterizations; in this case, the rim in the middle is actually a border of both the left- and the right-hand part:

```
plot(plot::Spherical(
       [(phi^2*thet), phi, thet^2],
       phi = -PI..PI, thet=0..0.25*PI,
       Mesh = [40,40], Submesh=[3,0],
       Color = [0.9$3], FillColorType=Flat, LineColor=[0.8$3]),
     Axes = None, CameraDirection = [1, 0, 0])
```

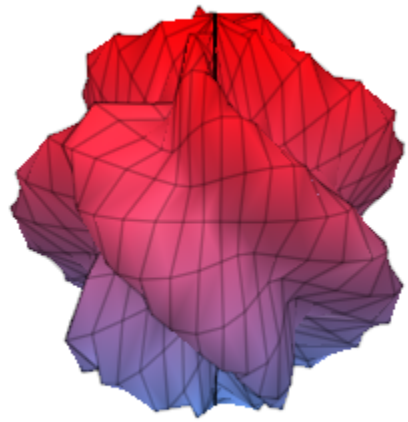


Example 3

For oscillating parameterizations or other surfaces with fine details, the default mesh may be too coarse. As stated above, the three attributes `Mesh`, `Submesh`, and `AdaptiveMesh` can be used for improving plots of these objects.

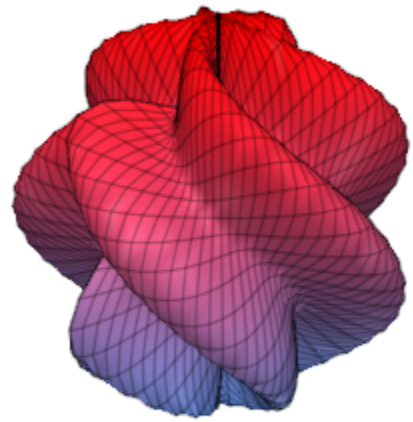
First, note that the following plot is not rendered with a sufficient resolution:

```
surf := plot::Spherical([4+sin(5*(u+v)), u, v], u = 0..PI, v = 0..2*PI):
plot(surf, Axes = None)
```



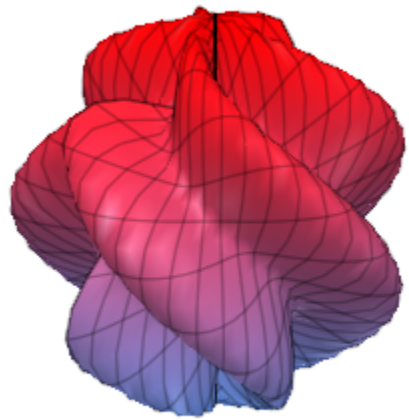
Setting `Mesh` to twice its default, we get a smoother surface with additional parameter lines:

```
surf::Mesh := [50, 50]:
plot(surf, Axes = None)
```



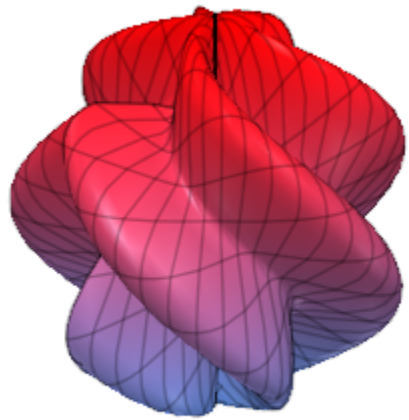
Almost the same effect, but without the additional parameter lines, can be achieved by setting `Submesh = [1, 1]`:

```
delete surf::Mesh:
surf::Submesh := [1, 1]:
plot(surf, Axes = None)
```



It is also possible to use adaptive mesh refinement in areas where neighboring patches have an angle of more than 10 degrees. While this option is mostly useful for surfaces which require refinement only in some parts, it is certainly feasible with a plot like this, too (but increasing `Submesh` is faster):

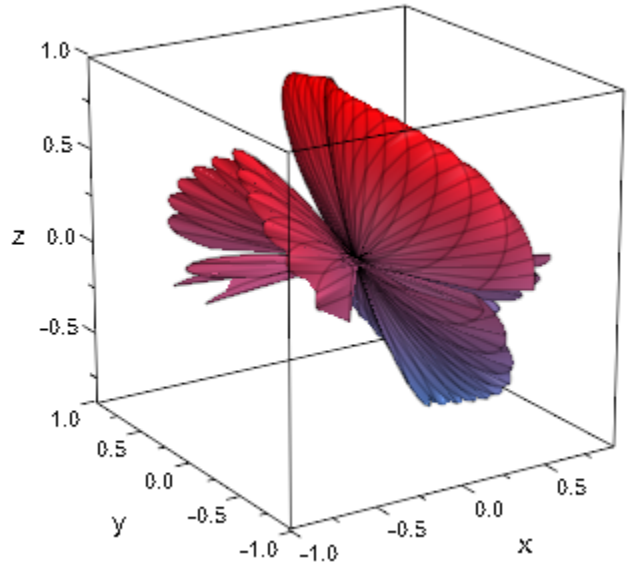
```
delete surf::Submesh:
surf::AdaptiveMesh := 2:
plot(surf, Axes = None)
```



Example 4

The radius function *r* may also take on negative values. With radius functions of changing sign, spherical surfaces often do self-intersect:

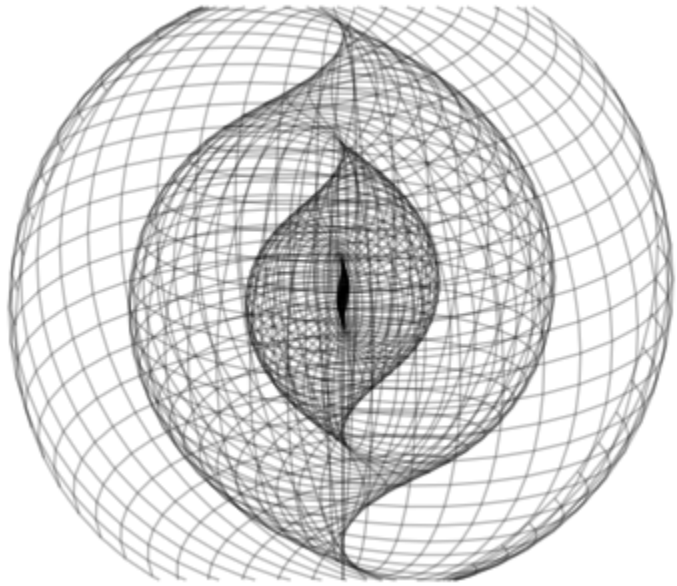
```
plot(plot::Spherical(
       [sin(phi^2*thet), phi, thet],
         phi = -PI..PI, thet = 0..0.5*PI,
       Mesh = [40, 20], Submesh=[0, 3]))
```



Example 5

The angular functions (ϕ and θ) are not limited in value:

```
plot(plot::Spherical([r, r, thet], r = 0..9, thet = -PI..PI,
                     Mesh = [60, 60], Filled = FALSE),
     Axes = None,
     plot::Camera([100, 100, 50], [0,0,0], 0.1))
```

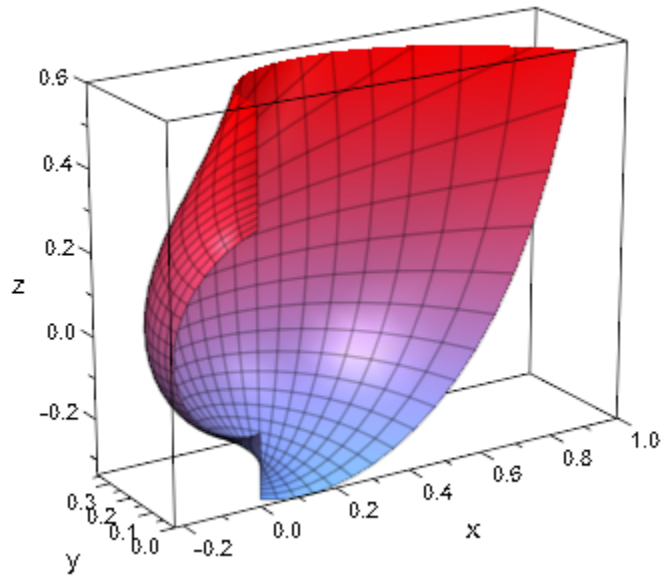


Note that we used an explicit `plot::Camera` object here because the automatic camera is always placed such that all of an object is visible, even when using `CameraDirection`. To get a "closer" look, use the interactive manipulation possibilities or an explicit camera.

Example 6

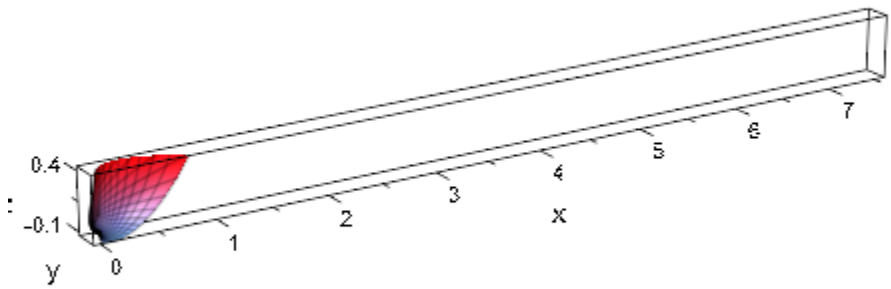
Singularities in the radius function are heuristically handled:

```
plot(plot::Spherical([1/(u + v), u, v], u = 0..PI, v = 0..PI))
```



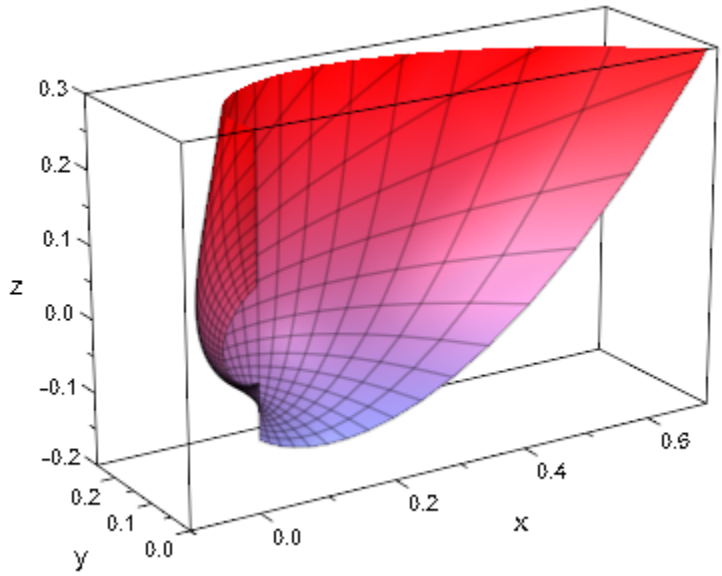
However, the heuristics fails for some examples:

```
plot(plot::Spherical([1/(u + v)^2, u, v], u = 0..PI, v = 0..PI))
```



In cases like this, we recommend setting a viewing box explicitly with the attribute `ViewingBox`:

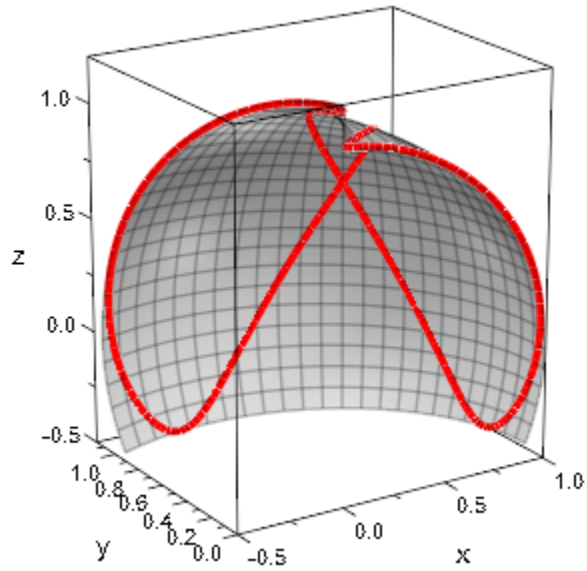
```
plot(plot::Spherical([1/(u + v)^2, u, v], u = 0..PI, v = 0..PI),
     ViewingBox = [-1/10..0.7, 0..1/4, -0.2..0.3])
```



Example 7

By setting one of the parameter ranges to a degenerate interval, it is possible to draw curves on a spherical surface:

```
f := (u, v) -> [1 + u/10, u, v]:
surface := plot::Spherical(f(u,v), u = 0..2, v = 0..2,
                           FillColor = RGB::Grey, FillColorType = Flat):
curve := plot::Spherical(f((1 + sin(u)), (1 + sin(2*u))),
                         u = 0..2*PI, v = 0..0, Mesh = [200, 1],
                         LineColor = RGB::Red, LineWidth = 1):
plot(surface, curve)
```



Example 8

While the transformation from spherical to Cartesian coordinates is not invertible, there are at least two ways of expressing each Cartesian point in spherical coordinates and any surface parameterizable in Cartesian coordinates can also be plotted using `plot::Spherical`:

```
trans := linalg::ogCoordTab[Spherical[RightHanded],
                            InverseTransformation]:
spher := trans(x, y, sin(x^2+y^2))
```

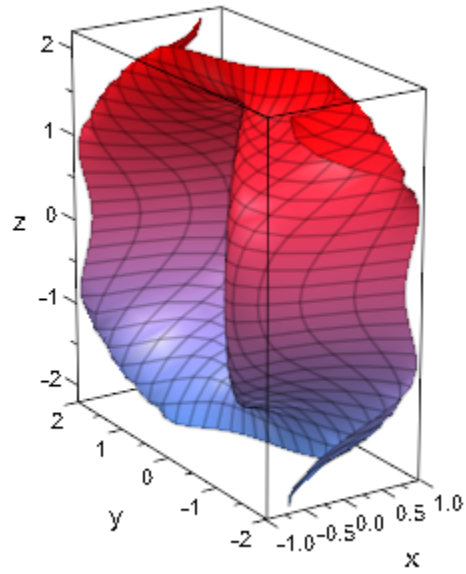
$$\left[\sigma_1,\ \arccos\left(\frac{\sin\left(x^2+y^2\right)}{\sigma_1}\right),\ \arccos\left(\frac{x}{\sqrt{x^2+y^2}}\right)+\operatorname{sign}(y)\ (\operatorname{sign}(y)-1)\right.$$

$$\left.\left(\pi-\arccos\left(\frac{x}{\sqrt{x^2+y^2}}\right)\right)\right]$$

where

$$\sigma_1=\sqrt{x^2+y^2+\sin(x^2+y^2)^2}$$

```
plot(plot::Spherical(spher, x = -2..2, y = -2..2))
```

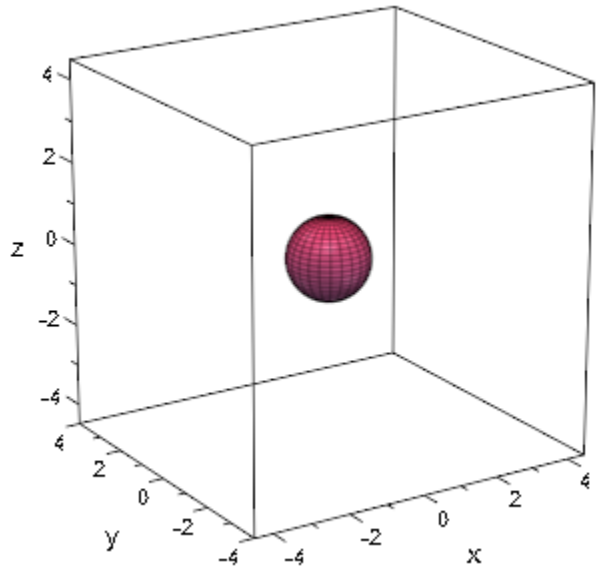


Example 9

Last but not least we can also produce animations with the help of `plot::Spherical`. The following shows a deformation from a general spherical object to a sphere. We have

used the animation parameter `a` inside of the argument for the sine function to obtain a slight rotation during the deformation process:

```
plot(
  plot::Spherical(
   [1+a*sin(3*Phi+a)*sin(2*Theta),Phi,Theta],
   Theta=0..PI, Phi=0..2*PI, a=5..0
  )
)
```



Parameters

r, ϕ, θ

The coordinate functions: arithmetical expressions or `piecewise` objects depending on the surface parameters *u*, *v* and the animation parameter `a`. Alternatively, procedures that accept 2 input parameters *u*, *v* or 3 input parameters *u*, *v*, *a* and return a real numerical value when the input parameters are numerical.

r, ϕ, θ are equivalent to the attributes `XFunction`, `YFunction`, `ZFunction`.

u

The first surface parameter: an identifier or an indexed identifier.

u is equivalent to the attribute `UName`.

u_{min} .. u_{max}

The plot range for the parameter u: u_{min}, u_{max} must be numerical real values or expressions of the animation parameter a.

u_{min} .. u_{max} is equivalent to the attributes `URange`, `UMin`, `UMax`.

v

The second surface parameter: an identifier or an indexed identifier.

v is equivalent to the attribute `VName`.

v_{min} .. v_{max}

The plot range for the parameter v: v_{min}, v_{max} must be numerical real values or expressions of the animation parameter a.

v_{min} .. v_{max} is equivalent to the attributes `VRange`, `VMin`, `VMax`.

a

Animation parameter, specified as a = a_{min}..a_{max}, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions

`linalg::ogCoordTab` | `plot` | `plot::copy`

MuPAD Graphical Primitives

`plot::Cylindrical` | `plot::Polar` | `plot::Surface`

plot::Streamlines2d

Streamlines of vector fields

Syntax

```
plot::Streamlines2d([v1, v2], x = xmin .. xmax, y = ymin .. ymax, <a = amin .. amax>, options
```

```
plot::Streamlines2d(v1, v2, x = xmin .. xmax, y = ymin .. ymax, <a = amin .. amax>, options)
```

```
plot::Streamlines2d(V, x = xmin .. xmax, y = ymin .. ymax, <a = amin .. amax>, options)
```

Description

`plot::Streamlines2d( [v1, v2] , x = `x_{min}`..`x_{max}` , y = `y_{min}`..`y_{max}` )` creates streamlines of the vector field defined by $(x, y) \to (v_1(x, y), v_2(x, y))$ with $(x, y) \in [x_{min}, x_{max}] \times [y_{min}, y_{max}]$.

A vector field is defined by a function $f: \mathbb{R}^2 \to \mathbb{R}^2$. `plot::Streamlines2d` displays a vector field by drawing almost evenly spaced streamlines of the vector field, i.e., curves to which the vector field is tangential at every point. The density of stream lines (and the time needed for calculation) is controlled with the attribute `MinimumDistance`.

As a rule of thumb: decreasing the value of `MinimumDistance` by a factor of 2 leads to an increase of the runtime by a factor of 4.

A user defined color scheme may be specified by `LineColorFunction = color`, where `color` is a MuPAD procedure accepting 6 input parameters and returning a list of `RGB` values. During plotting, this function is called in the form `color(x, y, v1, v2, t, l, n)`:

The values `x`, `y` are the coordinates of the current point.

The values `v1`, `v2` are the components of the vector field at the current point.

The value `t` is the "time" of the current point `(x, y)` on the current streamline. The scaling of this parameter depends on the vector field.

The value `l` is the curve length of the current streamline from its starting point the current point `(x, y)`, as a Euclidean distance. This parameter is invariant with respect to scalar changes of the vector field (up to changing the direction of the streamline).

The integer value `n` is a count of the current streamline. Each separate streamline has a different value.

Cf. “Example 3” on page 24-865.

Attributes

Attribute	Purpose	Default Value
`AbsoluteError`	maximal absolute discretization error	
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`LineColor`	color of lines	`RGB::Black`
`LineWidth`	width of lines	`0.35*unit::mm`
`LineColor2`	color of lines	`RGB::DeepPink`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LineColorType`	line coloring types	`Flat`
`LineColorFunction`	functional line coloring	
`MinimumDistance`	space between stream lines	
`Name`	the name of a plot object (for browser and legend)	

Attribute	Purpose	Default Value
ODEMethod	the numerical scheme used for solving the ODE	ABM4
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
RelativeError	maximal relative discretization error	1/100000
Stepsize	set a constant step size	
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
TipAngle	opening angle of arrow heads	(2*PI)/15
TipStyle	presentation style of arrow heads	Filled
TipLength	length of arrow heads	0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	

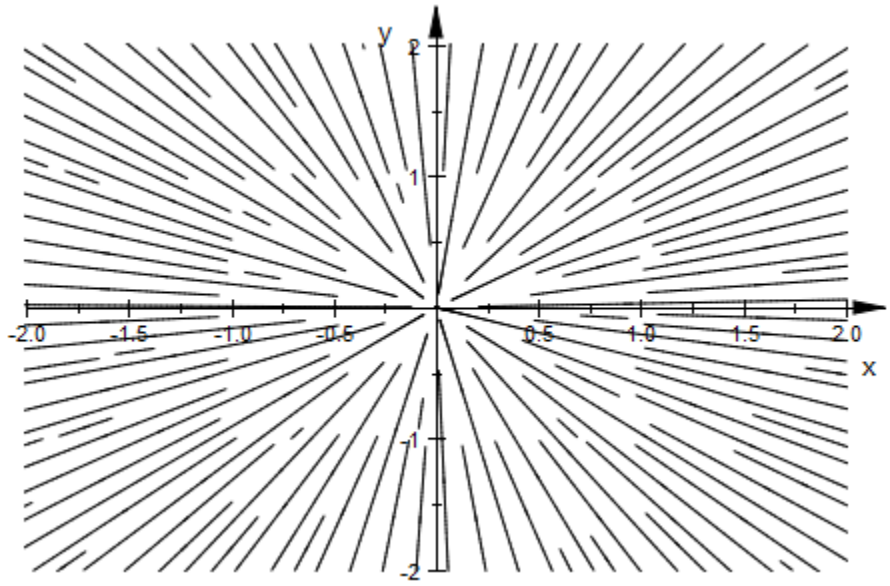
Attribute	Purpose	Default Value
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`
`XFunction`	function for x values	
`XMax`	final value of parameter “x”	
`XMin`	initial value of parameter “x”	
`XName`	name of parameter “x”	
`XRange`	range of parameter “x”	
`YFunction`	function for y values	
`YMax`	final value of parameter “y”	
`YMin`	initial value of parameter “y”	
`YName`	name of parameter “y”	
`YRange`	range of parameter “y”	

Examples

Example 1

`plot::Streamlines2d` depicts vector fields by (more or less) equidistant stream lines:

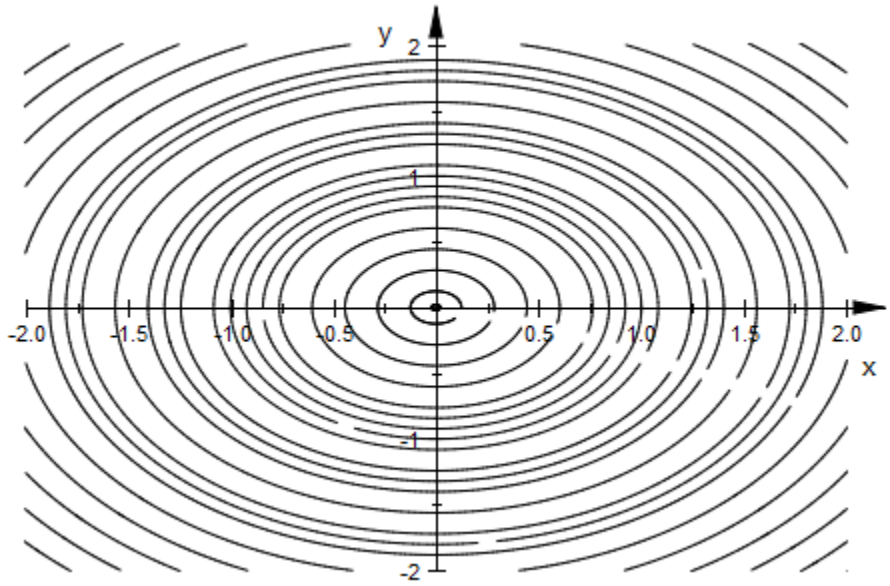
```
plot(plot::Streamlines2d(-x, -y, x=-2..2, y=-2..2))
```



Note that this style of display necessarily breaks symmetries, in this case the perfect rotational symmetry of the vector field.

Additionally, cycles will not be closed, but leave a gap:

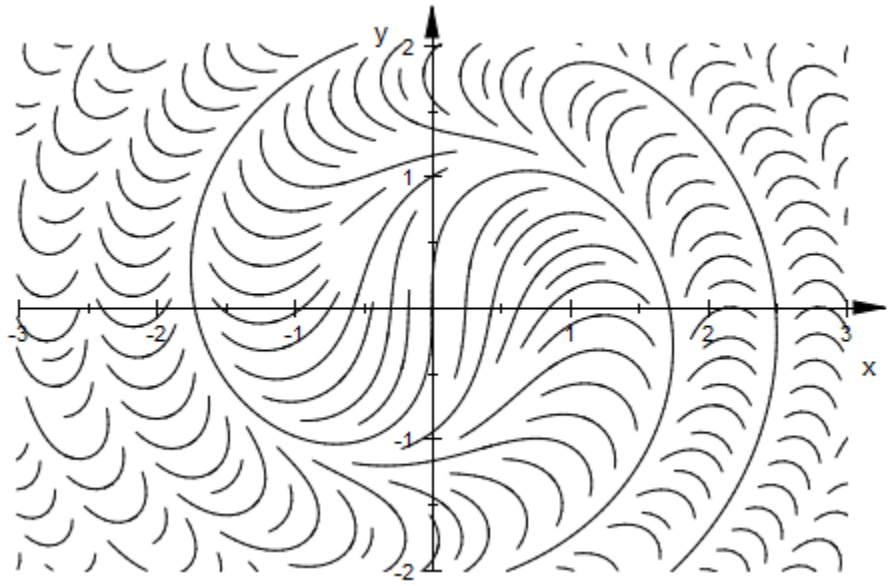
```
plot(plot::Streamlines2d(-y, x, x=-2..2, y=-2..2))
```



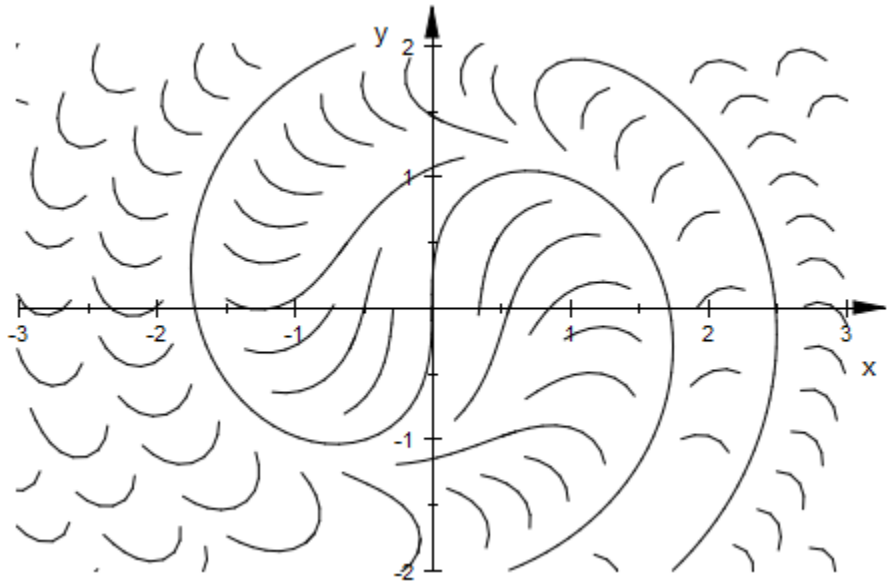
Example 2

Apart from the "usual" parameters such as parameter ranges, line color, or line width, `plot::Streamlines2d` can be controlled with the attribute `MinimumDistance`, which sets the minimum distance between stream lines:

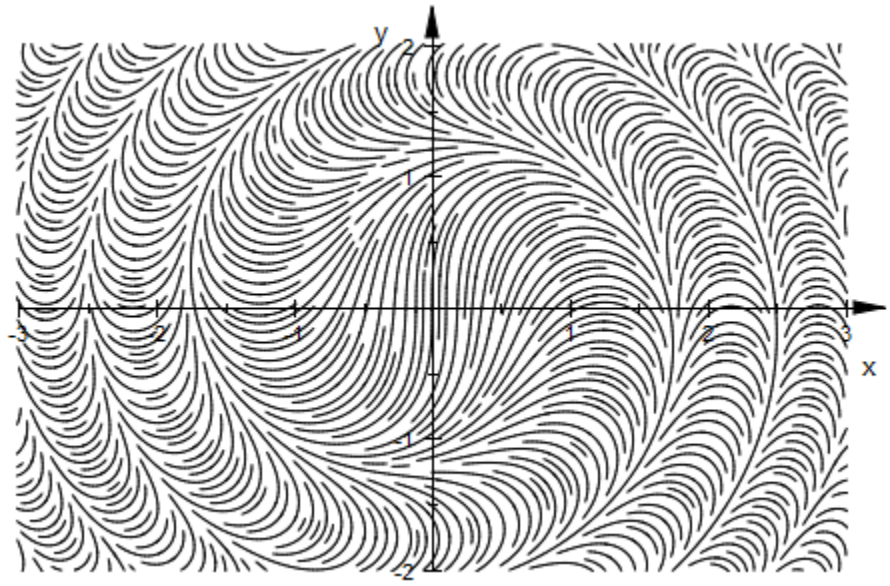
```
plot(plot::Streamlines2d(sin(x^2+y^2), cos(x^2+y^2),
                         x = -3..3, y = -2..2))
```




```
plot(plot::Streamlines2d(sin(x^2+y^2), cos(x^2+y^2),
                         x = -3..3, y = -2..2,
                         MinimumDistance = 0.2))
```



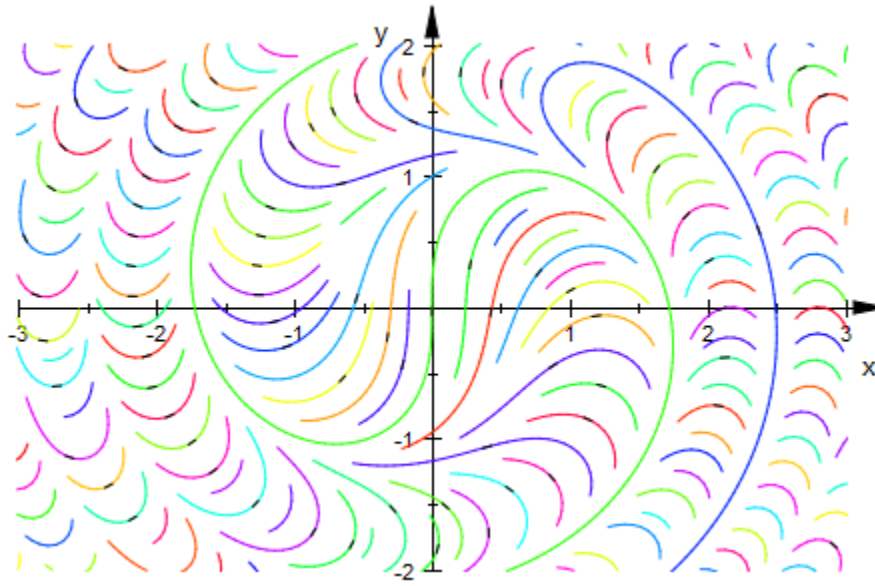

```
plot(plot::Streamlines2d(sin(x^2+y^2), cos(x^2+y^2),
                         x = -3..3, y = -2..2,
                         MinimumDistance = 0.05))
```



Example 3

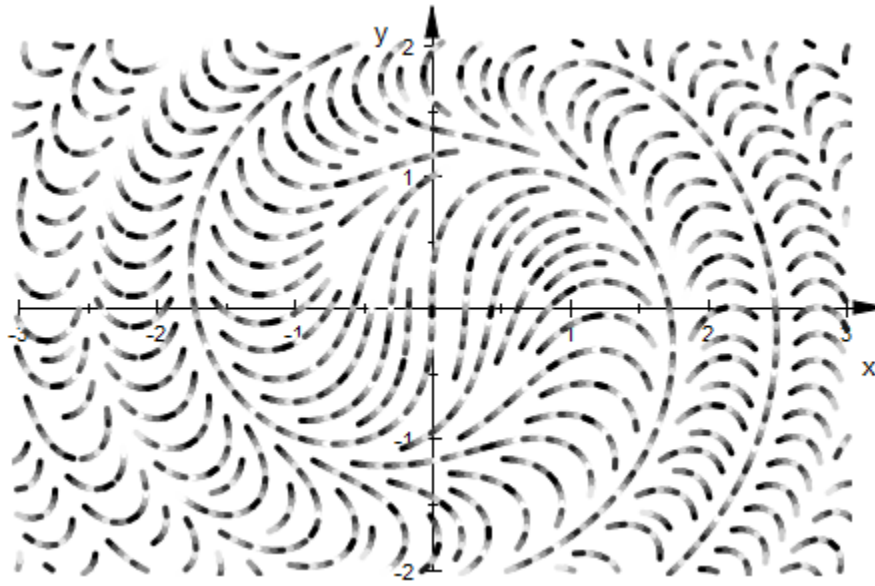
A line color function for `plot::Streamlines2d` has access to the current coordinates, to the components of the vector field at the current point, to the current length on the curve (both in terms of the "time" parameter and Euclidean distance) and an integer count of the current curve (which are not found in some predefined order). We use the curve number to generate a colorful display:

```
num2col := (x, y, vx, vy, t, l, n) -> RGB::fromHSV([111*n, 1, 1]):
plot(plot::Streamlines2d(sin(x^2+y^2), cos(x^2+y^2),
                         x = -3..3, y = -2..2,
                         LineColorFunction = num2col))
```



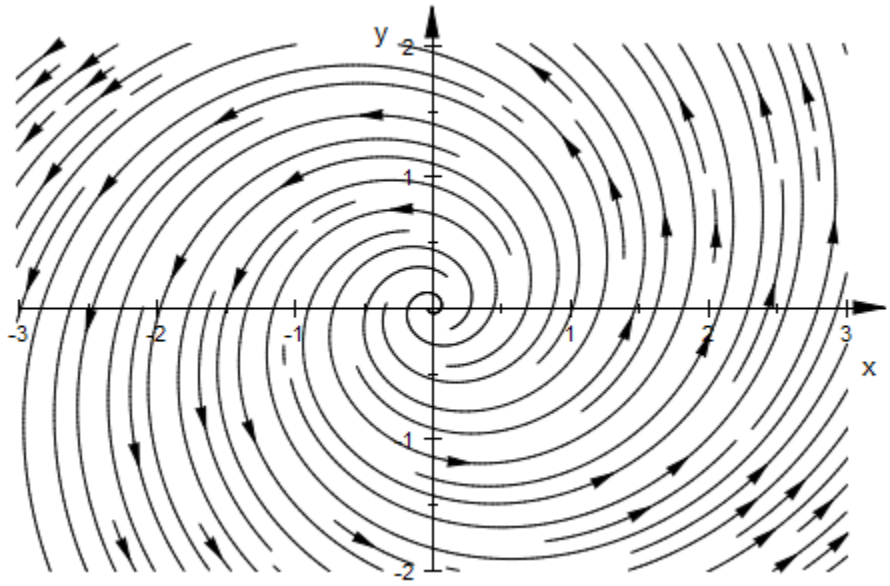
Using the curve length information allows us to include directional information in the visual display:

```
l2col := (x, y, vx, vy, t, l) -> [frac(5*l) $ 3]:
plot(plot::Streamlines2d(sin(x^2+y^2), cos(x^2+y^2),
                         x = -3..3, y = -2..2,
                         LineWidth = 0.75,
                         LineColorFunction = l2col))
```



Often, an easier way of plotting the orientation of the stream lines is to activate the arrow heads `plot::Streamlines2d` plots at the middle of each sufficiently long) stream line. These are made invisible by the default tip length of 0:

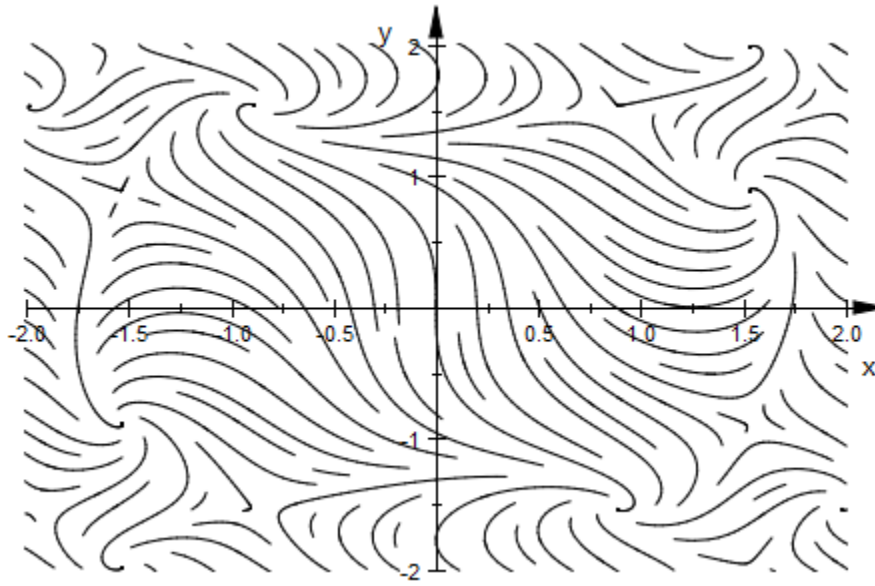
```
plot(plot::Streamlines2d(0.3*x-y, 0.3*y+x,
                         x = -3..3, y = -2..2,
                         TipLength = 3*unit::mm))
```



Example 4

Since the placement of stream lines is hard to predict, `plot::Streamlines2d` is not really suitable for animations. It is possible to animate `plot::Streamlines2d`, but coherence between the animation frames is less than usual:

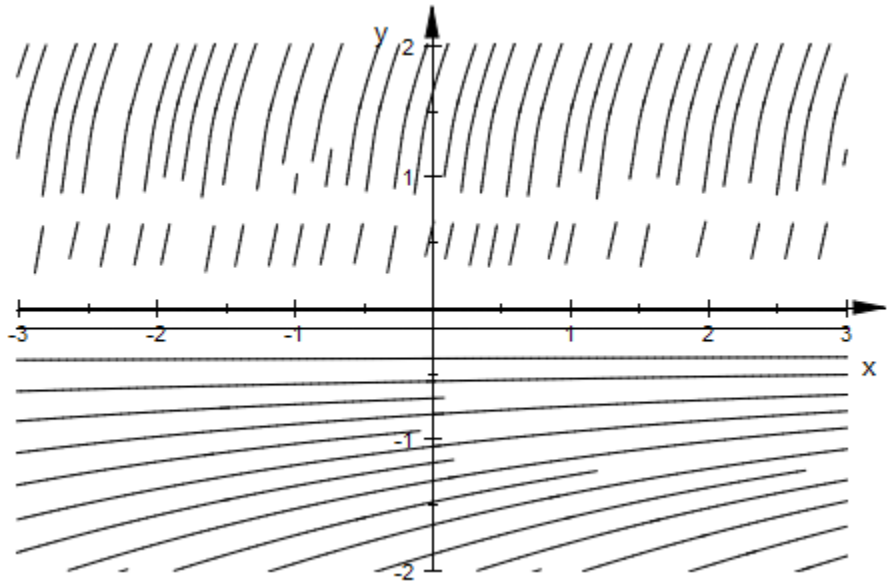
```
plot(plot::Streamlines2d(sin(x^2+y^2), cos(x^2-y^2+a),
                         x = -2..2, y = -2..2, a = -PI..PI,
                         MinimumDistance = 0.1,
                         Frames=10))
```



Example 5

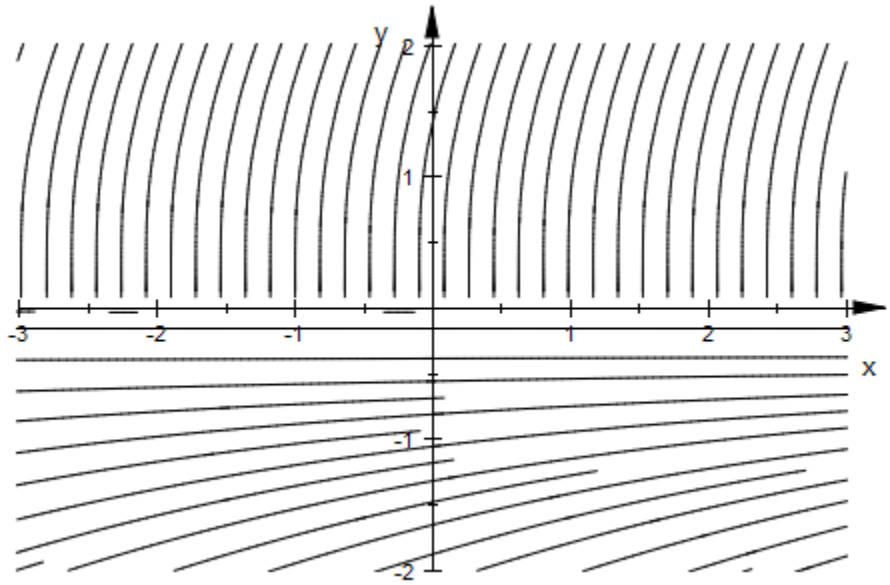
With the default settings, `plot::Streamlines2d` is not able to plot the vector field $\left[1, 3^{2/y}\right]$ (which is not Lipschitz continous) in a satisfying way:

```
plot(plot::Streamlines2d([1, surd(3,y)^2],
                         x=-3..3, y=-2..2))
```



By using a different numerical integrator, the problems can be overcome (at the cost of longer computation):

```
plot(plot::Streamlines2d([1, surd(3,y)^2],
                         x=-3..3, y=-2..2,
                         ODEMethod=RKF43,
                         RelativeError=1e-3))
```



Parameters

v_1, v_2

The *x*- and *y*-component of the vector field: arithmetical expressions in `x`, `y`, and, possibly, the animation parameter `a`.

v_1, v_2 are equivalent to the attributes `XFunction`, `YFunction`.

v

A matrix of category `Cat::Matrix` with two entries that provide the components v_1, v_2 of the vector field.

x, y

Identifiers.

x, y are equivalent to the attributes `XName`, `YName`.

x_{min} .. x_{max}, y_{min} .. y_{max}

Real numerical values.

x_{min} .. x_{max}, y_{min} .. y_{max} are equivalent to the attributes `XRange`, `YRange`, `XMin`, `XMax`, `YMin`, `YMax`.

a

Animation parameter, specified as a = a_{min}..a_{max}, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

Algorithms

The algorithm used in `plot::Streamlines2d` has been published in "Creating Evenly-Spaced Streamlines of Arbitrary Density" by Bruno Jobard and Wilfrid Lefer at the Eurographics Workshop in Boulogne-sur-Mer, France.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Ode2d` | `plot::Ode3d` | `plot::VectorField2d` | `plot::VectorField3d`

plot::Sum

Graphical primitive for symbolic sums

Syntax

```
plot::Sum(ex, i = m .. n, <a = a_min .. a_max>, options)
```

```
plot::Sum(sum(ex, i = m .. n), <a = a_min .. a_max>, options)
```

Description

`plot::Sum(ex, i = m..n)` creates a plot of summing `ex` over the range `m..n`.

`plot::Sum` creates a visual display of partial sums over a finite interval. Mathematically, `plot::Sum(ex, i = m..n)` plots the function $x \to \sum_{i=m}^{\lfloor x-m \rfloor} (m + \mathrm{ex})$.

To ease the use of `plot::Sum` in programs, symbolic sums are accepted in the input and `plot::Sum` takes care not to evaluate these. It is highly recommended, though, not to use this syntax in interactive applications, to avoid premature evaluation.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`FALSE`
`Color`	the main color	`RGB::Blue`
`Filled`	filled or transparent areas and surfaces	`FALSE`
`FillColor`	color of areas and surfaces	`RGB::Red`
`FillPattern`	type of area filling	`Solid`

Attribute	Purpose	Default Value
`Frames`	the number of frames in an animation	`50`
`Function`	function expression or procedure	
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`LineColor`	color of lines	`RGB::Blue`
`LineWidth`	width of lines	`0.35`
`LineColor2`	color of lines	`RGB::DeepPink`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LinesVisible`	visibility of lines	`TRUE`
`LineColorType`	line coloring types	`Flat`
`LineColorFunction`	functional line coloring	
`LineColorDirection`	the direction of color transitions on lines	[0, 1]
`LineColorDirectionX`	x-component of the direction of color transitions on lines	`0`
`LineColorDirectionY`	y-component of the direction of color transitions on lines	`1`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	

Attribute	Purpose	Default Value
ParameterRange	range of the animation parameter	
PointColor	the color of points	RGB::MidnightBlue
PointsVisible	visibility of mesh points	FALSE
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XMax	final value of parameter “x”	
XMin	initial value of parameter “x”	

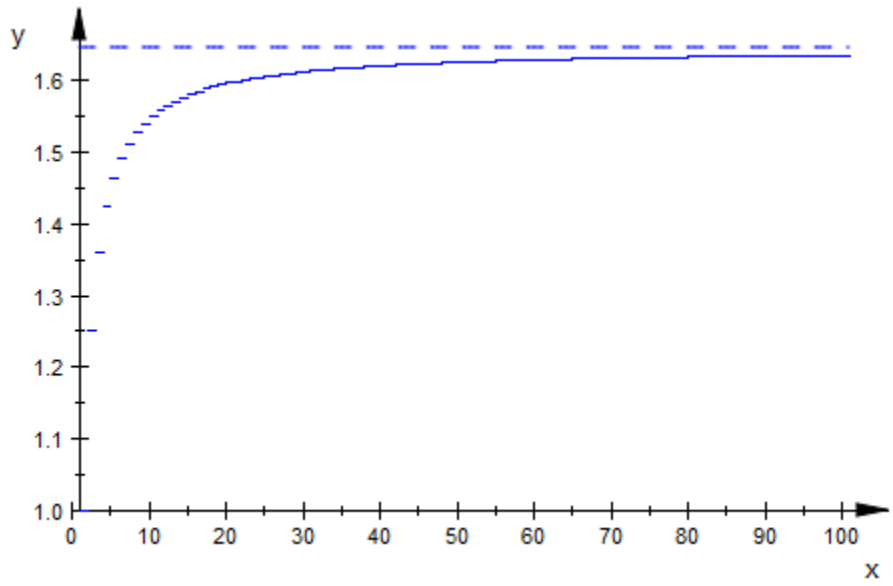
Attribute	Purpose	Default Value
XName	name of parameter “x”	
XRange	range of parameter “x”	

Examples

Example 1

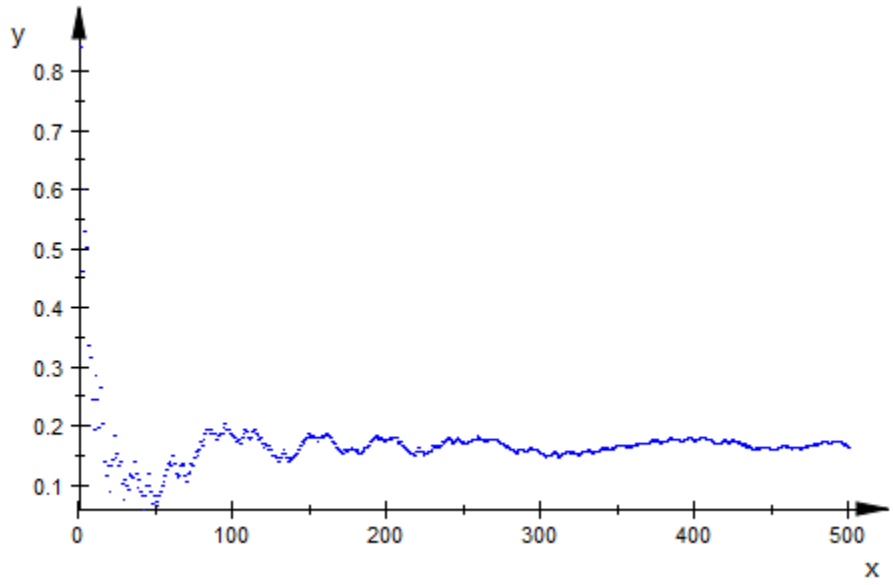
It is well known that $\sum_{j=1}^{\infty} \frac{1}{j^2} = \frac{\pi^2}{6}$. We use `plot::Sum` to display the first 100 partial sums:

```
plot(plot::Sum(1/j^2, j = 1..100),
     plot::Function2d(PI^2/6, x=1..101, LineStyle = Dashed))
```



With more partial sums, the steps approximate points:

```
plot(plot::Sum(sin(j^2)/j, j=1..500))
```



Example 2

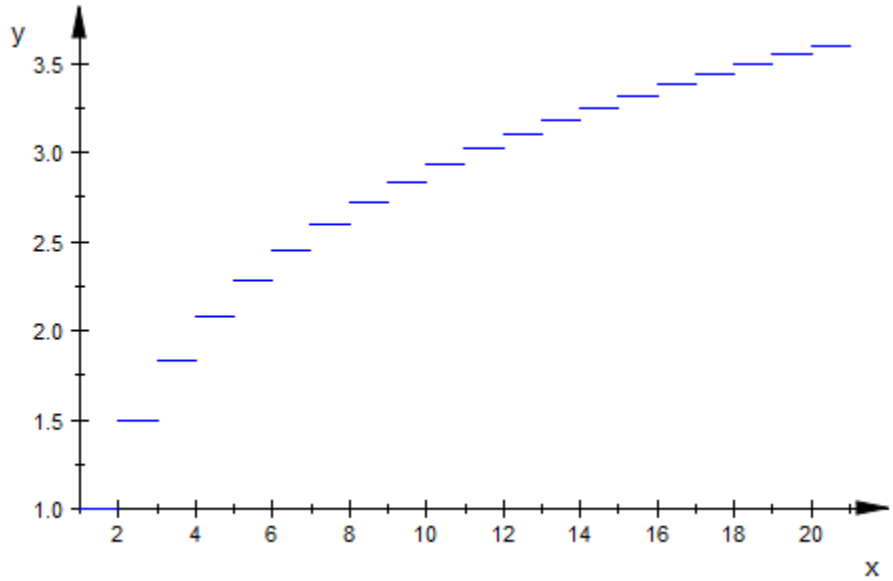
To show some of the formatting options of `plot::Sum`, we use the following sum:

```
s := plot::Sum(1/j, j = 1..20)
```

$$\text{plot::Sum}\left(\frac{1}{j},\ j = 1..20\right)$$

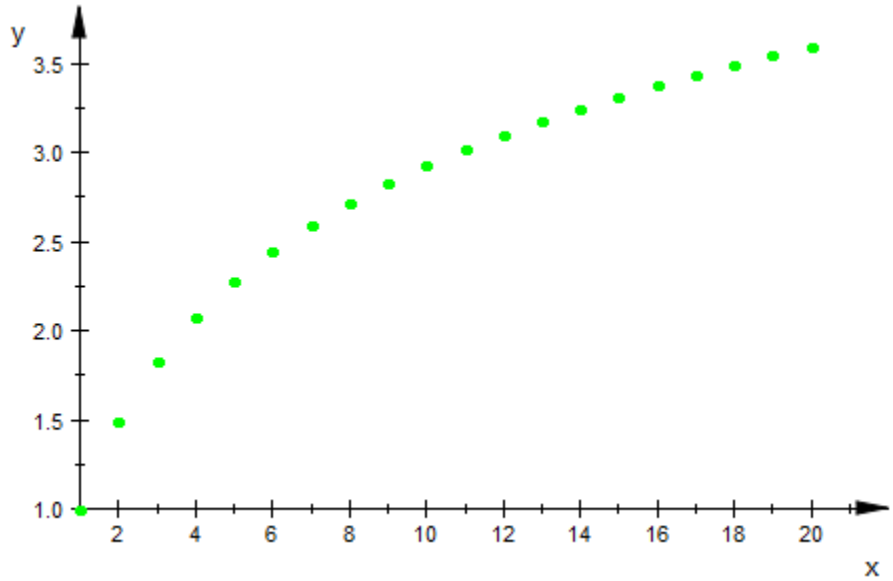
By default, this object is displayed as follows:

```
plot(s)
```



To change parameters, we can select them in the inspector and change the values, we can give other values directly in the `plot` command or we can set the new values in our object `s`:

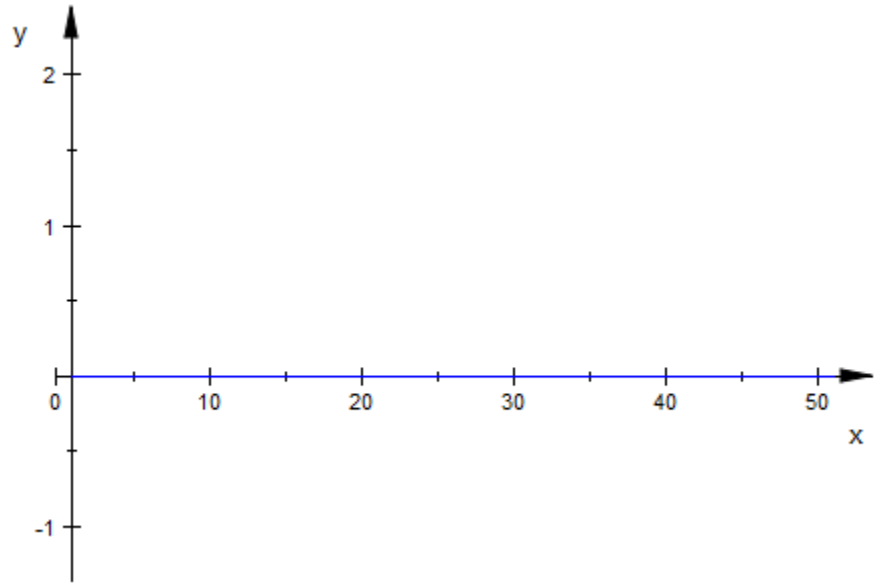
```
s::PointsVisible := TRUE:
s::LinesVisible := FALSE:
s::PointColor := RGB::Green:
plot(s)
```



Example 3

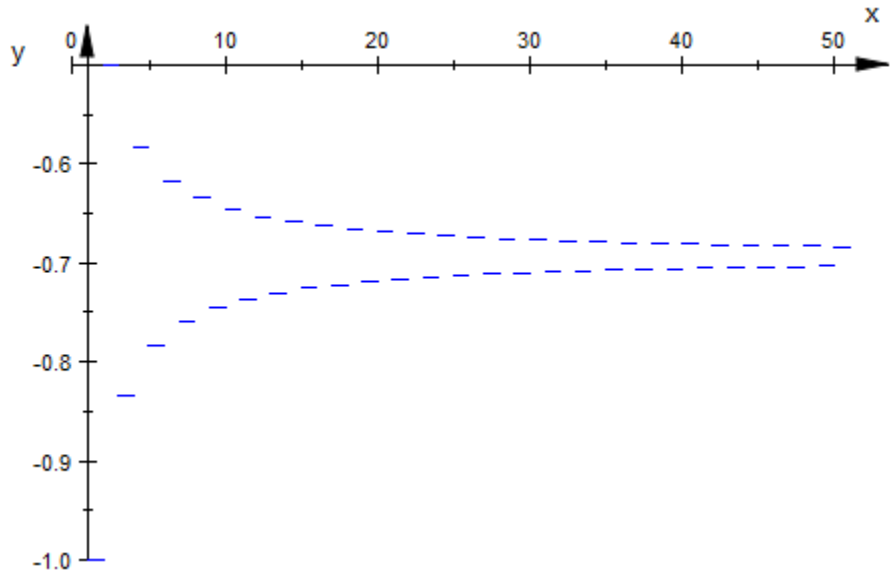
`plot::Sum` allows animation in the usual way, for example, in the term to sum:

```
plot(plot::Sum(sin(a*i^2)/i, i = 1..50, a = 0..PI))
```



Another interesting parameter to animate is the summation range:

```
plot(plot::Sum((-1)^j/j, j = 1..jmax, jmax = 1..50))
```



Parameters

`ex`

Arithmetical expression in `i` and the animation parameter `a`, if that is used.

ex is equivalent to the attribute `Function`.

`i`

An identifier or indexed identifier.

i is equivalent to the attribute `XName`.

`m .. n`

The range of `i`. `m` and `n` may be expressions in the animation parameter `a`. Summation goes over `m` + integer. If `n - m` is not an integer, `n` will not be reached.

m .. n is equivalent to the attributes `XRange`, `XMin`, `XMax`.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Function2d` | `plot::PointList2d`

plot::Surface

Parametrized surfaces in 3D

Syntax

```
plot::Surface([x, y, z], u = u_min .. u_max, v = v_min .. v_max, <a = a_min .. a_max>, options)
plot::Surface(xyz, u = u_min .. u_max, v = v_min .. v_max, <a = a_min .. a_max>, options)
plot::Surface(A, u = u_min .. u_max, v = v_min .. v_max, <a = a_min .. a_max>, options)
```

Description

`plot::Surface` creates a parametrized surface in 3D.

The surface given by a mapping ("parametrization") $(u, v) \to (x(u, v), y(u, v), z(u, v))$ is the set of all image points

$$\left\{ \begin{pmatrix} x(u, v) \\ y(u, v) \\ z(u, v) \end{pmatrix} \;\middle|\; u \in [u_{min}, u_{max}], v \in [v_{min}, v_{max}] \right\} \in \mathbb{R}^3$$

.

The expressions/functions `x`, `y`, `z` may have singularities in the plot range. Although a heuristics is used to find a reasonable viewing range when singularities are present, it is highly recommended to specify a viewingbox via the attribute

```
ViewingBox = [`x_{min}` .. `x_{max}`, `y_{min}` .. `y_{max}`,
`z_{min}` .. `z_{max}`]
```

with suitable numerical real values $x_{min}, \ldots, z_{max}$. See "Example 3" on page 24-891.

Animations are triggered by specifying a range `a = `a_{min}` .. `a_{max}`` for a parameter `a` that is different from the surface parameters `u`, `v`. See "Example 5" on page 24-894.

The functions `x`, `y`, `z` are evaluated on a regular equidistant mesh of sample points in the *u*-*v* plane. This mesh is determined by the attributes `UMesh`, `VMesh`. By default, the

attribute `AdaptiveMesh = 0` is set, i.e., no adaptive refinement of the equidistant mesh is used.

If the standard mesh does not suffice to produce a sufficiently detailed plot, one may either increase the value of `UMesh`, `VMesh` or `USubmesh`, `VSubmesh`, or set `AdaptiveMesh = n` with some (small) positive integer `n`. If necessary, up to $2^n - 1$ additional points are placed in each direction of the u-v plane between adjacent points of the initial equidistant mesh. See “Example 6” on page 24-895.

The “coordinate lines” (“parameter lines”) are curves on the surface.

The phrase “`ULines`” refers to the curves $(x(u, v_0), y(u, v_0), z(u, v_0))$ with the parameter u running from u_{min} to u_{max}, while v_0 is some fixed value from the interval $[v_{min}, v_{max}]$.

The phrase “`VLines`” refers to the curves $(x(u_0, v), y(u_0, v), z(u_0, v))$ with the parameter v running from v_{min} to v_{max}, while u_0 is some fixed value from the interval $[u_{min}, u_{max}]$.

By default, the parameter curves are visible. They may be “switched off” by specifying `ULinesVisible = FALSE` and `VLinesVisible = FALSE`, respectively.

The coordinate lines controlled by `ULinesVisible = TRUE/FALSE` and `VLinesVisible = TRUE/FALSE` indicate the equidistant mesh in the u-v plane set via the `UMesh`, `VMesh` attributes. If the mesh is refined by the `USubmesh`, `VSubmesh` attributes, or by the adaptive mechanism controlled by `AdaptiveMesh = n`, no additional parameter lines are drawn.

As far as the numerical approximation of the surface is concerned, the settings `UMesh` $= n_u$, `VMesh` $= n_v$, `USubmesh` $= m_u$, `VSubmesh` $= m_v$ and `UMesh` $= (n_u - 1)\ (m_u + 1) + 1$, `VMesh` $= (n_v - 1)\ (m_v + 1) + 1$, `USubmesh = 0`, `VSubmesh = 0` are equivalent. However, in the first setting, n_u parameter lines are visible in the u direction, while in the latter setting $(n_u - 1)\ (m_u + 1) + 1$ parameter lines are visible. See “Example 7” on page 24-898.

Use `Filled = FALSE` to render the surface as a wireframe.

Attributes

Attribute	Purpose	Default Value
`AdaptiveMesh`	adaptive sampling	`0`

Attribute	Purpose	Default Value
AffectViewingBox	influence of objects on the ViewingBox of a scene	TRUE
Color	the main color	RGB::Red
Filled	filled or transparent areas and surfaces	TRUE
FillColor	color of areas and surfaces	RGB::Red
FillColor2	second color of areas and surfaces for color blends	RGB::CornflowerBlue
FillColorType	surface filling types	Dichromatic
FillColorFunction	functional area/surface coloring	
FillColorDirection	the direction of color transitions on surfaces	[0, 0, 1]
FillColorDirectionX	x-component of the direction of color transitions on surfaces	0
FillColorDirectionY	y-component of the direction of color transitions on surfaces	0
FillColorDirectionZ	z-component of the direction of color transitions on surfaces	1
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	TRUE
LineColor	color of lines	RGB::Black.[0.25]
LineWidth	width of lines	0.35

Attribute	Purpose	Default Value
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	0
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Mesh	number of sample points	[25, 25]
MeshVisible	visibility of irregular mesh lines in 3D	FALSE
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE

Attribute	Purpose	Default Value
Shading	smooth color blend of surfaces	Smooth
Submesh	density of submesh (additional sample points)	[0, 0]
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
ULinesVisible	visibility of parameter lines (u lines)	TRUE
UMax	final value of parameter “u”	
UMesh	number of sample points for parameter “u”	25
UMin	initial value of parameter “u”	
UName	name of parameter “u”	
URange	range of parameter “u”	
USubmesh	density of additional sample points for parameter “u”	0

Attribute	Purpose	Default Value
VLinesVisible	visibility of parameter lines (v lines)	TRUE
VMax	final value of parameter “v”	
VMesh	number of sample points for parameter “v”	25
VMin	initial value of parameter “v”	
VName	name of parameter “v”	
VRange	range of parameter “v”	
VSubmesh	density of additional sample points for parameter “v”	0
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XContours	contour lines at constant x values	[]
XFunction	function for x values	
YContours	contour lines at constant y values	[]
YFunction	function for y values	
ZContours	contour lines at constant z values	[]
ZFunction	function for z values	

Examples

Example 1

Using standard spherical coordinates, a parametrization of a sphere of radius r by the azimuth angle $u \in [0, 2\,\pi]$ and the polar angle $v \in [0, \pi]$ is given by:

```
x := r*cos(u)*sin(v):
y := r*sin(u)*sin(v):
z := r*cos(v):
```

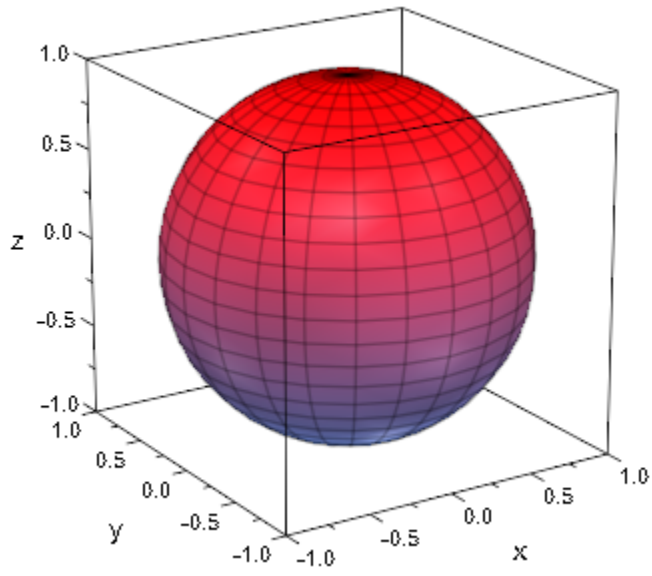
We fix $r = 1$ and create the surface object:

```
r := 1:
s := plot::Surface([x, y, z], u = 0 .. 2*PI, v = 0 .. PI)
```

$$\text{plot::Surface}([\cos(u)\ \sin(v),\ \sin(u)\ \sin(v),\ \cos(v)],\ u = 0..2\,\pi,\ v = 0..\pi)$$

We call `plot` to plot the surface:

```
plot(s, Scaling = Constrained):
```

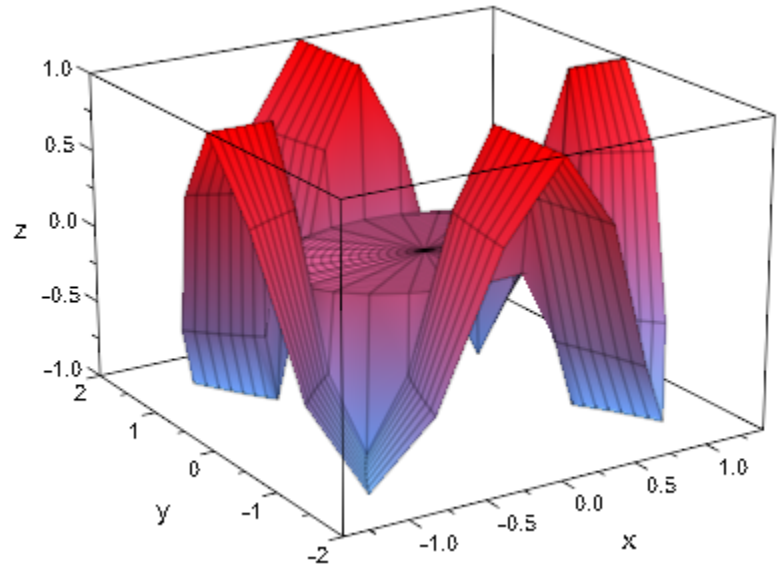


```
delete x, y, z, r, s:
```

Example 2

The parametrization can also be specified by `piecewise` objects or procedures:

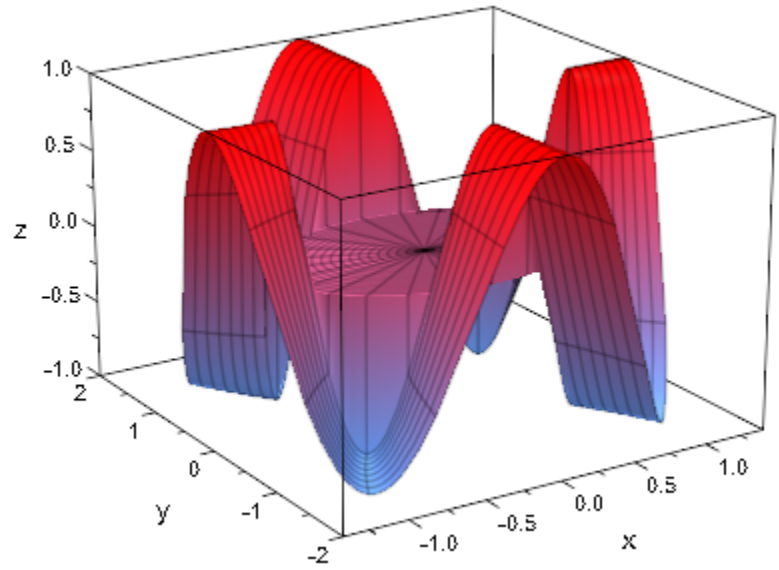
```
x := u*cos(v):
y := piecewise([u <= 1, u*sin(v)], [u >= 1, u^2*sin(v)]):
z := proc(u, v)
begin
  if u <= 1 then
     0
  else
     cos(4*v)
  end_if:
end_proc:
plot(plot::Surface([x, y, z], u = 0 .. sqrt(2), v = 0 .. 2*PI)):
```



We enable adaptive sampling to get a smoother graphical result:

```
plot(plot::Surface([x, y, z], u = 0 .. sqrt(2), v = 0 .. 2*PI),
```

```
                    AdaptiveMesh = 3):
```

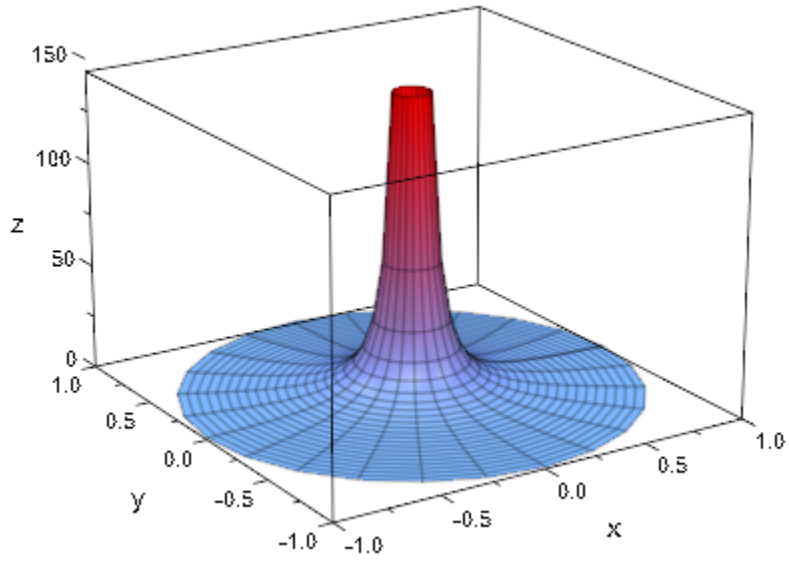



```
delete x, y, z, s, r:
```

Example 3

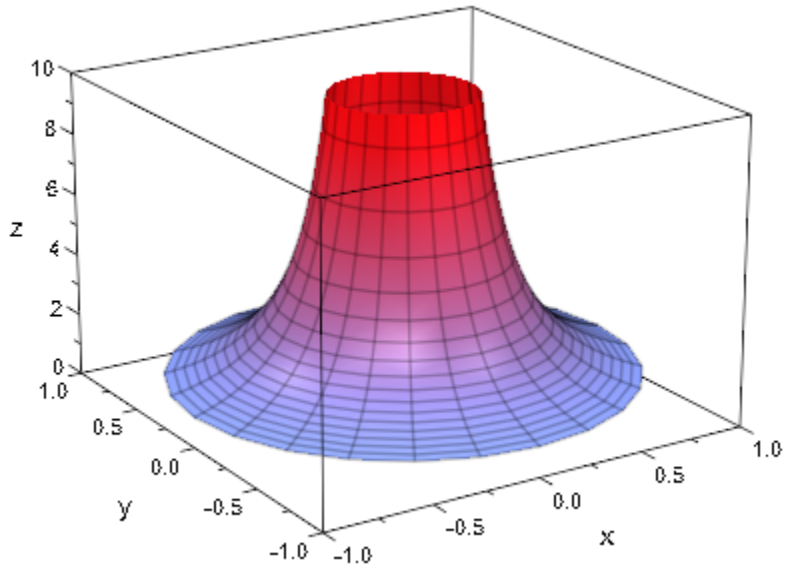
We plot a surface with singularities:

```
s := plot::Surface([u*cos(v), u*sin(v), 1/u^2],
                   u = 0 .. 1, v = 0 .. 2*PI):
plot(s):
```



We specify an explicit viewing range for the *z* coordinate:

```
plot(s, ViewingBox = [Automatic, Automatic, 0 .. 10]):
```

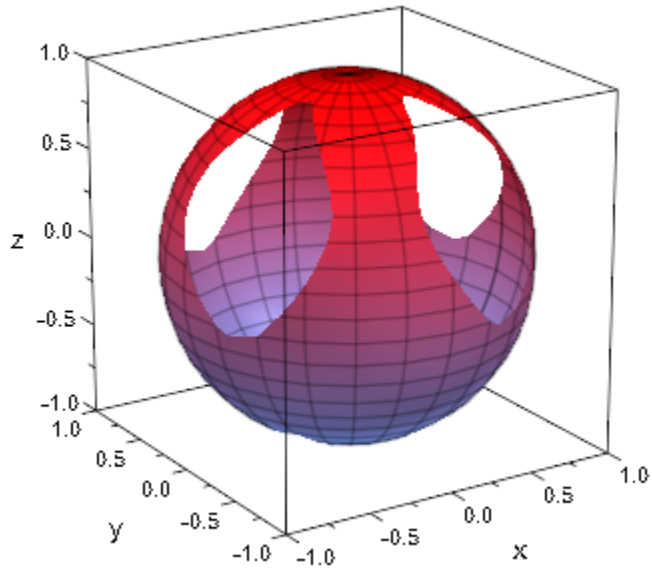



```
delete s:
```

Example 4

By introducing non-real function evaluations, we can plot surfaces with holes:

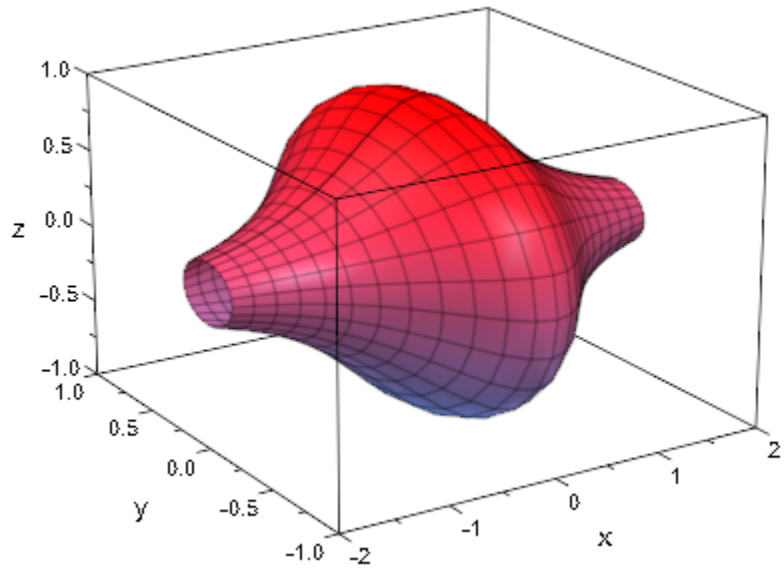
```
chi := piecewise([sin(4*u) < cos(3*v)+0.5, 1]):
plot(plot::Surface([cos(u)*sin(v),
                    sin(u)*sin(v),
                    chi*cos(v)],
                   u = 0 .. 2*PI, v = 0 .. PI,
                   AdaptiveMesh=2), Scaling = Constrained)
```



Example 5

We generate an animation of a surface of revolution. The graph of the function $f(u) = \frac{1}{(1+u^2)}$ is rotated around the *x*-axis:

```
f :=  u -> 1/(1 + u^2):
plot(plot::Surface([u, f(u)*sin(v), f(u)*cos(v)], u = -2 .. 2,
                   v = 0 .. a*2*PI, a = 0 .. 1)):
```



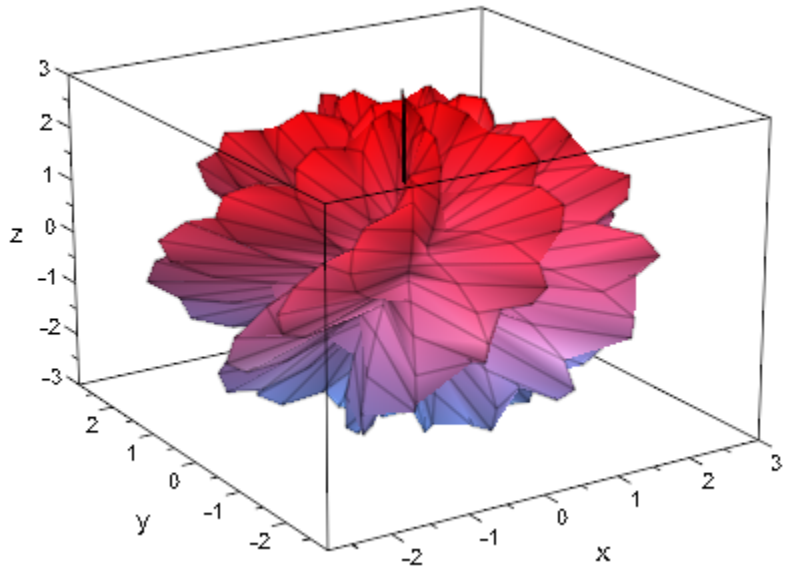
See `plot::XRotate`, `plot::ZRotate` for an alternative way to create surfaces of revolution.

```
delete f:
```

Example 6

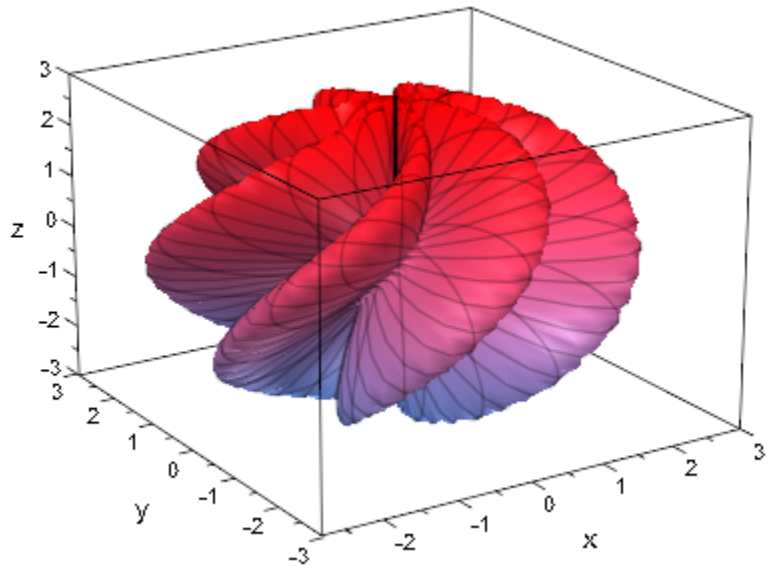
The standard mesh for the numerical evaluation of a surface does not suffice to generate a satisfying plot in the following case:

```
r := 2 + sin(7*u + 5*v):
x := r*cos(u)*sin(v):
y := r*sin(u)*sin(v):
z:=  r*cos(v):
plot(plot::Surface([x, y, z], u = 0 .. 2*PI, v = 0 .. PI)):
```



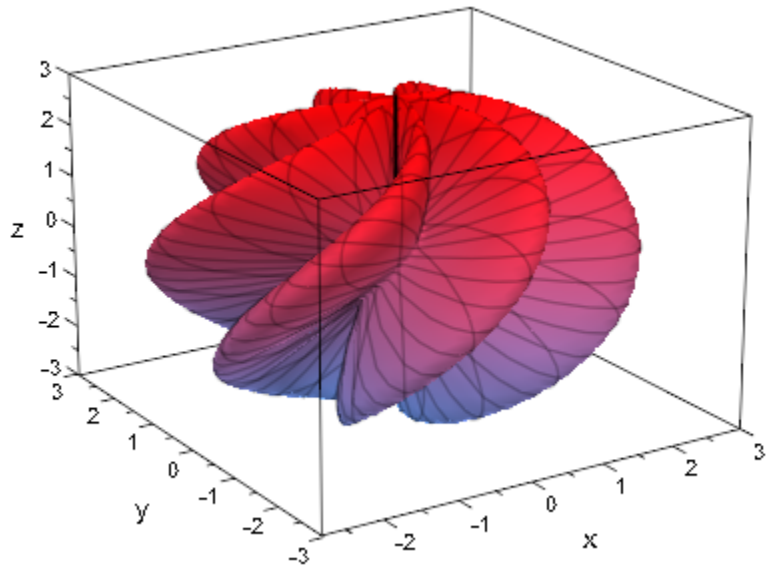
We increase the number of mesh points. Here, we use `USubmesh`, `VSubmesh` to place 2 additional points in each direction between each pair of neighboring points of the default mesh. This increases the runtime for computing the plot by a factor of 9:

```
plot(plot::Surface([x, y, z], u = 0 .. 2*PI, v = 0 .. PI,
                   USubmesh = 2, VSubmesh = 2)):
```



Alternatively, we enable adaptive sampling by setting the value of `AdaptiveMesh` to some positive value:

```
plot(plot::Surface([x, y, z], u = 0 .. 2*PI, v = 0 .. PI,
                   AdaptiveMesh = 2)):
```

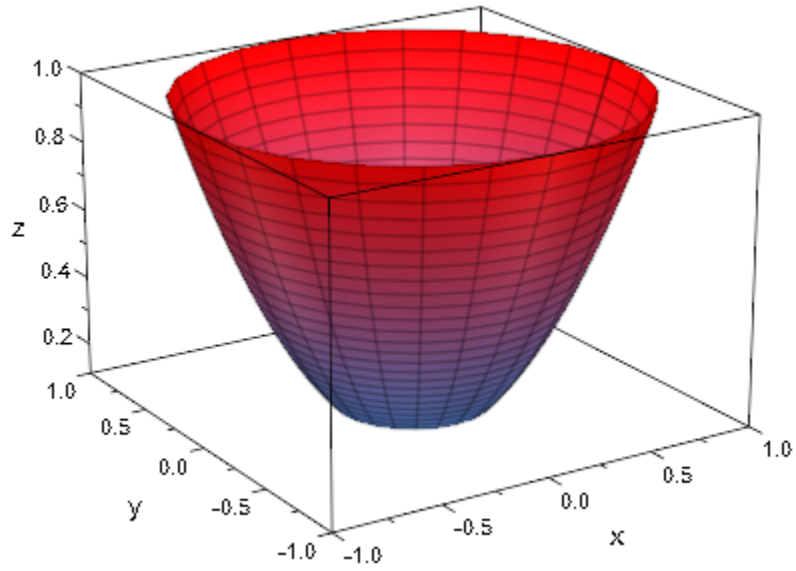



```
delete r, x, y, z:
```

Example 7

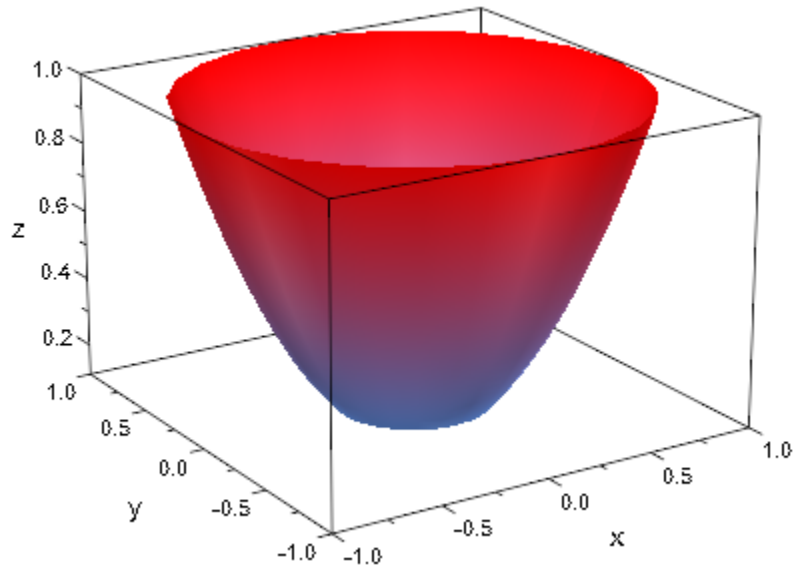
By default, the parameter lines of a parametrized surface are “switched on”:

```
x := r*cos(phi):
y := r*sin(phi):
z := r^2:
plot(plot::Surface([x, y, z], r = 1/3 .. 1, phi = 0 .. 2*PI)):
```



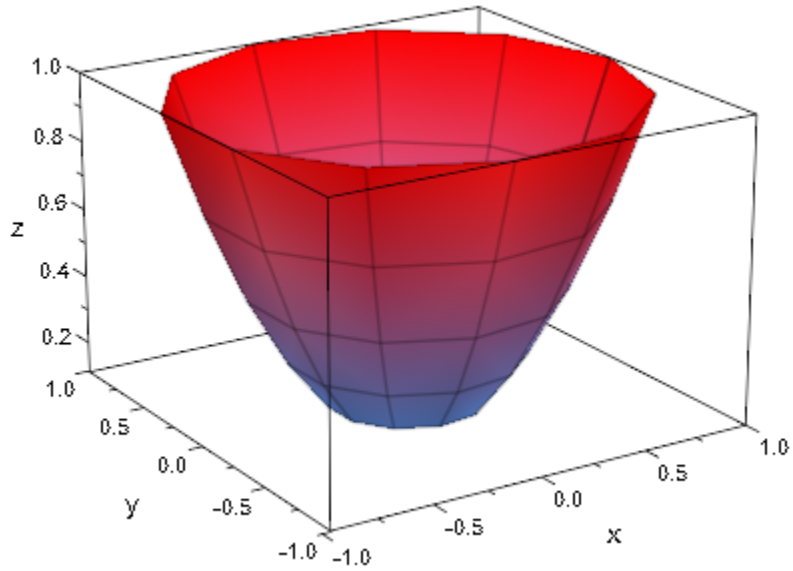
The parameter lines are “switched off”:

```
plot(plot::Surface([x, y, z], r = 1/3 .. 1, phi = 0 .. 2*PI,
                   ULinesVisible = FALSE,
                   VLinesVisible = FALSE)):
```



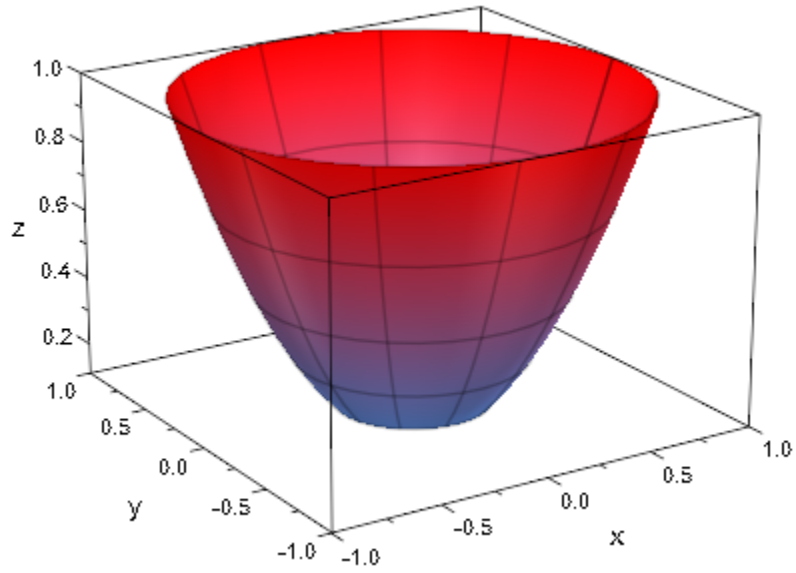
The number of parameter lines are determined by the attributes `UMesh` and `VMesh`:

```
plot(plot::Surface([x, y, z], r = 1/3 .. 1, phi = 0 .. 2*PI,
                   UMesh = 5, VMesh = 12)):
```



When the mesh is refined via the attributes `USubmesh`, `VSubmesh`, the numerical approximation of the surface becomes smoother. However, the number of parameter lines is determined by the values of `UMesh`, `VMesh` and is not increased:

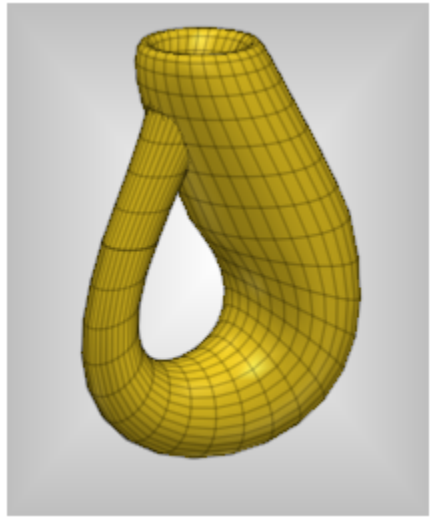
```
plot(plot::Surface([x, y, z], r = 1/3 .. 1, phi = 0 .. 2*PI,
                   UMesh = 5, VMesh = 12,
                   USubmesh = 1, VSubmesh = 2)):
```



Example 8

Klein's bottle is a surface without orientation. There is no “inside” and no “outside” of the following object:

```
bx := u -> -6*cos(u)*(1 + sin(u)):
by := u -> -14*sin(u):
r :=  u -> 4 - 2*cos(u):
x := (u, v) -> piecewise([u <= PI, bx(u) - r(u)*cos(u)*cos(v)],
                         [PI < u,  bx(u) + r(u)*cos(v)]):
y := (u, v) -> r(u)*sin(v):
z := (u, v) -> piecewise([u <= PI, by(u) - r(u)*sin(u)*cos(v)],
                         [PI < u,  by(u)]):
KleinBottle:= plot::Surface(
      [x, y, z], u = 0 .. 2*PI, v = 0 .. 2*PI,
      Mesh = [35, 31], LineColor = RGB::Black.[0.2],
      FillColorFunction = RGB::MuPADGold):
plot(KleinBottle, Axes = None, Scaling = Constrained,
     Width = 60*unit::mm, Height = 72*unit::mm,
     BackgroundStyle = Pyramid):
```

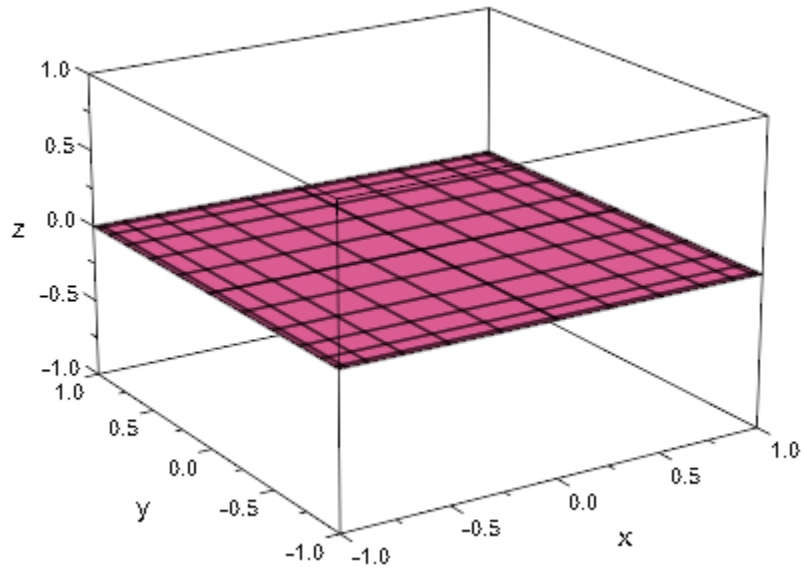


```
delete bx, by, r, x, y, z, KleinBottle:
```

Example 9

Finally we create an animated surface plot of $(u, v) \to (\sin(u), \sin(v), a \sin(u+v))$ where a is the animation parameter:

```
plot(
  plot::Surface(
   [sin(u),sin(v),a*sin(u+v)],
   u=0..2*PI, v=0..2*PI, a=1..0,
   AnimationStyle = BackAndForth
  )
)
```



Parameters

x, y, z

The coordinate functions: arithmetical expressions or `piecewise` objects depending on the surface parameters *u*, *v* and the animation parameter `a`. Alternatively, procedures that accept 2 input parameters *u*, *v* or 3 input parameters *u*, *v*, *a* and return a numerical value when the input parameters are numerical.

x, y, z are equivalent to the attributes `XFunction`, `YFunction`, `ZFunction`.

xyz

The parametrization: a procedure that accepts 2 input parameters *u*, *v* or 3 input parameters *u*, *v*, *a* and returns a list of 3 numerical values [*x*, *y*, *z*].

A

A matrix of category `Cat::Matrix` with three entries that provide the parametrization `x`, `y`, `z`

u

The first surface parameter: an identifier or an indexed identifier.

u is equivalent to the attribute `UName`.

u_{min} .. u_{max}

The plot range for the parameter u: u_{min}, u_{max} must be numerical real values or expressions of the animation parameter a.

u_{min} .. u_{max} is equivalent to the attributes `URange`, `UMin`, `UMax`.

v

The second surface parameter: an identifier or an indexed identifier.

v is equivalent to the attribute `VName`.

v_{min} .. v_{max}

The plot range for the parameter v: v_{min}, v_{max} must be numerical real values or expressions of the animation parameter a.

v_{min} .. v_{max} is equivalent to the attributes `VRange`, `VMin`, `VMax`.

a

Animation parameter, specified as a = a_{min}..a_{max}, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions

`plot` | `plot::copy` | `plotfunc3d`

MuPAD Graphical Primitives

`plot::Function3d` | `plot::Matrixplot`

plot::SurfaceSet

Triangle and quad surface meshes

Syntax

```
plot::SurfaceSet(meshlist, <MeshListType = t>, <MeshListNormals = n>, <UseNormals = b>
```

Description

`plot::SurfaceSet(MeshList)` creates a 3D graphical object from a given list of triangle or quad coordinates.

`MeshList` contains coordinates of points (and optional normals) of either triangles or quads which define a mesh of a 3D surface. The points must be given homogenous: If a normal is given, it must be given for all points or facets, respectively. The attribute `MeshListType` specifies how these points are to be interpreted for plotting the surface. The attribute `MeshListNormals` specifies whether the list contains normal vectors and at which positions they located.

`MeshListType` specifies how the points in `MeshList` are to be interpreted for plotting the surface. See `MeshList` for more information about mesh list types. Cf. “Example 4” on page 24-914.

`MeshListNormals` specifies whether `MeshList` contains normals and at which positions they are located. See `MeshList` for more information about normals and facet orientation.

When setting the attribute `UseNormals` to FALSE the normals defined in `MeshList` are ignored when plotting the object in MuPAD. This reduces the data volume of the graphics object and the computing time as well. However, it leads to a less brilliant image.

User-defined color functions `LineColorFunction` and `FillColorFunction` will be called with the index of the current point as its first parameter, followed by the x, y, and z coordinate of the current point.

The transformation objects `plot::Rotate3d`, `plot::Scale3d`, `plot::Translate3d` and `plot::Transform3d` can be applied to the imported STL object. Cf. “Example 8” on page 24-926.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Color`	the main color	`RGB::Red`
`Filled`	filled or transparent areas and surfaces	`TRUE`
`FillColor`	color of areas and surfaces	`RGB::Red`
`FillColor2`	second color of areas and surfaces for color blends	`RGB::CornflowerBlue`
`FillColorType`	surface filling types	`Dichromatic`
`FillColorFunction`	functional area/surface coloring	
`FillColorDirection`	the direction of color transitions on surfaces	`[0, 0, 1]`
`FillColorDirectionX`	x-component of the direction of color transitions on surfaces	`0`
`FillColorDirectionY`	y-component of the direction of color transitions on surfaces	`0`
`FillColorDirectionZ`	z-component of the direction of color transitions on surfaces	`1`
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`LineColor`	color of lines	`RGB::Black.[0.25]`

Attribute	Purpose	Default Value
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	0
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
MeshList	triangulation data	
MeshVisible	visibility of irregular mesh lines in 3D	FALSE
MeshListType	triangulation data	Triangles
MeshListNormals	triangulation data	None
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5

Attribute	Purpose	Default Value
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE
Shading	smooth color blend of surfaces	Smooth
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
UseNormals	use pre-defined normals?	TRUE
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE

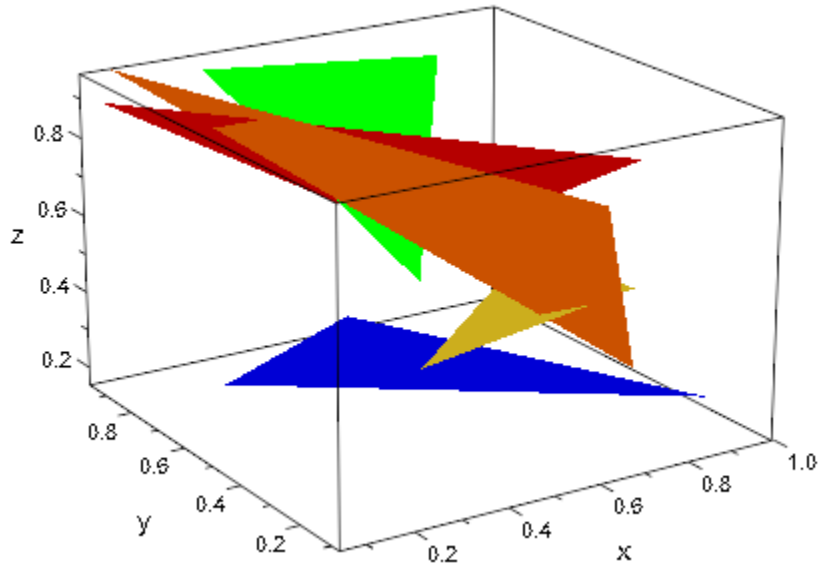
Attribute	Purpose	Default Value
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

When given a list of real numbers, `plot::SurfaceSet` by default considers them as the coordinates of points in 3D forming triangles. Note that we are using `FillColorFunction` here to make the triangles easier to see and that the number of values must be divisible by 9, since each triangle needs 9 numbers to be specified:

```
plot(plot::SurfaceSet([frandom() $ i = 1..9*5],
  FillColorFunction = (i -> RGB::ColorList[floor((i+2)/3)]))):
```



Example 2

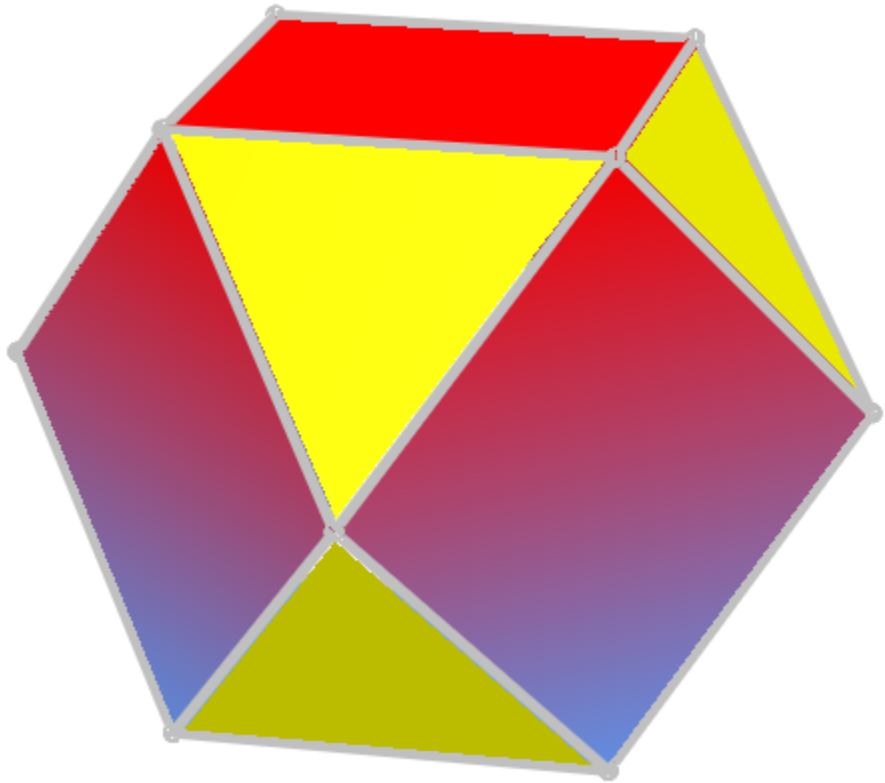
This examples demonstrates how surface sets can be created and animated in MuPAD. First we create a mesh of points:

```
delete cx, cy, cz, r:
F:= [[[cx-r  ,cy-r+a,cz+r], [cx-r+a,cy-r  ,cz+r],
      [cx+r-a,cy-r  ,cz+r], [cx+r  ,cy-r+a,cz+r],
      [cx+r  ,cy+r-a,cz+r], [cx+r-a,cy+r  ,cz+r],
      [cx-r+a,cy+r  ,cz+r], [cx-r  ,cy+r-a,cz+r]],
     [[cx+r,cy-r  ,cz-r+a], [cx+r,cy-r+a,cz-r  ],
      [cx+r,cy+r-a,cz-r  ], [cx+r,cy+r  ,cz-r+a],
      [cx+r,cy+r  ,cz+r-a], [cx+r,cy+r-a,cz+r  ],
      [cx+r,cy-r+a,cz+r  ], [cx+r,cy-r  ,cz+r-a]],
     [[cx-r  ,cy+r,cz-r+a], [cx-r+a,cy+r,cz-r  ],
      [cx+r-a,cy+r,cz-r  ], [cx+r  ,cy+r,cz-r+a],
      [cx+r  ,cy+r,cz+r-a], [cx+r-a,cy+r,cz+r  ],
      [cx-r+a,cy+r,cz+r  ], [cx-r  ,cy+r,cz+r-a]]]:

F:= F.[subs(F[1], cz+r=cz-r),
       subs(F[2], cx+r=cx-r),
       subs(F[3], cy+r=cy-r)]:

T:= [[cx+r,cy-r+a,cz+r], [cx+r-a,cy-r,cz+r], [cx+r,cy-r,cz+r-a]]:
T:= T.subs(T, cx+r-a=cx-r+a, cx+r=cx-r):
T:= T.subs(T, cy-r+a=cy+r-a, cy-r=cy+r):
T:= T.subs(T, cz+r-a=cz-r+a, cz+r=cz-r):
```

Then we create plot objects using the mesh above:

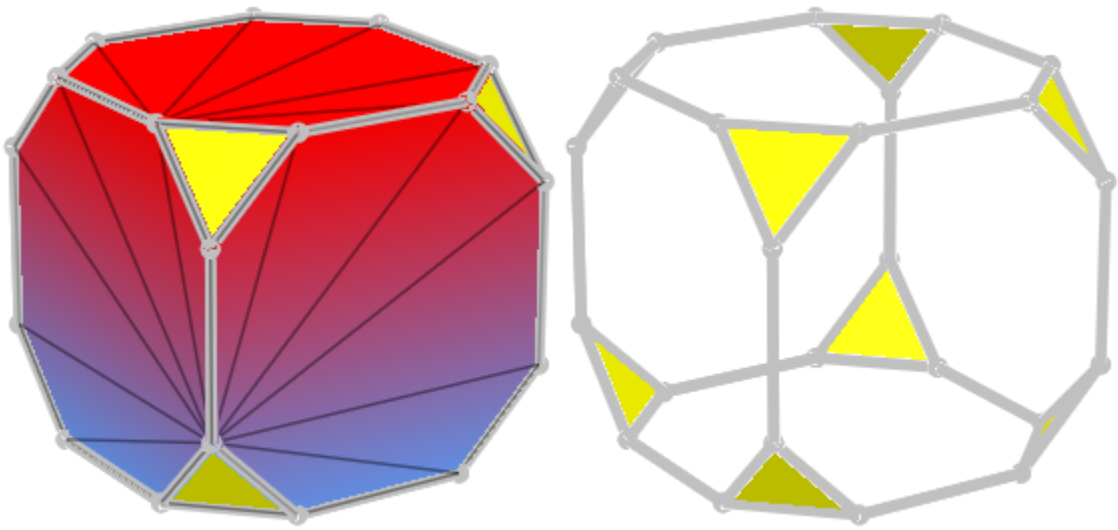
```
cx := 0: cy := 0: cz := 0:
r := 1:
P := range ->
  plot::Group3d(
    plot::Group3d(
        plot::SurfaceSet(map(F[i], op), a = range,
                         MeshListType = TriangleFan)
        $ i=1..6),
    plot::Group3d(
        plot::Polygon3d(F[i], a = range, Closed) $ i=1..6,
        LineWidth = 1.5,
        LineColor = RGB::Grey,
        PointsVisible,
        PointSize = 3),
```

```
      plot::Group3d(
        plot::SurfaceSet(map(T, op), a = range),
        FillColorType = Flat,
        FillColor=RGB::Yellow,
        Filled)
  ):

plot(P(0..r),
     Scaling = Constrained,
     Width = 120, Height = 120,
     Axes = None):
```



The first half of this animation is plotted again. In the left image we can see how parts of the surface set are constructed as triangle fan. In the right image parts of the surface are displayed as wireframe:

```
plot(
  plot::Scene3d(P(0..r/2), MeshVisible = TRUE),
  plot::Scene3d(P(0..r/2), Filled = FALSE),
  Scaling = Constrained, Width = 150, Height = 75,
  Axes = None, Layout = Horizontal
):
```

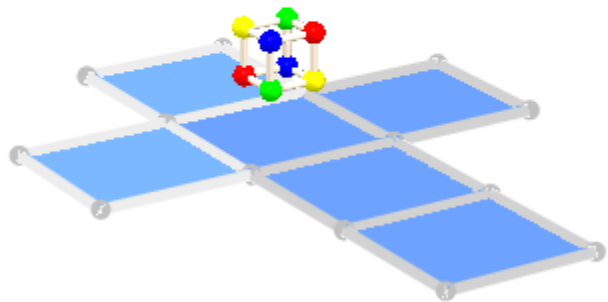


Example 3

A second animation demonstrates the fold back of a cube:

```
r := 1:
bottom := [[0, 0, 0], [r, 0, 0],
           [r, r, 0], [0, r, 0]]:
left   := [[0, 0, 0], [0, -r*sin(a), r*cos(a)],
           [r, -r*sin(a), r*cos(a)], [r, 0, 0]]:
right  := map(left, l -> [l[1], r-l[2], l[3]]):
front  := map(left, l -> [l[2], l[1], l[3]]):
back   := map(right, l -> [l[2], l[1], l[3]]):
```

```
top    := [left[3], left[2],
           zip(left[2], [0, -r*sin(2*a-PI/2), r*cos(2*a-PI/2)], `+`),
           zip(left[3], [0, -r*sin(2*a-PI/2), r*cos(2*a-PI/2)], `+`)]:

plot(plot::SurfaceSet(map(bottom.left.top.right.front.back, op),
                      MeshListType = Quads,
                      PointsVisible = TRUE,
                      PointSize = 3,
                      MeshVisible = TRUE,
                      LineWidth = 1.5,
                      LineColor = RGB::Grey,
                      a=0..PI/2),
     plot::MuPADCube(Radius = r/3, Center = [r/2 $ 3]),
     Scaling = Constrained)
```



```
delete r, bottom, left, right, front, back, top:
```

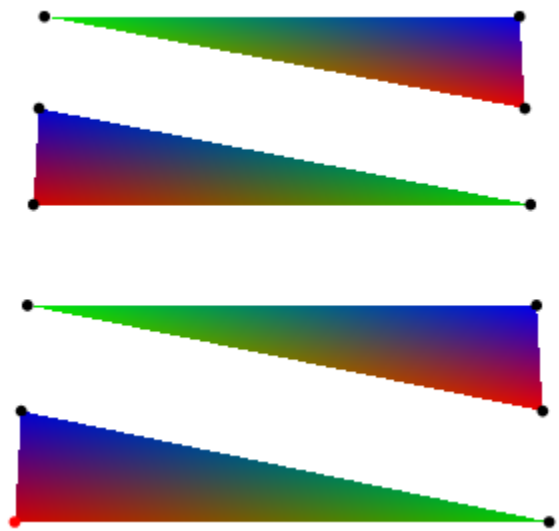
Example 4

Let's have a deeper look on the different kind of mesh types. We create a mesh of points first and then plot it using the different mesh types available. The first point will always be plotted in red color:

```
PL:= [((0,i,0.5-0.1*i), (1,i,0.5-0.1*i)) $ i = 0..5]:
SO:= FillColorFunction =
     ((n,x,y,z)->[RGB::Blue,RGB::Red,RGB::Green][(n mod 3)+1]),
     LineColorFunction =
     ((n,x,y,z)-> if n=1 then RGB::Red else RGB::Black end_if),
     PointsVisible:
VO:= plot::Camera([0.5,2.5,4.5], [0.5,2.51,0], 0.2),
     ViewingBox = [0..1,0..5,0..0.5],
     Axes = None:
```

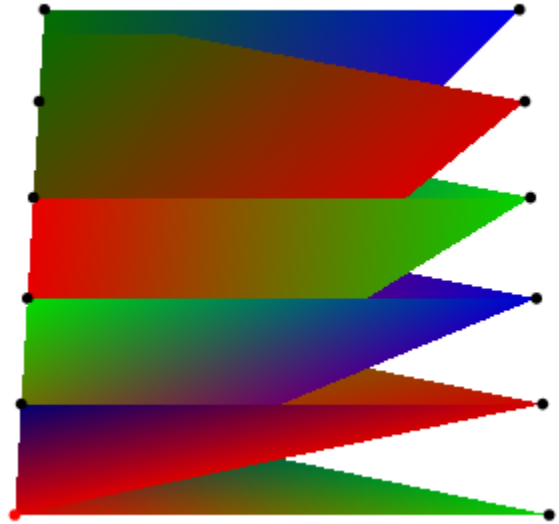
We tell MuPAD to interpret the given mesh list as a set of separate triangles. The corresponding plot looks like this:

```
plot(plot::SurfaceSet(PL, SO, MeshListType = Triangles), VO):
```



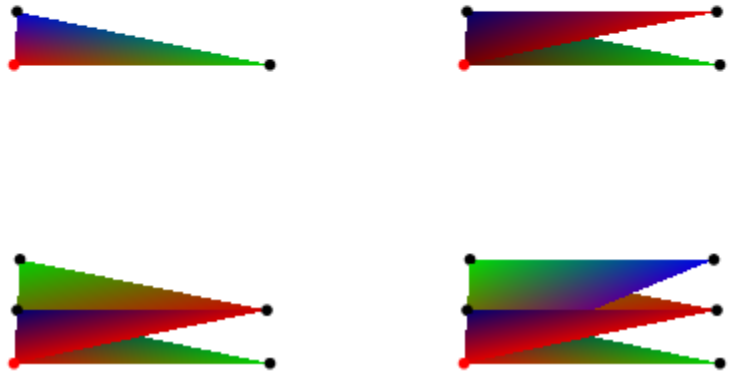
We tell MuPAD to interpret the given mesh list as a triangle fan. The corresponding plot looks like this:

```
plot(plot::SurfaceSet(PL, SO, MeshListType = TriangleFan), VO):
```



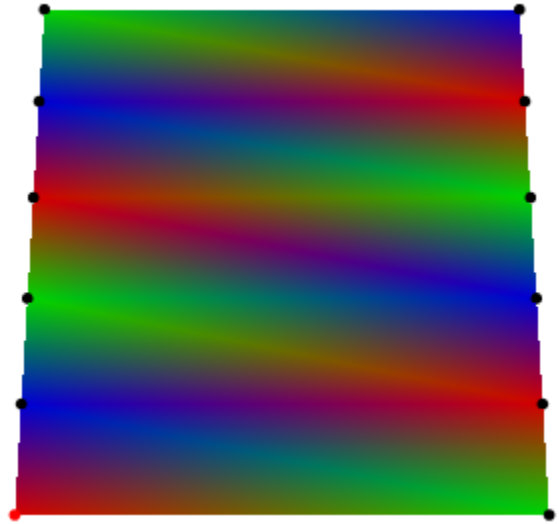
The plot above looks a little bit confusing, thus we let MuPAD plot the first four triangles step by step in order to learn how the whole fan will be created:

```
plot(
  plot::Scene3d(
    plot::SurfaceSet(PL[1..3*n], SO, MeshListType = TriangleFan),
    VO
  ) $ n=3..6
):
```



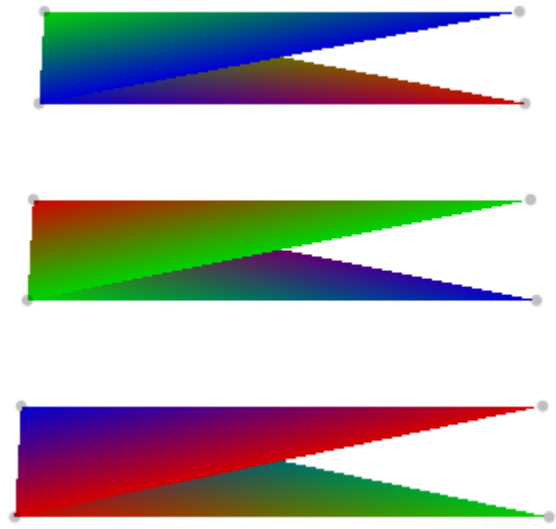
We tell MuPAD to interpret the given mesh list as a triangle strip. The corresponding plot looks like this:

```
plot(plot::SurfaceSet(PL, SO, MeshListType = TriangleStrip), VO):
```



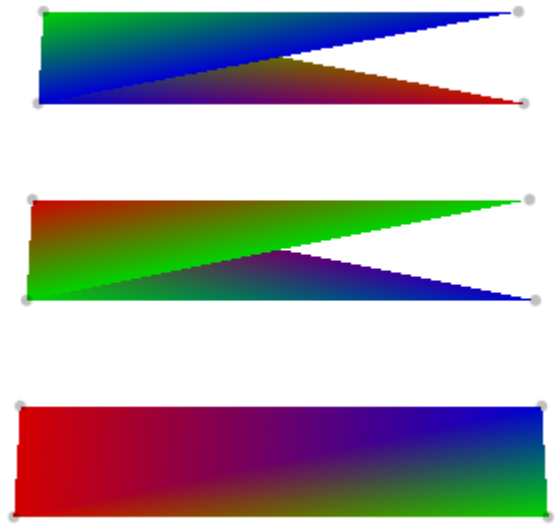
We tell MuPAD to interpret the given mesh list as a set of separate quads. The corresponding plot looks like this:

```
plot(plot::SurfaceSet(PL, SO, MeshListType = Quads), VO):
```



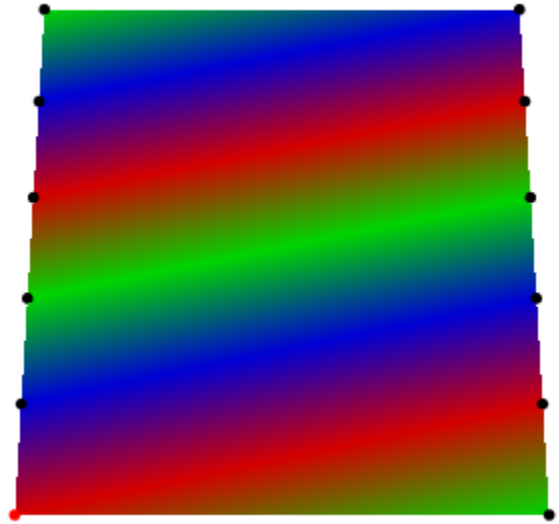
The reason for plotting triangles instead of (the expected) rectangles is the order of the points in the point list. Changing the order of the second and third point, we get the expected result:

```
PK:= PL: tmp:= PK[7]: PK[7]:= PK[10]: PK[10]:=tmp:
plot(plot::SurfaceSet(PK, SO, MeshListType = Quads), VO):
delete PK, tmp:
```



We tell MuPAD to interpret the given mesh list as a quad strip. The corresponding plot looks like this:

```
plot(plot::SurfaceSet(PL, SO, MeshListType = QuadStrip), VO):
delete PL, SO, VO:
```



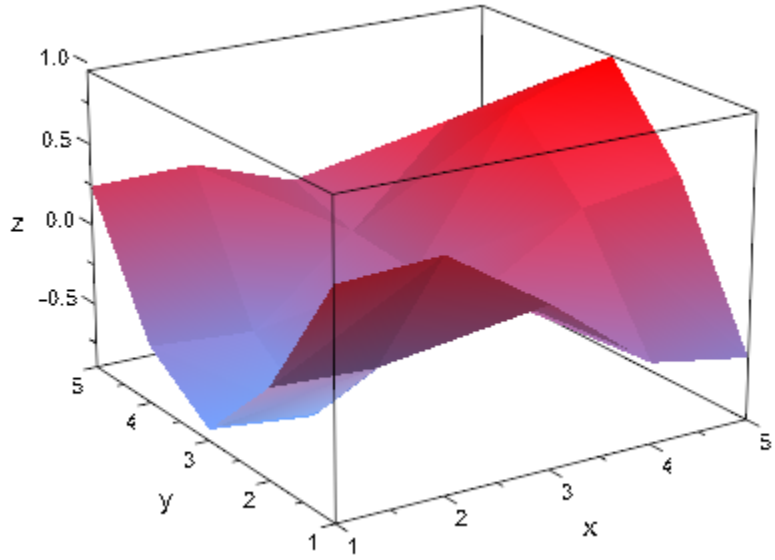
Example 5

It is possible to include normals to give smooth shading for surfaces that are not supposed to look like flat triangles. In the following example, we use a triangulation of a rectangle:

```
trias := [([x,  y   ], [(x+1), y], [ x   ,(y+1)],
          [x, (y+1)], [(x+1), y], [(x+1),(y+1)])
         $ x = 1..4 $ y = 1..4]:
```

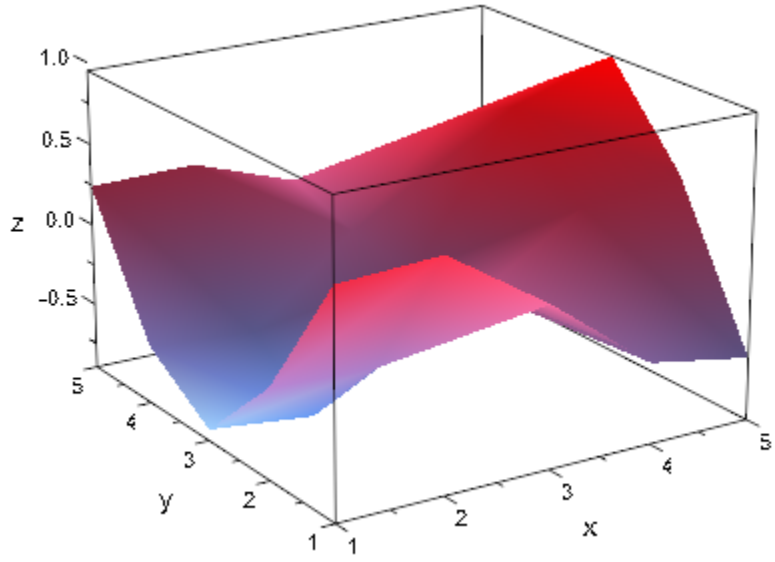
Mapping the function $(x, y) \to \sin(x)\cos(y)$ to these points, we get the following surface plot:

```
f := (x,y) -> sin(x)*cos(y):
meshList := map(trias, l -> [l[1], l[2], f(l[1], l[2])]):
plot(plot::SurfaceSet(meshList, MeshListType = Triangles))
```



The triangulation is clearly visible. One way to reduce this would be to refine the mesh, but this may take a lot of time with more complicated functions or be completely impossible for measurement data. It is much faster to give MuPAD more information on the surface, namely, the direction of the tangent planes at the points we evaluated:

```
normals  := map(trias, l -> [D([1], f)(l[1], l[2]),
                             D([2], f)(l[1], l[2]), 1]):
plot(plot::SurfaceSet(zip(meshList, normals, _exprseq),
                      MeshListType = Triangles,
                      MeshListNormals = BehindPoints))
```



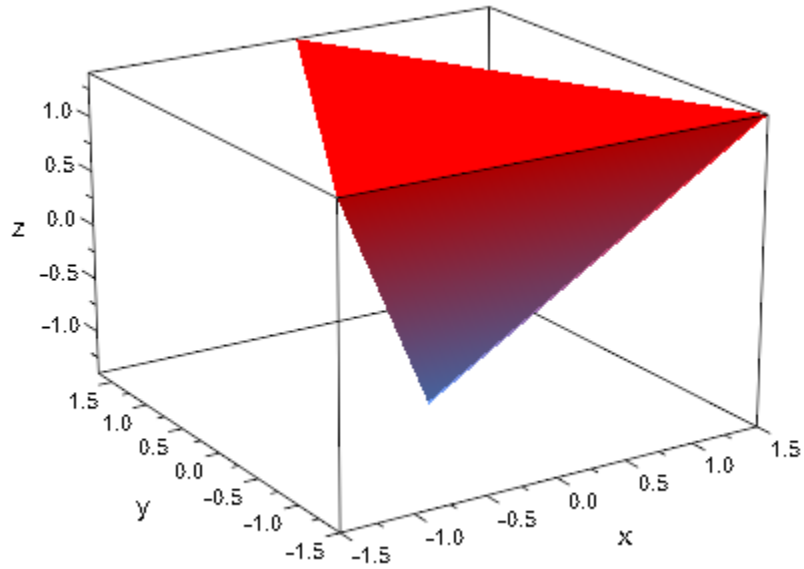
As you can see (especially at the border; otherwise, switch on `LinesVisible`), MuPAD still draws the triangles at exactly the same places, but uses a color shading to create the illusion of a smooth surface.

Example 6

We create a triangle mesh with normals in front of each triangle and plot this object, a tetrahedron, afterwards:

```
meshList:= [
   0.0 ,  0.0 , -1.0 ,
  -1.5 , -1.5 ,  1.4 ,  0.0,  1.7, 1.4, 1.5, -1.5,  1.4,
   0.0 ,  0.88,  0.47,
  -1.5 , -1.5 ,  1.4 ,  1.5, -1.5, 1.4, 0.0,  0.0, -1.4,
  -0.88, -0.41,  0.25,
   1.5 , -1.5 ,  1.4 ,  0.0,  1.7, 1.4, 0.0,  0.0, -1.4,
   0.88, -0.41,  0.25,
   0.0 ,  1.7 ,  1.4 , -1.5, -1.5, 1.4, 0.0,  0.0, -1.4
]:
plot(
  plot::SurfaceSet(meshList,
```

```
        MeshListType    = Triangles,
        MeshListNormals = BeforeFacets
  )
):
```

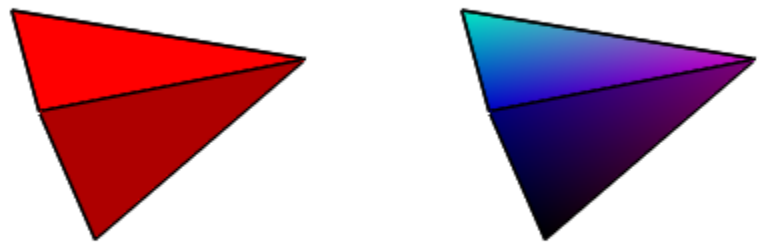


Example 7

A color function `FillColorFunction` can be specified. The procedure is called for each vertex: the parameters are the index of the current triangle followed by the x-, y- and z-coordinate of the current vertex:

```
plot(
  plot::Scene3d(
    plot::SurfaceSet(meshList, MeshListNormals = BeforeFacets,
      MeshVisible = TRUE,
      LineColor    = RGB::Black,
      FillColorFunction =
      (n ->[RGB::Red,RGB::Blue,RGB::Green,RGB::Yellow]
             [n+2 div 3])
    )
  ),
  plot::Scene3d(
```

```
    plot::SurfaceSet(meshList, MeshListNormals = BeforeFacets,
      MeshVisible = TRUE,
      LineColor    = RGB::Black,
      FillColorFunction = ((n,x,y,z) -> [x/2,y/2,z/2])
    )
  ),
  Axes = None, Layout = Horizontal
):
```



The same is true for a `LineColorFunction`:

```
plot(
  plot::Scene3d(
    plot::SurfaceSet(meshList, MeshListNormals = BeforeFacets,
      MeshVisible  = TRUE,
      PointsVisible = TRUE,
      Filled        = FALSE,
      LineWidth     = 2,
      LineColorFunction =
      (n -> [RGB::Red,RGB::Blue,RGB::Green,RGB::Yellow][n+2 div 3])
    )
  ),
  plot::Scene3d(
```

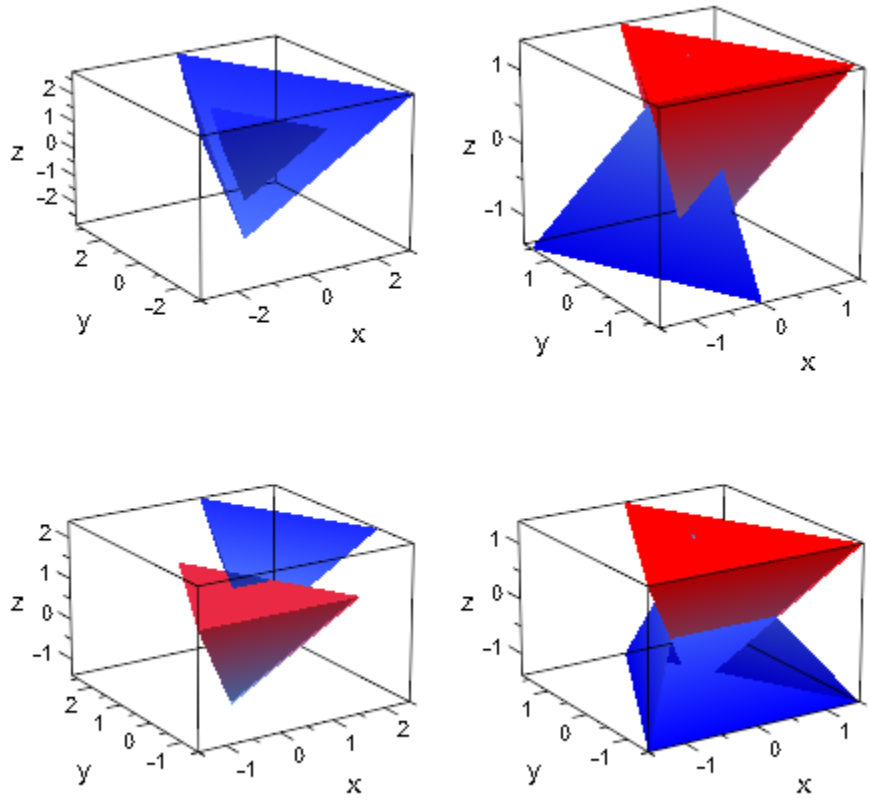
```
    plot::SurfaceSet(meshList, MeshListNormals = BeforeFacets,
      MeshVisible  = TRUE,
      PointsVisible = TRUE,
      Filled        = FALSE,
      LineWidth     = 2,
      LineColorFunction = ((n,x,y,z) -> [x/4,y/4,z/4])
    )
  ),
  Axes = None, Layout = Horizontal
):
```

Example 8

Again we plot the object defined in “Example 6” on page 24-923, but now we add a rotated, scaled and translated copy of it:

```
plot(
  plot::Scene3d(
    plot::SurfaceSet(meshList, MeshListNormals = BeforeFacets),
    plot::Scale3d([2,2,2],
      plot::SurfaceSet(meshList, MeshListNormals = BeforeFacets,
                       Color = RGB::Blue.[0.1])
```

```
    )
  ),
  plot::Scene3d(
    plot::SurfaceSet(meshList, MeshListNormals = BeforeFacets),
    plot::Rotate3d(PI, Axis=[1,0,0],
      plot::SurfaceSet(meshList, MeshListNormals = BeforeFacets,
                       Color = RGB::Blue.[0.1])
    )
  ),
  plot::Scene3d(
    plot::SurfaceSet(meshList, MeshListNormals = BeforeFacets),
    plot::Translate3d([1,1,1],
      plot::SurfaceSet(meshList, MeshListNormals = BeforeFacets,
                       Color = RGB::Blue.[0.1])
    )
  ),
  plot::Scene3d(
    plot::SurfaceSet(meshList, MeshListNormals = BeforeFacets),
    plot::Transform3d([0,0,0], [1,0,0, 0,1,0, 0,0,-1],
      plot::SurfaceSet(meshList, MeshListNormals = BeforeFacets,
                       Color = RGB::Blue.[0.1])
    )
  ),
  Width = 120, Height = 120
):
```



Parameters

`meshlist`

The point list: a list of coordinates of type `DOM_FLOAT`.

meshlist is equivalent to the attribute `MeshList`.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

Options

MeshListType

Option, specified as `MeshListType = t`

`t` may be `Triangles`, `TriangleFan`, `TriangleStrip`, `Quads` or `QuadStrip`. This attribute specifies the kind of surface mesh given in MeshList. This means it specifies how the point coordinates in MeshList are to be interpreted.

MeshListNormals

Option, specified as `MeshListNormals = n`

`n` may be `None`, `BeforePoints`, `BehindPoints`, `BeforeFacets` or `BehindFacets`. This attribute specifies whether MeshList contains normal vectors and at which positions they are located.

UseNormals

Option, specified as `UseNormals = b`

`b` may be `TRUE` or `FALSE`. This attribute specifies whether the normals defined in the STL file are used for the MuPAD plot.

Algorithms

The normal of a facet (a triangle or quad) given in `MeshList` is used for all its vertices when plotting this object. Due to the fact that some facets may share points with other facets, these points may be specified with different normals.

See Also

MuPAD Functions

`plot | readbytes`

MuPAD Graphical Primitives

`plot::Rotate3d` | `plot::Scale3d` | `plot::Surface` | `plot::SurfaceSTL` | `plot::Transform3d` | `plot::Translate3d`

plot::SurfaceSTL

Import of STL graphics files

Syntax

```
plot::SurfaceSTL(filename, <UseNormals = b>, <a = a_min .. a_max>, options)
```

Description

`plot::SurfaceSTL(filename)` creates a 3D surface object from the data of a given STL graphics file named "`filename`".

Stereolithography (STL) files were introduced in software by 3D Systems of Valencia, CA, as a simple method of storing information about 3D objects.

STL files contain triangulation data of 3D surfaces. Each triangle is stored as a unit normal and three vertices. The normal and the vertices are specified by three coordinates each, so there is a total of 12 numbers stored for each triangle. Read the 'Background' section of this help page for further details.

Depending on your hardware we recommend to plot STL objects with no more than 50.000 to 150.000 facets (triangles). You should activate the option 'Accelerate OpenGL®' in the VCam options menu.

`plot::SurfaceSTL` reacts to the MuPAD environment variable `READPATH`. For example, after

```
>> READPATH := READPATH, "C:\\STLFILES":
```

the file '`C:\STLFILES\xyz.stl`' is found by the command

```
>> S := plot::SurfaceSTL("xyz.stl"):
```

Alternatively, the file name can be specified as an absolute pathname:

```
>> S := plot::SurfaceSTL("C:\\STLFILES\\xyz.stl"):
```

If a MuPAD notebook was saved to a file, its location is available inside the notebook as the environment variable NOTEBOOKPATH. If your STL file is in the same folder as the notebook, you may call

```
>> S := plot::SurfaceSTL(NOTEBOOKPATH."xyz.stl"):
```

When setting the attribute UseNormals to FALSE, the normals defined in the STL graphics file are ignored when plotting the object in MuPAD. This reduces the data volume of the graphics object in the MuPAD session and improves the computing time as well. However, it leads to a slightly less brilliant image. Cf. “Example 2” on page 24-936.

The STL data do not include any color information. Hence, the imported graphics reacts to the usual settings of FillColor, FillColorType etc. for MuPAD surfaces.

Also user-defined color functions LineColorFunction and FillColorFunction can be used to color the imported surface. These functions are called with the index of the current triangle as its first parameter, followed by the *x*, *y*, and *z* coordinate of the current point.

The transformation objects plot::Rotate3d, plot::Scale3d, plot::Translate3d and plot::Transform3d can be applied to the imported STL object. Cf. “Example 7” on page 24-945.

If an object of type plot::SurfaceSTL is to be plotted together with other objects, one needs to know the coordinates of the surface objects. To this end, an object S := plot::SurfaceSTL(...) provides the methods S::center and S::boundingBox.

The call S::center() returns a list of 3 floating-point values representing the 3D center of the STL object.

The call S::boundingBox() returns a list of 3 ranges of floating-point values representing the ranges for the *x*, *y*, and *z* coordinates of the STL surface.

See “Example 2” on page 24-936 and “Example 5” on page 24-942.

plot::SurfaceSTL::center(S) and plot::SurfaceSTL::boundingBox(S), respectively, are alternative calls.

Note that the STL graphics file must be read completely for computing these data. Also note that after a first call to S::center() or S::boundingBox(), the data are not recomputed by these functions even if the STL object S has changed. Use

`plot::SurfaceSTL::center(S)`, `plot::SurfaceSTL::boundingBox(S)` in such a case.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Color`	the main color	`RGB::Red`
`Filled`	filled or transparent areas and surfaces	`TRUE`
`FillColor`	color of areas and surfaces	`RGB::Red`
`FillColor2`	second color of areas and surfaces for color blends	`RGB::CornflowerBlue`
`FillColorType`	surface filling types	`Dichromatic`
`FillColorFunction`	functional area/surface coloring	
`FillColorDirection`	the direction of color transitions on surfaces	[0, 0, 1]
`FillColorDirectionX`	x-component of the direction of color transitions on surfaces	0
`FillColorDirectionY`	y-component of the direction of color transitions on surfaces	0
`FillColorDirectionZ`	z-component of the direction of color transitions on surfaces	1
`Frames`	the number of frames in an animation	50
`InputFile`	input file for import functions	
`Legend`	makes a legend entry	

Attribute	Purpose	Default Value
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Black.[0.25]
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	0
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
MeshVisible	visibility of irregular mesh lines in 3D	FALSE
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	

Attribute	Purpose	Default Value
`PointSize`	the size of points	`1.5`
`PointStyle`	the presentation style of points	`FilledCircles`
`PointsVisible`	visibility of mesh points	`FALSE`
`Shading`	smooth color blend of surfaces	`Smooth`
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`
`Title`	object title	
`TitleFont`	font of object titles	`[" sans-serif ", 11]`
`TitlePosition`	position of object titles	
`TitleAlignment`	horizontal alignment of titles w.r.t. their coordinates	`Center`
`TitlePositionX`	position of object titles, x component	
`TitlePositionY`	position of object titles, y component	
`TitlePositionZ`	position of object titles, z component	
`UseNormals`	use pre-defined normals?	`TRUE`
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`

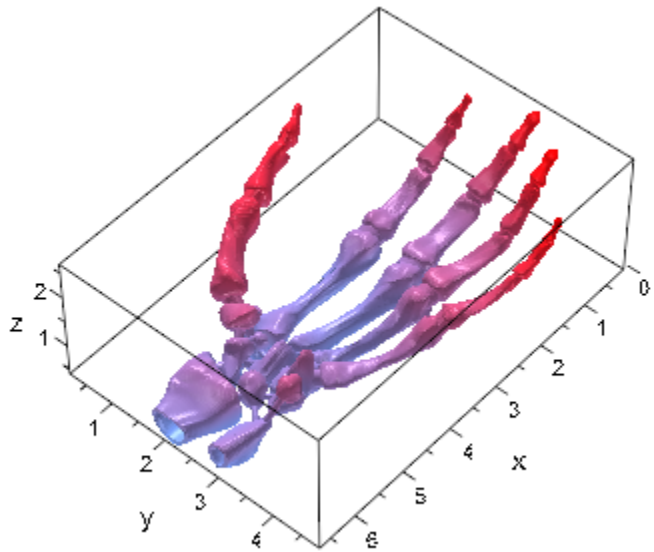
Attribute	Purpose	Default Value
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

The following imported STL graphics consists of about 110.000 triangles:

```
plot(plot::SurfaceSTL("hand.stl"),
     CameraDirection = [15, 13, 22])
```

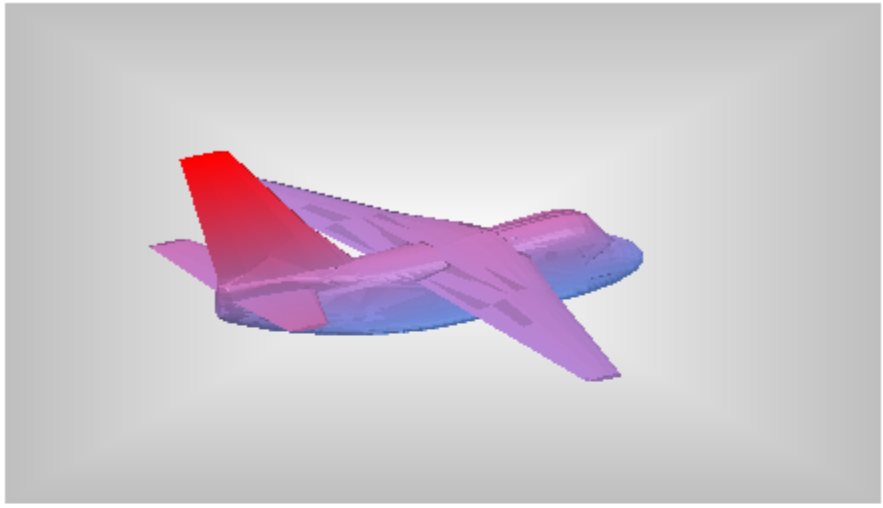


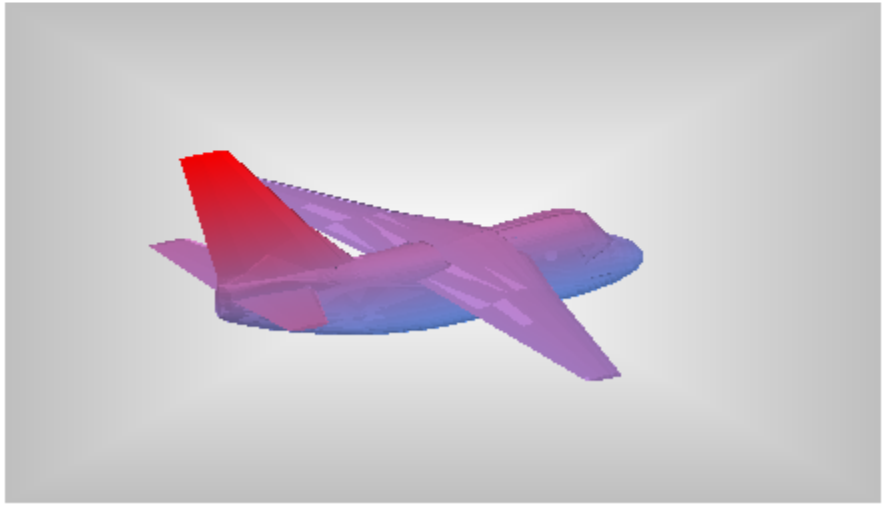
Example 2

By default, the normals defined in an STL graphics file are used when plotting the object in MuPAD. Suppressing the use of these normals may reduce the data volume of the

graphical object and speed up plotting. However, in general, this leads to slightly less brilliant images. For comparison, the following STL graphics is plotted with and without using its normals:

```
S1 := plot::SurfaceSTL("skin.stl"):
S2 := plot::SurfaceSTL("skin.stl", UseNormals = FALSE):
plot(plot::Scene3d(S1), plot::Scene3d(S2), Layout = Vertical,
     Width = 120*unit::mm, Height = 140*unit::mm,
     Axes = None, BackgroundStyle = Pyramid):
```





We compute the center and the bounding box of the surface:

```
S1::center()
```

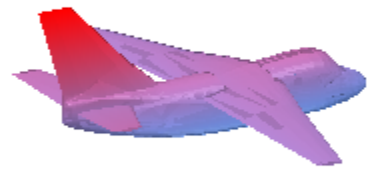
```
[-422.035, 0.0, 282.1]
```

```
S1::boundingBox()
```

$$[-738.07..-106.0,\ -401.5..401.5,\ 156.0..408.2]$$

We rotate the object around its center:

```
plot(plot::Rotate3d(a, S1::center(), [0, 0, 1], S1,
                    a = 0..2*PI), Axes = None)
```

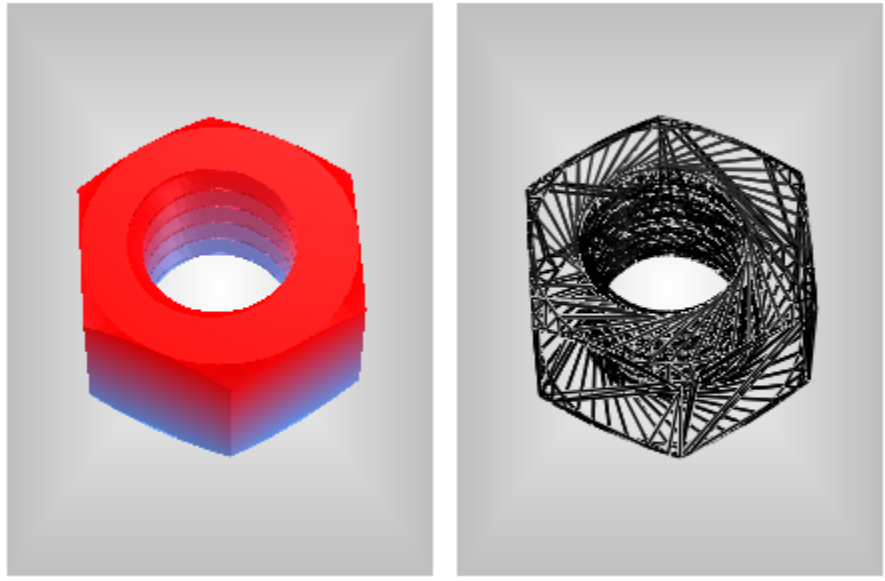


```
delete S1, S2:
```

Example 3

The following STL graphics is displayed as a surface model and as a wireframe model:

```
nut := plot::SurfaceSTL("nut.stl"):
plot(plot::Scene3d(nut, CameraDirection = [10, 15, 30]),
     plot::Scene3d(nut, CameraDirection = [10, 15, 30],
                   MeshVisible = TRUE, Filled = FALSE,
                   LineColor = RGB::Black),
     Axes = None, Layout = Horizontal,
     BackgroundStyle = Pyramid):
```



```
delete nut:
```

Example 4

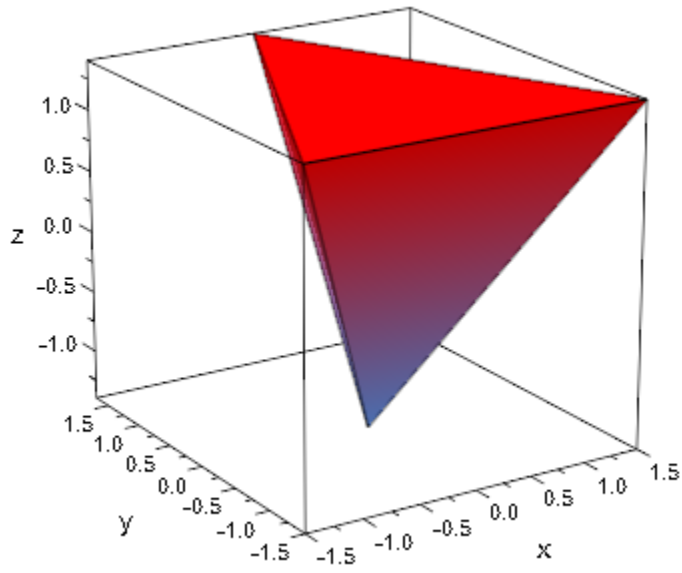
For demonstrating further features of STL file import, we first create our own STL graphics file which defines a tetrahedron:

```
stlFile := "demo.stl":

fprint(Unquoted, Text, stlFile,
        "SOLID TRI
         FACET NORMAL 0.0 0.0 -1.0
           OUTER LOOP
             VERTEX -1.5 -1.5 1.4
             VERTEX 0.0 1.7 1.4
             VERTEX 1.5 -1.5 1.4
           ENDLOOP
         ENDFACET
         FACET NORMAL 0.0 0.88148 0.472221
           OUTER LOOP
             VERTEX -1.5 -1.5 1.4
             VERTEX 1.5 -1.5 1.4
```

```
    VERTEX 0.0 0.0 -1.4
  ENDLOOP
ENDFACET
FACET NORMAL -0.876814 -0.411007 0.24954
  OUTER LOOP
    VERTEX 1.5 -1.5 1.4
    VERTEX 0.0 1.7 1.4
    VERTEX 0.0 0.0 -1.4
  ENDLOOP
ENDFACET
FACET NORMAL 0.876814 -0.411007 0.24954
  OUTER LOOP
    VERTEX 0.0 1.7 1.4
    VERTEX -1.5 -1.5 1.4
    VERTEX 0.0 0.0 -1.4
  ENDLOOP
ENDFACET
ENDSOLID TRI"
)
```

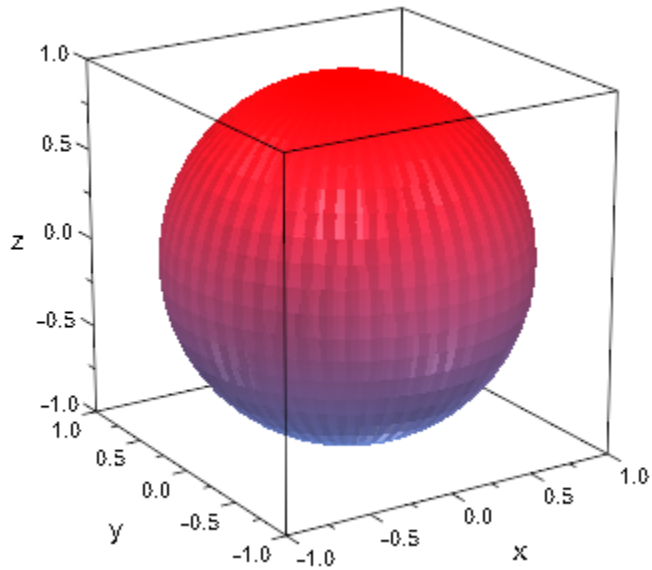
This STL graphics file is imported as a MuPAD plot object and rendered:

```
plot(plot::SurfaceSTL(stlFile, MeshVisible = TRUE)):
```



We create another STL file using `export::stl`. It contains a sphere of radius 1 parametrized by spherical coordinates:

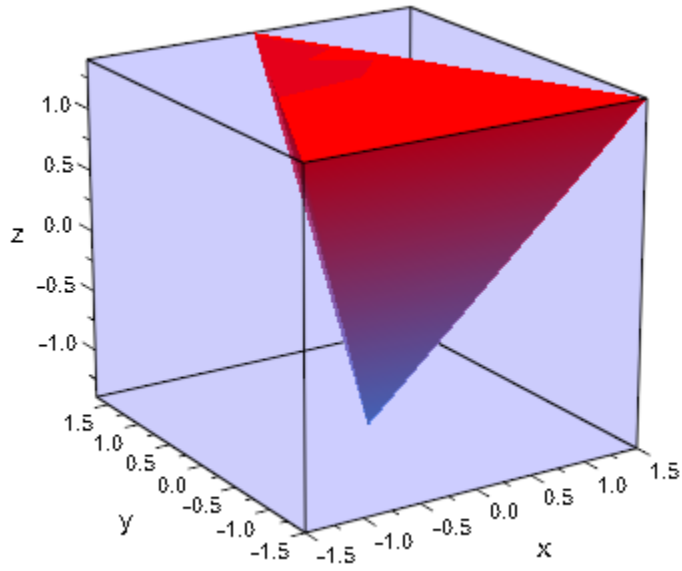
```
export::stl("sphere.stl", [cos(u)*sin(v), sin(u)*sin(v), cos(v)],
                                  u = 0 .. 2*PI, v = 0 .. 2*PI,
                          Mesh = [50, 50], Scaling = Constrained,
                          OutputBox = [-1 .. 1, -1 .. 1, -1 .. 1]):

plot(plot::SurfaceSTL("sphere.stl", Scaling = Constrained)):
```



Example 5

We plot the object defined in the STL graphics file of "Example 4" on page 24-940 with its bounding box:

```
S := plot::SurfaceSTL(stlFile):
plot(S, plot::Box(op(S::boundingBox()),
                  Color = RGB::Blue.[0.1])):
```




```
delete S:
```

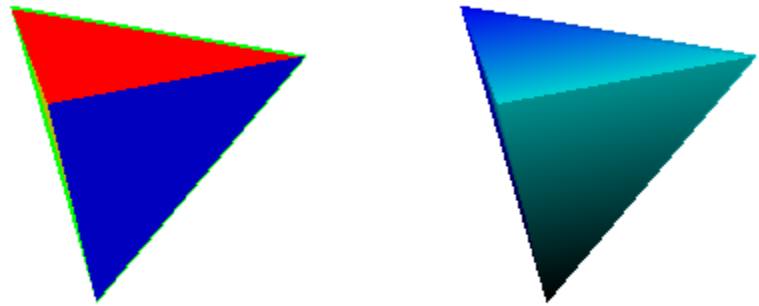
Example 6

A color function `FillColorFunction` can be specified. This will be called with the index of the current facet as its first parameter followed by the x-, y- and z-coordinate of the current point.

We use the object defined in the STL graphics file of “Example 4” on page 24-940:

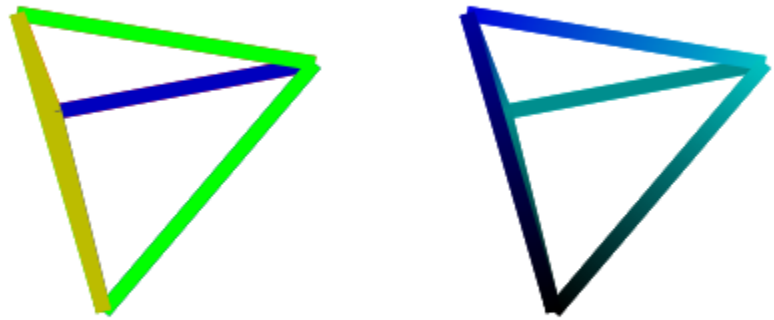
```
mycolorlist:= [RGB::Red, RGB::Blue, RGB::Green, RGB::Yellow]:
plot(plot::Scene3d(plot::SurfaceSTL(stlFile,
                      FillColorFunction =
                         proc(n, x, y, z) begin
                            mycolorlist[n]
                         end_proc)),
     plot::Scene3d(plot::SurfaceSTL(stlFile,
                      FillColorFunction =
                         proc(n, x, y, z) begin
                            [abs(x)/2, abs(y)/2, abs(z)/2]
```

```
                          end_proc)),
  Axes = None, Layout = Horizontal):
```



We define a `LineColorFunction`:

```
plot(plot::Scene3d(plot::SurfaceSTL(stlFile,
                       LineColorFunction =
                          proc(n, x, y, z) begin
                             mycolorlist[n]
                          end_proc)),
     plot::Scene3d(plot::SurfaceSTL(stlFile,
                       LineColorFunction =
                          proc(n, x, y, z) begin
                             [abs(x)/2, abs(y)/2, abs(z)/2]
                          end_proc)),
  Axes = None, Filled = FALSE, MeshVisible = TRUE,
  LineWidth = 2*unit::mm, Layout = Horizontal):
```



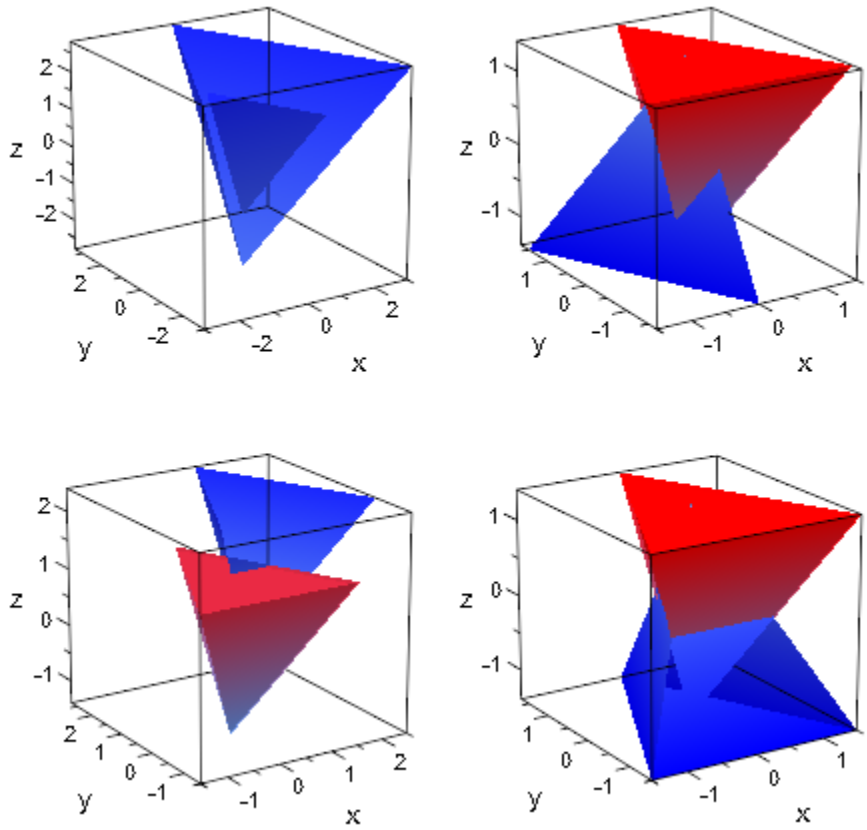
```
delete mycolorlist:
```

Example 7

Again, we plot the object of the STL graphics file defined in “Example 4” on page 24-940. Here, we add rotated, scaled and translated copies:

```
plot(
  plot::Scene3d(
    plot::SurfaceSTL(stlFile),
    plot::Scale3d([2, 2, 2],
      plot::SurfaceSTL(stlFile, Color = RGB::Blue.[0.1])
    )
  ),
  plot::Scene3d(
    plot::SurfaceSTL(stlFile),
    plot::Rotate3d(PI, Axis = [1, 0, 0],
      plot::SurfaceSTL(stlFile, Color = RGB::Blue.[0.1])
    )
  ),
  plot::Scene3d(
```

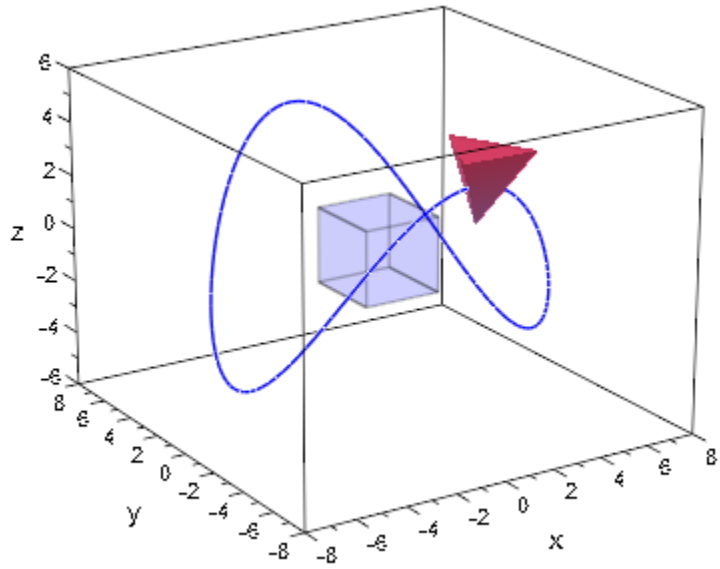
```
  plot::SurfaceSTL(stlFile),
  plot::Translate3d([1, 1, 1],
    plot::SurfaceSTL(stlFile, Color = RGB::Blue.[0.1])
  )
),
plot::Scene3d(
  plot::SurfaceSTL(stlFile),
  plot::Transform3d([0, 0, 0], [1, 0, 0, 0, 1, 0, 0, 0, -1],
    plot::SurfaceSTL(stlFile, Color = RGB::Blue.[0.1])
  )
),
Width = 120*unit::mm, Height = 120*unit::mm):
```



Example 8

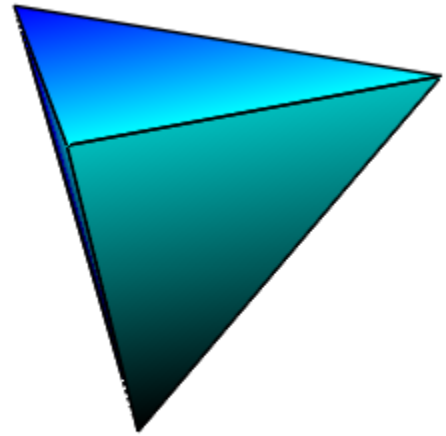
STL objects can be animated. The tetrahedron defined in “Example 4” on page 24-940 moves around a box:

```
SO := plot::SurfaceSTL(stlFile):
BO := plot::Box(op(SO::boundingBox(SO)), Color = RGB::Blue.[0.1]):
GO := [6*sin(a), -6*cos(a), 4*cos(2*a)], a = 0..2*PI:
CU := plot::Curve3d(GO):
plot(BO, CU, plot::Translate3d(GO, SO),
     ViewingBox = [-8..8, -8..8, -6..6]):
```



Below, the color function `FillColorFunction` of an STL object is animated:

```
plot(plot::SurfaceSTL(stlFile,
        MeshVisible = TRUE,
        LineColor = RGB::Black,
        FillColorFunction =
            proc(n, x, y, z) begin
              [sin(x + a)^2 ,sin(y + a)^2, sin(z + a)^2]
            end_proc,
        a = 0..2*PI, TimeRange = 1..4),
  Axes = None, Layout = Horizontal)
```



```
delete SO, BO, GO, CU, stlFile:
```

Parameters

`filename`

The file name: a character string of type `DOM_STRING`.

filename is equivalent to the attribute `InputFile`.

`a`

Animation parameter, specified as a `= a`$_{min}$`..a`$_{max}$, where `a`$_{min}$ is the initial parameter value, and `a`$_{max}$ is the final parameter value.

Options

`UseNormals`

Option, specified as `UseNormals = b`

b may be TRUE or FALSE. This attribute specifies whether the normals defined in the STL file are used for the MuPAD plot.

Algorithms

The STL sample files presented on this help page were taken from the ftp site of the Clemson University, South Carolina, USA:

ftp.vr.clemson.edu/pub/rp/STL_objects.

There are two storage formats available for STL files, which are ASCII and BINARY. ASCII files are human-readable while BINARY files are smaller and faster to process. Both formats can be read by `plot::SurfaceSTL`. A typical ASCII STL file looks like this:

```
solid sample
 facet normal -4.470293E-02 7.003503E-01 -7.123981E-01
  outer loop
   vertex -2.812284E+00 2.298693E+01 0.000000E+00
   vertex -2.812284E+00 2.296699E+01 -1.960784E-02
   vertex -3.124760E+00 2.296699E+01 0.000000E+00
  endloop
 endfacet
 ...
endsolid sample
```

STL BINARY files have the following format:

```
Bytes  Type   Description
80     ASCII  header, no data significance
4      uint   number of facets in file

4      float  normal  x - start of facet
4      float  normal  y
4      float  normal  z
4      float  vertex1 x
4      float  vertex1 y
4      float  vertex1 z
4      float  vertex2 x
4      float  vertex2 y
4      float  vertex2 z
```

```
4      float  vertex3 x
4      float  vertex3 y
4      float  vertex3 z
2      byte   not used  - end of facet
       ...
```

Facet orientation: The facets define the surface of a 3D object. As such, each facet is part of the boundary between the interior and the exterior of the object. The orientation of the facets (which way is "out" and which way is "in") is specified redundantly in two ways which should be consistent. First, the direction of the normal is outward. Second, which is most commonly used nowadays, the facet vertices are listed in counterclockwise order when looking at the object from the outside (right-hand rule).

Vertex-to-vertex rule: Each triangle must share two vertices with each of its adjacent triangles. In other words, a vertex of one triangle cannot lie on the side of another.

Axes: The format specifies that all vertex coordinates must be strictly positive numbers. However, it seems that — with a few exceptions — most software used today (MuPAD included) allow negative coordinates as well.

Units: The STL file does not contain any scale information; the coordinates may be interpreted in arbitrary units.

Further details about the STL file format are available in the web, e.g., at:

- www.ennex.com/fabbers/StL.asp,
- www.math.iastate.edu/burkardt/data/stl/stl.html and
- rpdrc.ic.polyu.edu.hk/content/stl/stl_introduction.htm.

Collections of STL sample files can be found in the web, e.g., at:

- www.wohlersassociates.com/Software-for-Rapid-Prototyping.html and
- www.cs.duke.edu/~edels/Tubes.

Information about rapid prototyping technologies is available in the web, e.g., at:

www.cs.hut.fi/~ado/rp/rp.html.

Note that MuPAD only accepts the following notations for the keywords “`facet`” and “`vertex`” in STL ASCII files: `facet`, `FACET`, `Facet` and `vertex`, `VERTEX`, `Vertex`, respectively.

The normal of a facet defined in an STL file is used for all its vertices when plotting this object. Due to the fact that some facets (triangles) share points with other ones, these points are plotted with different normals.

See Also

MuPAD Functions

`export::stl` | `import::readbitmap` | `plot`

MuPAD Graphical Primitives

`plot::Rotate3d` | `plot::Scale3d` | `plot::Surface` | `plot::SurfaceSet` | `plot::Transform3d` | `plot::Translate3d`

plot::Sweep

Sweep surface from the deformation of a 3D curve

Syntax

```
plot::Sweep([x1, y1, z1], <Ground = g>, u = umin .. umax, <a = amin .. amax>, options)
plot::Sweep(A1, <Ground = g>, u = umin .. umax, <a = amin .. amax>, options)
plot::Sweep(C1, <Ground = g>, options)
plot::Sweep([x1, y1, z1], [x2, y2, z2], u = umin .. umax, <a = amin .. amax>, options)
plot::Sweep(A1, A2, u = umin .. umax, <a = amin .. amax>, options)
plot::Sweep(C1, C2, options)
```

Description

`plot::Sweep([x_1(u), y_1(u), z_1(u)], u = `u_{min}`..`u_{max}`)` creates the surface swept out by the (linear) deformation of the parameterized curve $(x_1(u), y_1(u), z_1(u))$ to its projection $(x_1(u), y_1(u), 0)$ to the x-y-plane.

`plot::Sweep([x_1(u), y_1(u), z_1(u)], [x_2(u), y_2(u), z_2(u)], u = `u_{min}`..`u_{max}`)` creates the surface swept out by the (linear) deformation of the parameterized curve $(x_1(u), y_1(u), z_1(u))$ to the parameterized curve $(x_2(u), y_2(u), z_2(u))$.

`plot::Sweep` creates the parametrized surface

$$
\begin{aligned}
x(u, v) &= x_1(u) + v\,(x_2(u) - x_1(u)) \\
y(u, v) &= y_1(u) + v\,(y_2(u) - y_1(u)) \\
z(u, v) &= z_1(u) + v\,(z_2(u) - z_1(u))
\end{aligned}
$$

with the two surface parameters u (ranging from u_{min} to u_{max}) and v (ranging from 0 to 1). This is the linear deformation of the curve $(x_1(u), y_1(u), z_1(u))$ defining one border of the surface to the curve $(x_2(u), y_2(u), z_2(u))$ defining the other border of the surface.

If no "target curve" $(x_2(u), y_2(u), z_2(u))$ is specified, the projection $x_2(u) = x_1(u)$, $y_2(u) = y_1(u)$, $z_2(u) = g$ of the "source curve" $(x_1(u), y_1(u), z_1(u))$ to the x-y-plane with constant value $z = g$ is used. The value g is set by the attribute `Ground` = `g`. The default value is `g = 0`.

When a target curve $[x_2(u), y_2(u), z_2(u)]$ is specified, the `Ground` attribute does not have any effect.

If the curves are specified by objects C_1, C_2 of type `plot::Curve3d`, the graphical attributes of the object created by `plot::Sweep` are copied from C_1. The parametrization of C_2 is automatically rewritten in terms of the curve parameter used in the definition of C_1. This, however, will only work if the parametrization of C_2 is defined by symbolic expressions.

Note: If the parametrization of C_2 is defined by procedures, make sure that the parameter ranges of C_1 and C_2 coincide!

Attributes

Attribute	Purpose	Default Value
`AdaptiveMesh`	adaptive sampling	`0`
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Color`	the main color	`RGB::Black.[0.25]`
`DiscontinuitySearch`	semi-symbolic search for discontinuities	`TRUE`
`Filled`	filled or transparent areas and surfaces	`TRUE`
`FillColor`	color of areas and surfaces	`RGB::Red`
`FillColor2`	second color of areas and surfaces for color blends	`RGB::CornflowerBlue`
`FillColorType`	surface filling types	`Dichromatic`
`FillColorFunction`	functional area/surface coloring	

Attribute	Purpose	Default Value
FillColorDirection	the direction of color transitions on surfaces	[0, 0, 1]
FillColorDirectionX	x-component of the direction of color transitions on surfaces	0
FillColorDirectionY	y-component of the direction of color transitions on surfaces	0
FillColorDirectionZ	z-component of the direction of color transitions on surfaces	1
Frames	the number of frames in an animation	50
Ground	base value	0
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	TRUE
LineColor	color of lines	RGB::Black.[0.25]
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0

Attribute	Purpose	Default Value
LineColorDirectionY	y-component of the direction of color transitions on lines	0
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Mesh	number of sample points	25
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE
Submesh	density of submesh (additional sample points)	4
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center

Attribute	Purpose	Default Value
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
ULinesVisible	visibility of parameter lines (u lines)	TRUE
UMax	final value of parameter “u”	
UMesh	number of sample points for parameter “u”	25
UMin	initial value of parameter “u”	
UName	name of parameter “u”	
URange	range of parameter “u”	
USubmesh	density of additional sample points for parameter “u”	4
VLinesVisible	visibility of parameter lines (v lines)	TRUE
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XFunction1	parametrization of the curves in sweep surfaces	

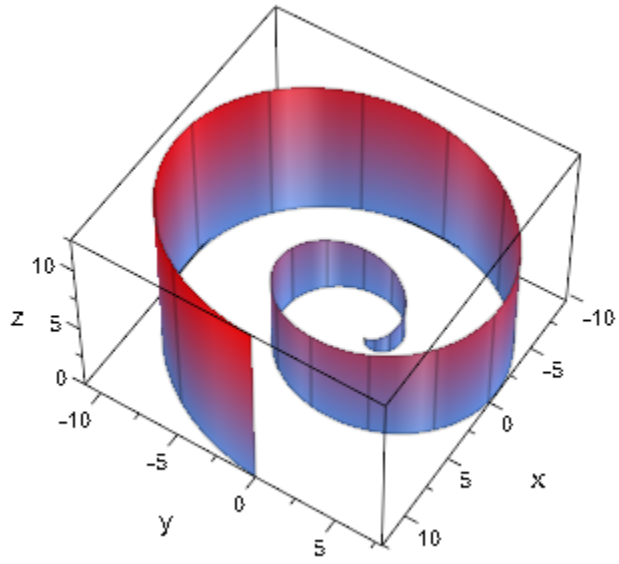
Attribute	Purpose	Default Value
XFunction2	parametrization of the curves in sweep surfaces	
YFunction1	parametrization of the curves in sweep surfaces	
YFunction2	parametrization of the curves in sweep surfaces	
ZFunction1	parametrization of the curves in sweep surfaces	
ZFunction2	parametrization of the curves in sweep surfaces	

Examples

Example 1

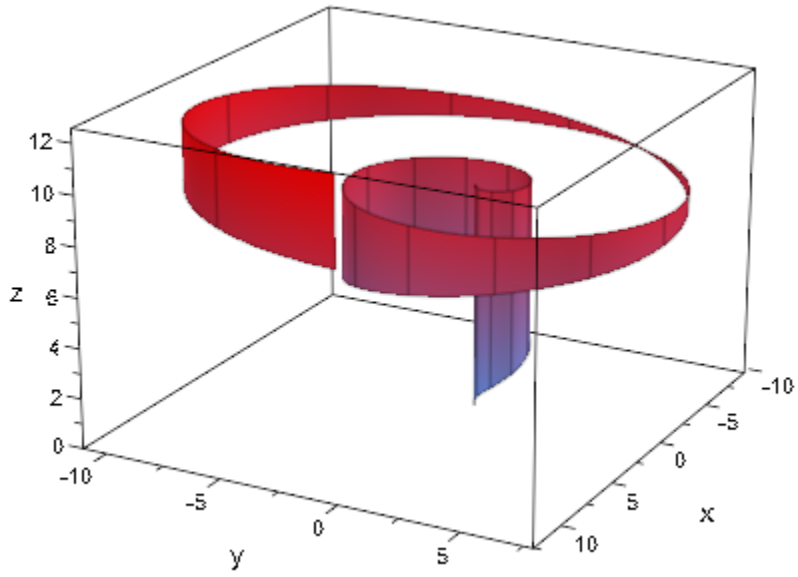
We deform a 3D spiral to its projection to the *x*-*y*-plane:

```
plot(plot::Sweep([u*cos(u), u*sin(u), u], u = 0..4*PI),
     CameraDirection = [90, 50, 120])
```



We use the `Ground` attribute to project the spiral to the x-y-plane with $z = 9$:

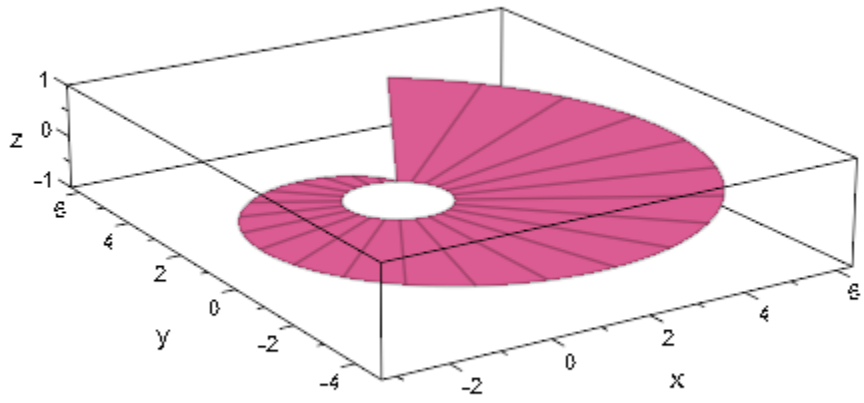
```
plot(plot::Sweep([u*cos(u), u*sin(u), u], u = 0..4*PI, Ground = 9),
     CameraDirection = [130, 60, 45])
```



Example 2

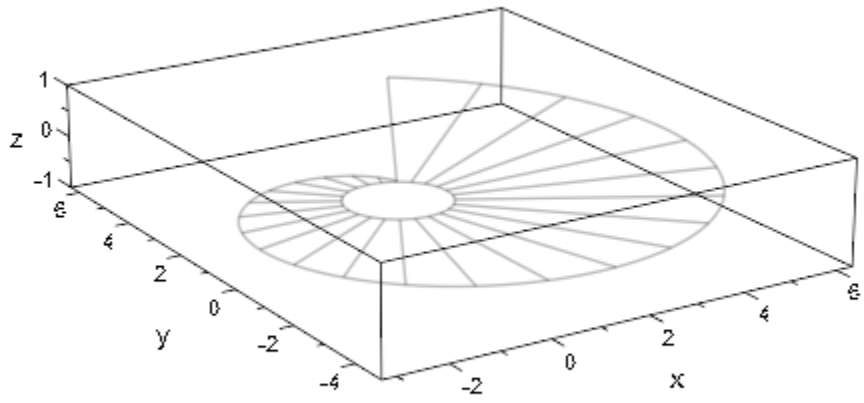
We deform a circle in the x-y-plane to a planar spiral:

```
plot(plot::Sweep([cos(u), sin(u), 0], [u*cos(u), u*sin(u), 0],
                 u = PI/3..7/3*PI), Scaling = Constrained)
```



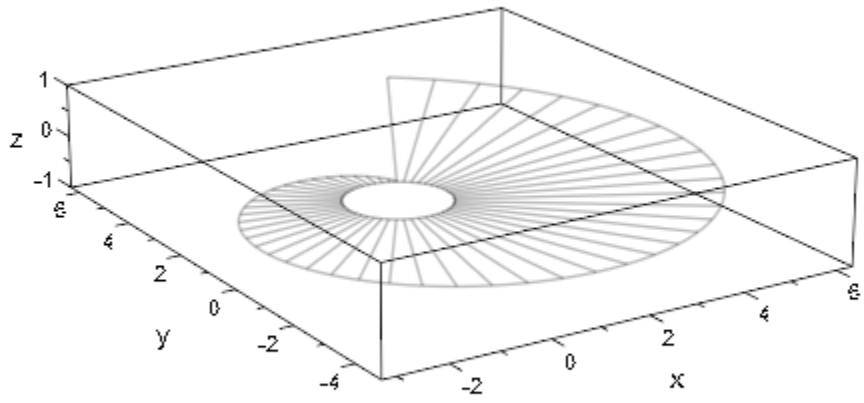
With `Filled` = `FALSE`, only the lines are visible along which the mesh points of the curves are moved:

```
plot(plot::Sweep([cos(u), sin(u), 0], [u*cos(u), u*sin(u), 0],
                 u = PI/3..7/3*PI), Scaling = Constrained,
                 Filled = FALSE)
```



We increase the number of mesh points:

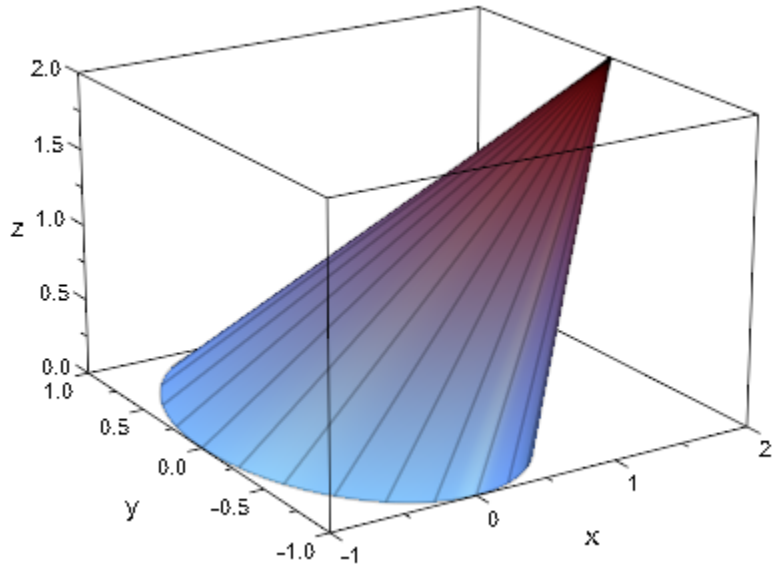
```
plot(plot::Sweep([cos(u), sin(u), 0], [u*cos(u), u*sin(u), 0],
                 u = PI/3..7/3*PI, Mesh = 50),
                 Scaling = Constrained, Filled = FALSE)
```



Example 3

We deform a circle to an animated point. The resulting sweep surface is an animated cone:

```
plot(plot::Sweep([cos(u), sin(u), 0], [a, 0, a],
                 u = 0..2*PI, a = 0..2))
```



Parameters

x_1, y_1, z_1

The parametrization of the initial 3D curve: real-valued expressions in `u` (and possibly the animation parameter).

x_1, y_1, z_1 are equivalent to the attributes `XFunction1`, `YFunction1`, `ZFunction1`.

x_2, y_2, z_2

The parametrization of the "target curve": real-valued expressions in `u` (and possibly the animation parameter).

x_2, y_2, z_2 are equivalent to the attributes `XFunction2`, `YFunction2`, `ZFunction2`.

u

The curve parameter: an identifier or an indexed identifier.

u is equivalent to the attribute `UName`.

u_{min}, u_{max}

Real-valued expressions (possibly in the animation parameter).

u_{min}, u_{max} are equivalent to the attributes `UMin`, `UMax`.

g

Real-valued expression (possibly in the animation parameter).

g is equivalent to the attribute `Ground`.

A_1, A_2

matrices of category `Cat::Matrix` with three entries that provide the parametrizations x_1, y_1, z_1 and x_2, y_2, z_2, respectively.

C_1, C_2

Curves of type `plot::Curve3d`. C_1 provides the "initial curve" $[x_1, y_1, z_1]$, C_2 provides the "target curve" $[x_2, y_2, z_2]$.

a

Animation parameter, specified as $a = a_{min}..a_{max}$, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Curve3d` | `plot::Hatch` | `plot::Polygon3d` | `plot::Surface`

plot::Tetrahedron

Regular Tetrahedra

Syntax

```
plot::Tetrahedron(<a = a_min .. a_max>, options)
```

Description

`plot::Tetrahedron()` creates regular polyhedra.

Per default, all polyhedra are centered at the origin. The attribute `Center` allows to choose a different center. This is helpful to align the polyhedra relative to other objects in the graphical scene. Cf. “Example 1” on page 24-969.

All polyhedra fit into a box extending from -1 to 1 in all coordinate directions. Their size can be changed by the attribute `Radius`. In case of a hexahedron (a box), this attribute represents the radius of the inscribed sphere. For the other polyhedra, it is the radius of the circumscribed sphere.

The default value of `Radius` is 1 for all polyhedra.

Further to the attributes `Center` and `Radius`, you can modify the polyhedra by applying transformation objects of type `plot::Rotate3d`, `plot::Scale3d`, `plot::Translate3d` , and `plot::Transform3d`. Cf. “Example 3” on page 24-971.

User-defined color functions (`LineColorFunction`, `FillColorFunction`) are called with the index of the current facet as its first parameter, followed by the *x*, *y*, and *z* coordinate of the current point, followed by the current value of the animation parameter (if animated). Cf. “Example 4” on page 24-972.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`

Attribute	Purpose	Default Value
`Center`	center of objects, rotation center	[0, 0, 0]
`CenterX`	center of objects, rotation center, x-component	0
`CenterY`	center of objects, rotation center, y-component	0
`CenterZ`	center of objects, rotation center, z-component	0
`Color`	the main color	`RGB::Red`
`Filled`	filled or transparent areas and surfaces	`TRUE`
`FillColor`	color of areas and surfaces	`RGB::Red`
`FillColor2`	second color of areas and surfaces for color blends	`RGB::CornflowerBlue`
`FillColorType`	surface filling types	`Dichromatic`
`FillColorFunction`	functional area/surface coloring	
`FillColorDirection`	the direction of color transitions on surfaces	[0, 0, 1]
`FillColorDirectionX`	x-component of the direction of color transitions on surfaces	0
`FillColorDirectionY`	y-component of the direction of color transitions on surfaces	0
`FillColorDirectionZ`	z-component of the direction of color transitions on surfaces	1
`Frames`	the number of frames in an animation	50
`Legend`	makes a legend entry	

Attribute	Purpose	Default Value
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`LineColor`	color of lines	`RGB::Black.[0.25]`
`LineWidth`	width of lines	`0.35`
`LineColor2`	color of lines	`RGB::DeepPink`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LinesVisible`	visibility of lines	`TRUE`
`LineColorType`	line coloring types	`Flat`
`LineColorFunction`	functional line coloring	
`LineColorDirection`	the direction of color transitions on lines	[0, 0, 1]
`LineColorDirectionX`	x-component of the direction of color transitions on lines	`0`
`LineColorDirectionY`	y-component of the direction of color transitions on lines	`0`
`LineColorDirectionZ`	z-component of the direction of color transitions on lines	`1`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`PointSize`	the size of points	`1.5`

Attribute	Purpose	Default Value
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE
Radius	radius of circles, spheres etc.	1
Shading	smooth color blend of surfaces	Smooth
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE

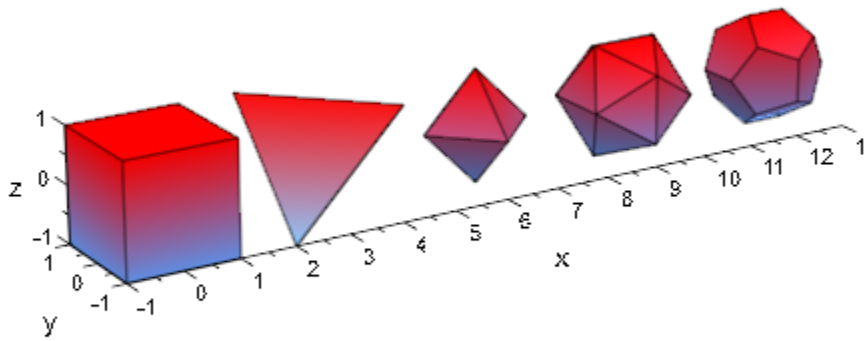
Attribute	Purpose	Default Value
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

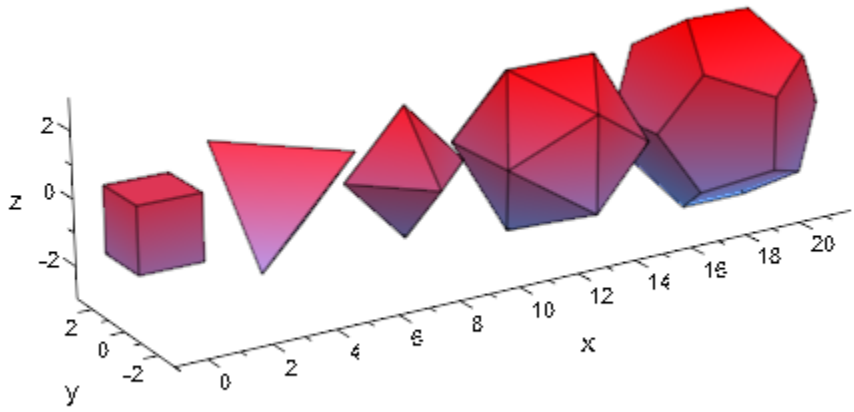
Using different `Center`s, the five regular polyhedra are placed side by side:

```
plot(plot::Hexahedron  (Center = [0, 0, 0]),
     plot::Tetrahedron (Center = [3, 0, 0]),
     plot::Octahedron  (Center = [6, 0, 0]),
     plot::Icosahedron (Center = [9, 0, 0]),
     plot::Dodecahedron(Center = [12, 0, 0]),
     Axes = Frame);
```



With the attribut `Radius`, the size of the polyhydra can be changed:

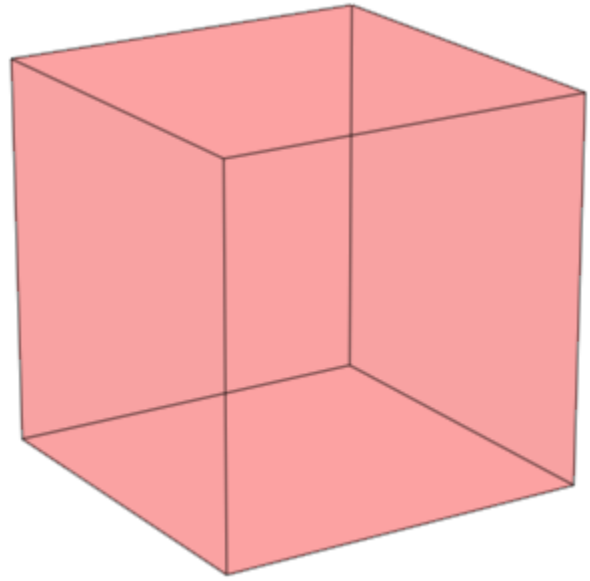
```
plot(plot::Hexahedron  (Radius = 1.0, Center = [0, 0, 0]),
     plot::Tetrahedron (Radius = 1.5, Center = [4, 0, 0]),
     plot::Octahedron  (Radius = 2.0, Center = [8, 0, 0]),
     plot::Icosahedron (Radius = 2.5, Center = [13, 0, 0]),
     plot::Dodecahedron(Radius = 3.0, Center = [19, 0, 0]),
     Axes = Frame);
```



Example 2

A tetrahedron and an octahedron are embedded in a hexahedron:

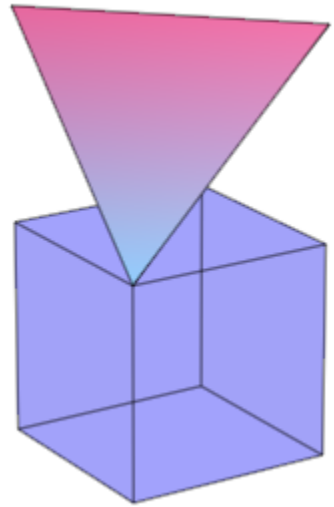
```
plot(plot::Hexahedron (FillColorFunction = RGB::Red.[0.2],
                       VisibleFromTo = 0..8),
     plot::Tetrahedron(FillColorFunction = RGB::Green.[0.2],
                       VisibleFromTo = 1..5),
     plot::Octahedron (FillColorFunction = RGB::Blue.[0.2],
                       VisibleFromTo = 3..7),
     Axes = None)
```



Example 3

Transformation objects can be applied to polyhedra as demonstrated below:

```
H := plot::Hexahedron(Color = RGB::Blue.[0.2],
                      FillColorType = Flat):
T := plot::Tetrahedron(Color = RGB::Red):
plot(plot::Rotate3d(a, [0, 0, 0], [0, 0, 1], a = 0..2*PI,
                    H,
                    plot::Translate3d([0, 0, a], T, a = 0..2)
                   ), Axes = None)
```

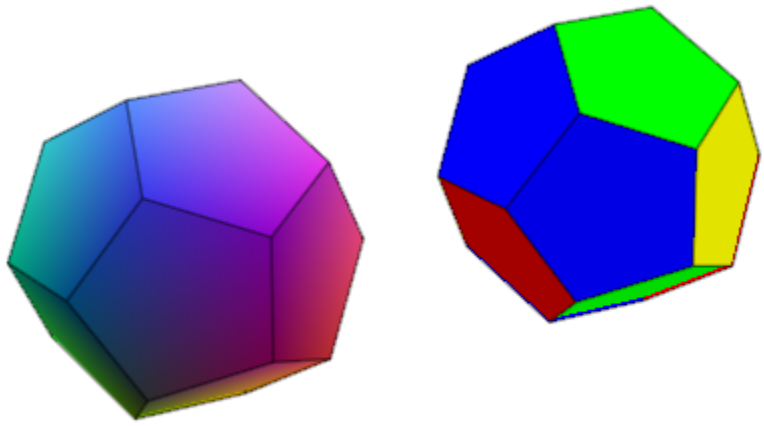


```
delete T, H:
```

Example 4

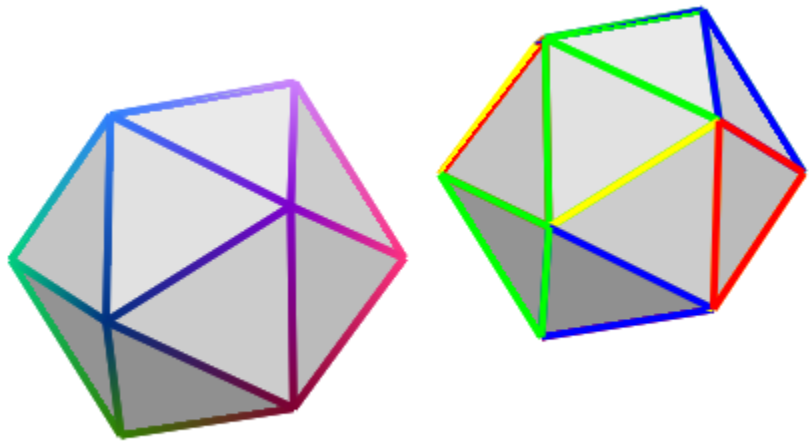
A `FillColorFunction` can be specified. This will be called with the index of the current facet as its first parameter, followed by the *x*-, *y*- and *z*-coordinate of the current point:

```
mycolorlist := [RGB::Red, RGB::Blue, RGB::Green, RGB::Yellow]:
plot(plot::Dodecahedron(Center = [0, 0, 0],
                        FillColorFunction =
                          proc(n, x, y, z) begin
                            [(1 + x)/2, (1 + y)/2, (1 + z)/2]
                          end_proc),
     plot::Dodecahedron(Center = [3, 0, 0],
                        FillColorFunction =
                          proc(n, x, y, z) begin
                             mycolorlist[(n mod 4)+1]
                          end_proc),
     Axes = None):
```



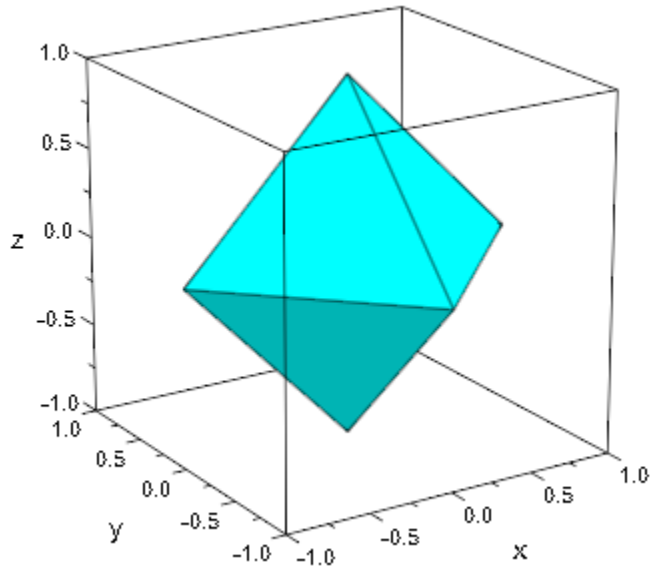
The same holds true for a `LineColorFunction`:

```
plot(plot::Icosahedron(Center = [0, 0, 0],
                       LineColorFunction =
                          proc(n, x, y, z) begin
                            [(1 + x)/2, (1 + y)/2, (1 + z)/2]
                          end_proc),
     plot::Icosahedron(Center = [3, 0, 0],
                       LineColorFunction =
                           proc(n, x, y, z) begin
                              mycolorlist[(n mod 4)+1]
                           end_proc),
     Axes = None, LineWidth = 1.0*unit::mm,
     FillColor = RGB::Grey80, FillColorType = Flat):
```



If the polyhedron is animated, the color functions are called with an additional argument: the current value of the animation parameter:

```
plot(plot::Octahedron(FillColorFunction =
                        proc(n, x, y, z, a)
                        begin
                          [sin(n*a)^2, cos(n*a)^2, 1]:
                        end_proc,
                      a = 0..2*PI))
```




```
delete mycolorlist:
```

Algorithms

A polyhedron is called regular if all its facets consist of the same regular polygon and each vertex has the same number of coincidence polygons.

Since Plato we know that only five regular polyhedrons exist:

- the tetrahedron with 4 (greek *tetra*) triangles,
- the hexahedron with 6 (greek *hexa*) squares,
- the octahedron with 8 (greek *okta*) triangles,
- the dodecahedron with 12 (greek *dodeka*) pentagons and
- the icosahedron with 20 (greek *eikosi*) triangles.

The following table lists some important geometrical data of the polyhedra with the edge length a. Where R is the radius of the outer spherem r the radius of the inner sphere, A the surface area and V the volume:

Ratio	Tetrahedron	Hexahedron	Octahedron	Dodecahedron	Icosahedron
$\frac{R}{a}$	$\frac{\sqrt{6}}{4}$	$\frac{\sqrt{3}}{2}$	$\frac{\sqrt{2}}{2}$	$\frac{\sqrt{3}\,(1+\sqrt{5})}{4}$	$\frac{\sqrt{10+2\sqrt{5}}}{4}$
$\frac{r}{a}$	$\frac{\sqrt{6}}{12}$	$\frac{1}{2}$	$\frac{\sqrt{6}}{6}$	$\frac{\sqrt{250+110\sqrt{5}}}{20}$	$\frac{\sqrt{3}\,(3+\sqrt{5})}{12}$
$\frac{A}{a^2}$	$\sqrt{3}$	6	$2\sqrt{3}$	$3\sqrt{25+10\sqrt{5}}$	$5\sqrt{3}$
$\frac{V}{a^3}$	$\frac{\sqrt{2}}{12}$	1	$\frac{\sqrt{2}}{3}$	$\frac{(15+7\sqrt{5})}{4}$	$\frac{5\,(3+\sqrt{5})}{12}$

See Also

MuPAD Functions
`plot`

MuPAD Graphical Primitives
`plot::Box` | `plot::Cone` | `plot::Cylinder` | `plot::Dodecahedron` | `plot::Hexahedron` | `plot::Icosahedron` | `plot::Octahedron` | `plot::Parallelogram3d` | `plot::Sphere` | `plot::Transform3d`

plot::Hexahedron

Regular Hexahedra

Syntax

```
plot::Hexahedron(<a = a_min .. a_max>, options)
```

Description

`plot::Hexahedron()` creates regular polyhedra.

Per default, all polyhedra are centered at the origin. The attribute `Center` allows to choose a different center. This is helpful to align the polyhedra relative to other objects in the graphical scene. Cf. “Example 1” on page 24-981.

All polyhedra fit into a box extending from -1 to 1 in all coordinate directions. Their size can be changed by the attribute `Radius`. In case of a hexahedron (a box), this attribute represents the radius of the inscribed sphere. For the other polyhedra, it is the radius of the circumscribed sphere.

The default value of `Radius` is 1 for all polyhedra.

Further to the attributes `Center` and `Radius`, you can modify the polyhedra by applying transformation objects of type `plot::Rotate3d`, `plot::Scale3d`, `plot::Translate3d` , and `plot::Transform3d`. Cf. “Example 3” on page 24-983.

User-defined color functions (`LineColorFunction`, `FillColorFunction`) are called with the index of the current facet as its first parameter, followed by the *x*, *y*, and *z* coordinate of the current point, followed by the current value of the animation parameter (if animated). Cf. “Example 4” on page 24-984.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`

Attribute	Purpose	Default Value
`Center`	center of objects, rotation center	[0, 0, 0]
`CenterX`	center of objects, rotation center, x-component	0
`CenterY`	center of objects, rotation center, y-component	0
`CenterZ`	center of objects, rotation center, z-component	0
`Color`	the main color	`RGB::Red`
`Filled`	filled or transparent areas and surfaces	`TRUE`
`FillColor`	color of areas and surfaces	`RGB::Red`
`FillColor2`	second color of areas and surfaces for color blends	`RGB::CornflowerBlue`
`FillColorType`	surface filling types	`Dichromatic`
`FillColorFunction`	functional area/surface coloring	
`FillColorDirection`	the direction of color transitions on surfaces	[0, 0, 1]
`FillColorDirectionX`	x-component of the direction of color transitions on surfaces	0
`FillColorDirectionY`	y-component of the direction of color transitions on surfaces	0
`FillColorDirectionZ`	z-component of the direction of color transitions on surfaces	1
`Frames`	the number of frames in an animation	50
`Legend`	makes a legend entry	

Attribute	Purpose	Default Value
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Black.[0.25]
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	0
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5

Attribute	Purpose	Default Value
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE
Radius	radius of circles, spheres etc.	1
Shading	smooth color blend of surfaces	Smooth
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE

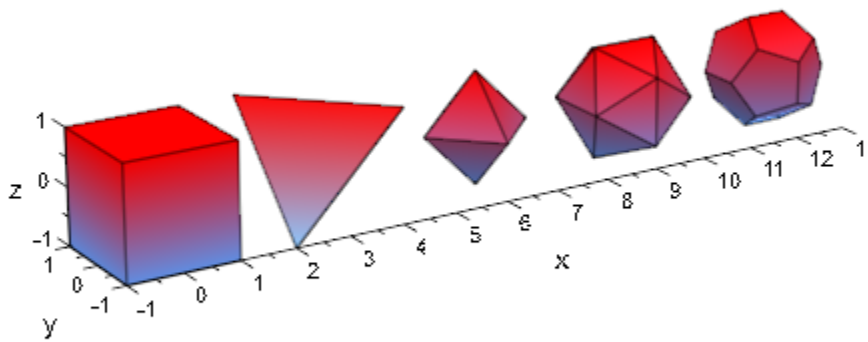
Attribute	Purpose	Default Value
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

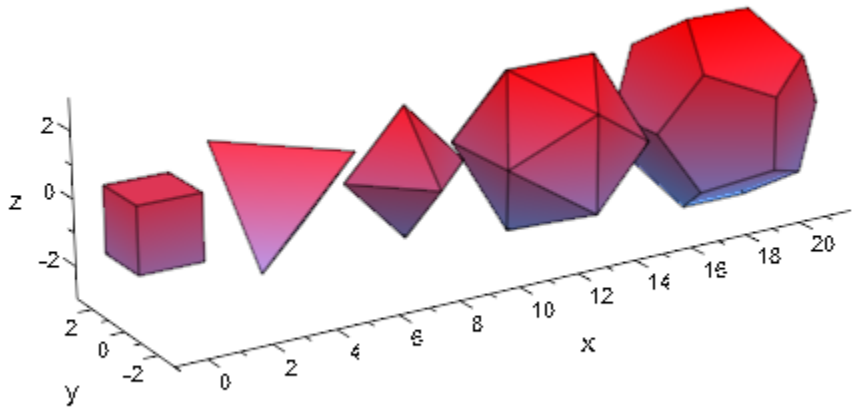
Using different `Center`s, the five regular polyhedra are placed side by side:

```
plot(plot::Hexahedron  (Center = [0, 0, 0]),
     plot::Tetrahedron (Center = [3, 0, 0]),
     plot::Octahedron  (Center = [6, 0, 0]),
     plot::Icosahedron (Center = [9, 0, 0]),
     plot::Dodecahedron(Center = [12, 0, 0]),
     Axes = Frame);
```



With the attribut `Radius`, the size of the polyhydra can be changed:

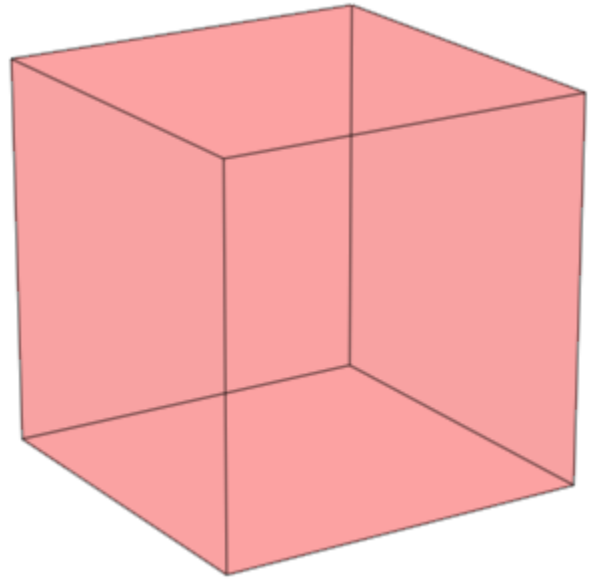
```
plot(plot::Hexahedron  (Radius = 1.0, Center = [0, 0, 0]),
     plot::Tetrahedron (Radius = 1.5, Center = [4, 0, 0]),
     plot::Octahedron  (Radius = 2.0, Center = [8, 0, 0]),
     plot::Icosahedron (Radius = 2.5, Center = [13, 0, 0]),
     plot::Dodecahedron(Radius = 3.0, Center = [19, 0, 0]),
     Axes = Frame);
```



Example 2

A tetrahedron and an octahedron are embedded in a hexahedron:

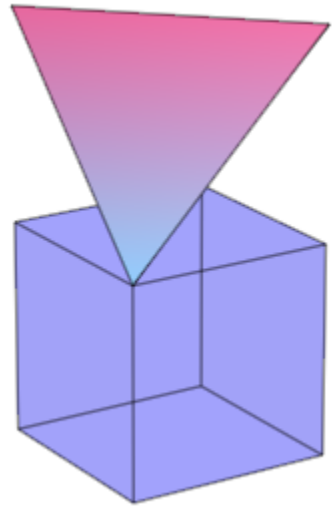
```
plot(plot::Hexahedron (FillColorFunction = RGB::Red.[0.2],
                       VisibleFromTo = 0..8),
     plot::Tetrahedron(FillColorFunction = RGB::Green.[0.2],
                       VisibleFromTo = 1..5),
     plot::Octahedron (FillColorFunction = RGB::Blue.[0.2],
                       VisibleFromTo = 3..7),
     Axes = None)
```



Example 3

Transformation objects can be applied to polyhedra as demonstrated below:

```
H := plot::Hexahedron(Color = RGB::Blue.[0.2],
                      FillColorType = Flat):
T := plot::Tetrahedron(Color = RGB::Red):
plot(plot::Rotate3d(a, [0, 0, 0], [0, 0, 1], a = 0..2*PI,
                    H,
                    plot::Translate3d([0, 0, a], T, a = 0..2)
                   ), Axes = None)
```

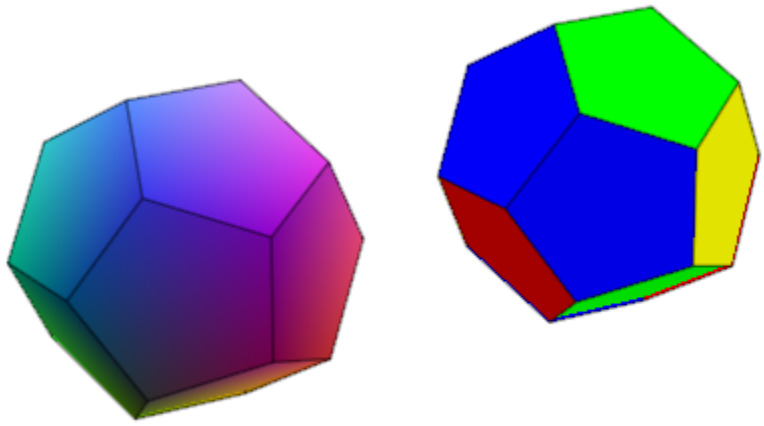


```
delete T, H:
```

Example 4

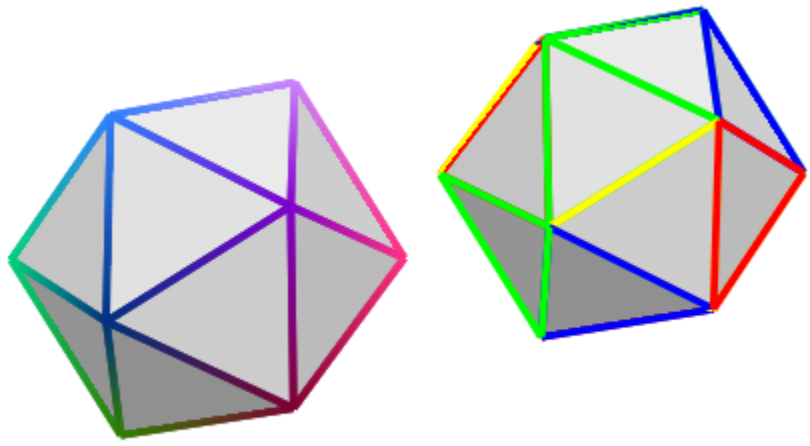
A `FillColorFunction` can be specified. This will be called with the index of the current facet as its first parameter, followed by the *x*-, *y*- and *z*-coordinate of the current point:

```
mycolorlist := [RGB::Red, RGB::Blue, RGB::Green, RGB::Yellow]:
plot(plot::Dodecahedron(Center = [0, 0, 0],
                        FillColorFunction =
                          proc(n, x, y, z) begin
                            [(1 + x)/2, (1 + y)/2, (1 + z)/2]
                          end_proc),
     plot::Dodecahedron(Center = [3, 0, 0],
                        FillColorFunction =
                          proc(n, x, y, z) begin
                             mycolorlist[(n mod 4)+1]
                          end_proc),
     Axes = None):
```



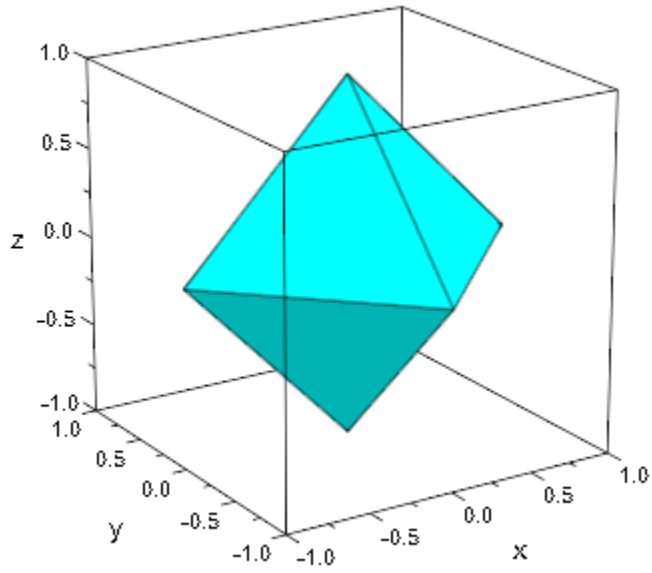
The same holds true for a `LineColorFunction`:

```
plot(plot::Icosahedron(Center = [0, 0, 0],
                       LineColorFunction =
                          proc(n, x, y, z) begin
                            [(1 + x)/2, (1 + y)/2, (1 + z)/2]
                          end_proc),
     plot::Icosahedron(Center = [3, 0, 0],
                       LineColorFunction =
                           proc(n, x, y, z) begin
                              mycolorlist[(n mod 4)+1]
                           end_proc),
     Axes = None, LineWidth = 1.0*unit::mm,
     FillColor = RGB::Grey80, FillColorType = Flat):
```



If the polyhedron is animated, the color functions are called with an additional argument: the current value of the animation parameter:

```
plot(plot::Octahedron(FillColorFunction =
                        proc(n, x, y, z, a)
                        begin
                          [sin(n*a)^2, cos(n*a)^2, 1]:
                        end_proc,
                      a = 0..2*PI))
```




```
delete mycolorlist:
```

Algorithms

A polyhedron is called regular if all its facets consist of the same regular polygon and each vertex has the same number of coincidence polygons.

Since Plato we know that only five regular polyhedrons exist:

- the tetrahedron with 4 (greek *tetra*) triangles,
- the hexahedron with 6 (greek *hexa*) squares,
- the octahedron with 8 (greek *okta*) triangles,
- the dodecahedron with 12 (greek *dodeka*) pentagons and
- the icosahedron with 20 (greek *eikosi*) triangles.

The following table lists some important geometrical data of the polyhedra with the edge length a. Where R is the radius of the outer spherem r the radius of the inner sphere, A the surface area and V the volume:

Ratio	Tetrahedron	Hexahedron	Octahedron	Dodecahedron	Icosahedron
$\frac{R}{a}$	$\frac{\sqrt{6}}{4}$	$\frac{\sqrt{3}}{2}$	$\frac{\sqrt{2}}{2}$	$\frac{\sqrt{3}\,(1+\sqrt{5})}{4}$	$\frac{\sqrt{10+2\,\sqrt{5}}}{4}$
$\frac{r}{a}$	$\frac{\sqrt{6}}{12}$	$\frac{1}{2}$	$\frac{\sqrt{6}}{6}$	$\frac{\sqrt{250+110\,\sqrt{5}}}{20}$	$\frac{\sqrt{3}\,(3+\sqrt{5})}{12}$
$\frac{A}{a^2}$	$\sqrt{3}$	6	$2\,\sqrt{3}$	$3\,\sqrt{25+10\,\sqrt{5}}$	$5\,\sqrt{3}$
$\frac{V}{a^3}$	$\frac{\sqrt{2}}{12}$	1	$\frac{\sqrt{2}}{3}$	$\frac{(15+7\,\sqrt{5})}{4}$	$\frac{5\,(3+\sqrt{5})}{12}$

See Also

MuPAD Functions
`plot`

MuPAD Graphical Primitives
`plot::Box` | `plot::Cone` | `plot::Cylinder` | `plot::Dodecahedron` | `plot::Icosahedron` | `plot::Octahedron` | `plot::Parallelogram3d` | `plot::Sphere` | `plot::Tetrahedron` | `plot::Transform3d`

plot::Octahedron

Regular Octahedra

Syntax

```
plot::Octahedron(<a = a_min .. a_max>, options)
```

Description

`plot::Octahedron()` creates regular polyhedra.

Per default, all polyhedra are centered at the origin. The attribute `Center` allows to choose a different center. This is helpful to align the polyhedra relative to other objects in the graphical scene. Cf. “Example 1” on page 24-993.

All polyhedra fit into a box extending from -1 to 1 in all coordinate directions. Their size can be changed by the attribute `Radius`. In case of a hexahedron (a box), this attribute represents the radius of the inscribed sphere. For the other polyhedra, it is the radius of the circumscribed sphere.

The default value of `Radius` is 1 for all polyhedra.

Further to the attributes `Center` and `Radius`, you can modify the polyhedra by applying transformation objects of type `plot::Rotate3d`, `plot::Scale3d`, `plot::Translate3d` , and `plot::Transform3d`. Cf. “Example 3” on page 24-995.

User-defined color functions (`LineColorFunction`, `FillColorFunction`) are called with the index of the current facet as its first parameter, followed by the *x*, *y*, and *z* coordinate of the current point, followed by the current value of the animation parameter (if animated). Cf. “Example 4” on page 24-996.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`

Attribute	Purpose	Default Value
`Center`	center of objects, rotation center	[0, 0, 0]
`CenterX`	center of objects, rotation center, x-component	0
`CenterY`	center of objects, rotation center, y-component	0
`CenterZ`	center of objects, rotation center, z-component	0
`Color`	the main color	`RGB::Red`
`Filled`	filled or transparent areas and surfaces	`TRUE`
`FillColor`	color of areas and surfaces	`RGB::Red`
`FillColor2`	second color of areas and surfaces for color blends	`RGB::CornflowerBlue`
`FillColorType`	surface filling types	`Dichromatic`
`FillColorFunction`	functional area/surface coloring	
`FillColorDirection`	the direction of color transitions on surfaces	[0, 0, 1]
`FillColorDirectionX`	x-component of the direction of color transitions on surfaces	0
`FillColorDirectionY`	y-component of the direction of color transitions on surfaces	0
`FillColorDirectionZ`	z-component of the direction of color transitions on surfaces	1
`Frames`	the number of frames in an animation	50
`Legend`	makes a legend entry	

Attribute	Purpose	Default Value
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Black.[0.25]
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	0
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5

Attribute	Purpose	Default Value
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE
Radius	radius of circles, spheres etc.	1
Shading	smooth color blend of surfaces	Smooth
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE

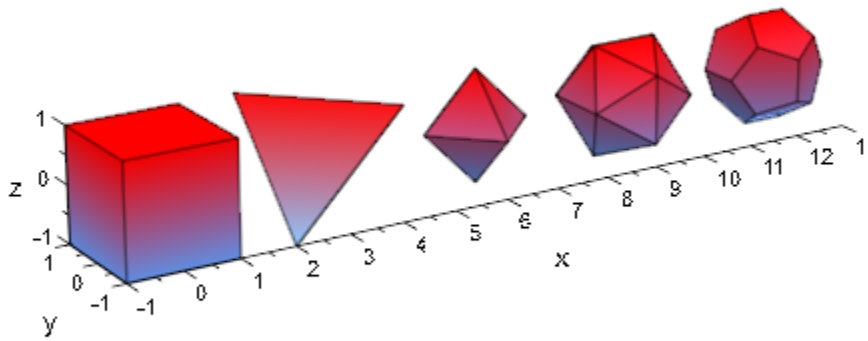
Attribute	Purpose	Default Value
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

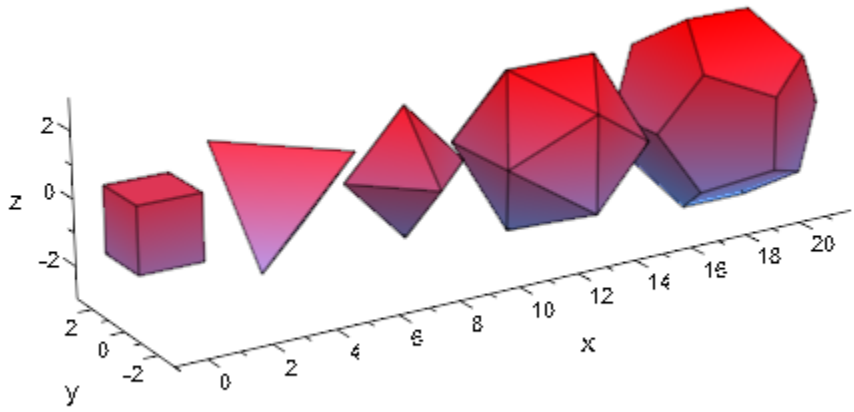
Using different `Center`s, the five regular polyhedra are placed side by side:

```
plot(plot::Hexahedron  (Center = [0, 0, 0]),
     plot::Tetrahedron (Center = [3, 0, 0]),
     plot::Octahedron  (Center = [6, 0, 0]),
     plot::Icosahedron (Center = [9, 0, 0]),
     plot::Dodecahedron(Center = [12, 0, 0]),
     Axes = Frame);
```



With the attribut `Radius`, the size of the polyhydra can be changed:

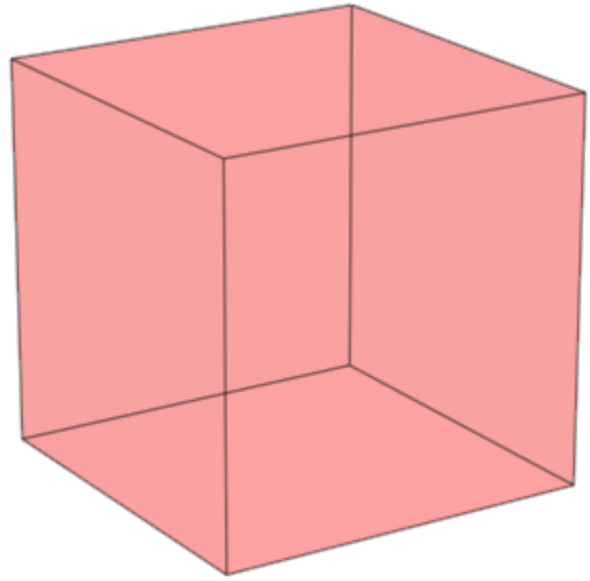
```
plot(plot::Hexahedron  (Radius = 1.0, Center = [0, 0, 0]),
     plot::Tetrahedron (Radius = 1.5, Center = [4, 0, 0]),
     plot::Octahedron  (Radius = 2.0, Center = [8, 0, 0]),
     plot::Icosahedron (Radius = 2.5, Center = [13, 0, 0]),
     plot::Dodecahedron(Radius = 3.0, Center = [19, 0, 0]),
     Axes = Frame);
```



Example 2

A tetrahedron and an octahedron are embedded in a hexahedron:

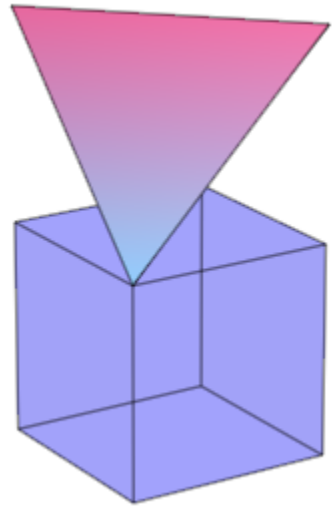
```
plot(plot::Hexahedron (FillColorFunction = RGB::Red.[0.2],
                       VisibleFromTo = 0..8),
     plot::Tetrahedron(FillColorFunction = RGB::Green.[0.2],
                       VisibleFromTo = 1..5),
     plot::Octahedron (FillColorFunction = RGB::Blue.[0.2],
                       VisibleFromTo = 3..7),
     Axes = None)
```



Example 3

Transformation objects can be applied to polyhedra as demonstrated below:

```
H := plot::Hexahedron(Color = RGB::Blue.[0.2],
                      FillColorType = Flat):
T := plot::Tetrahedron(Color = RGB::Red):
plot(plot::Rotate3d(a, [0, 0, 0], [0, 0, 1], a = 0..2*PI,
                    H,
                    plot::Translate3d([0, 0, a], T, a = 0..2)
                   ), Axes = None)
```

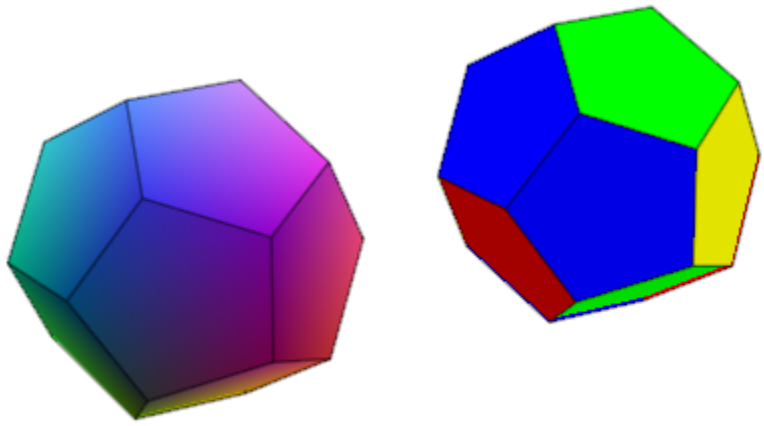


```
delete T, H:
```

Example 4

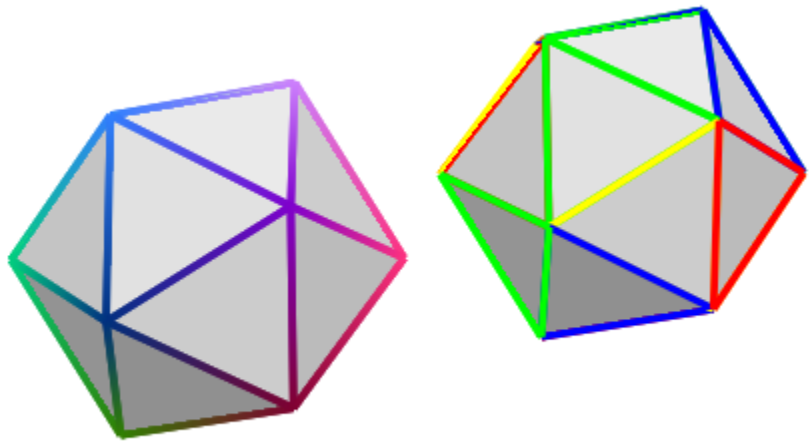
A `FillColorFunction` can be specified. This will be called with the index of the current facet as its first parameter, followed by the *x*-, *y*- and *z*-coordinate of the current point:

```
mycolorlist := [RGB::Red, RGB::Blue, RGB::Green, RGB::Yellow]:
plot(plot::Dodecahedron(Center = [0, 0, 0],
                        FillColorFunction =
                          proc(n, x, y, z) begin
                            [(1 + x)/2, (1 + y)/2, (1 + z)/2]
                          end_proc),
     plot::Dodecahedron(Center = [3, 0, 0],
                        FillColorFunction =
                          proc(n, x, y, z) begin
                             mycolorlist[(n mod 4)+1]
                          end_proc),
     Axes = None):
```



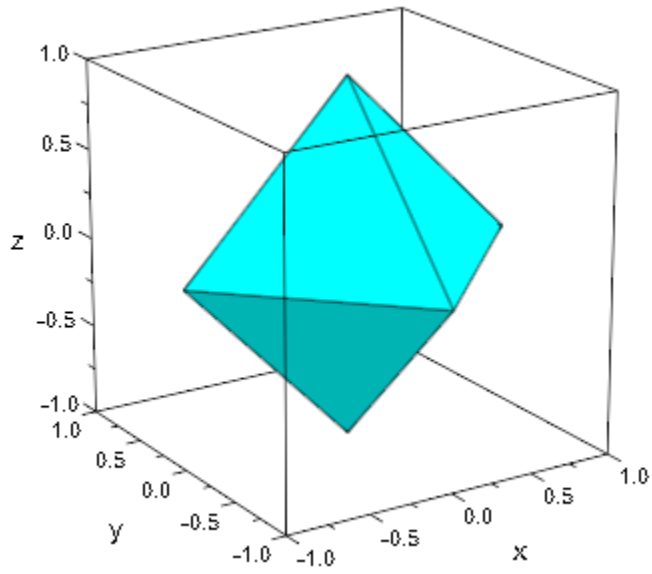
The same holds true for a `LineColorFunction`:

```
plot(plot::Icosahedron(Center = [0, 0, 0],
                       LineColorFunction =
                          proc(n, x, y, z) begin
                            [(1 + x)/2, (1 + y)/2, (1 + z)/2]
                          end_proc),
     plot::Icosahedron(Center = [3, 0, 0],
                       LineColorFunction =
                           proc(n, x, y, z) begin
                              mycolorlist[(n mod 4)+1]
                           end_proc),
     Axes = None, LineWidth = 1.0*unit::mm,
     FillColor = RGB::Grey80, FillColorType = Flat):
```



If the polyhedron is animated, the color functions are called with an additional argument: the current value of the animation parameter:

```
plot(plot::Octahedron(FillColorFunction =
                        proc(n, x, y, z, a)
                        begin
                          [sin(n*a)^2, cos(n*a)^2, 1]:
                        end_proc,
                      a = 0..2*PI))
```




```
delete mycolorlist:
```

Algorithms

A polyhedron is called regular if all its facets consist of the same regular polygon and each vertex has the same number of coincidence polygons.

Since Plato we know that only five regular polyhedrons exist:

- the tetrahedron with 4 (greek *tetra*) triangles,
- the hexahedron with 6 (greek *hexa*) squares,
- the octahedron with 8 (greek *okta*) triangles,
- the dodecahedron with 12 (greek *dodeka*) pentagons and
- the icosahedron with 20 (greek *eikosi*) triangles.

The following table lists some important geometrical data of the polyhedra with the edge length a. Where R is the radius of the outer spherem r the radius of the inner sphere, A the surface area and V the volume:

Ratio	Tetrahedron	Hexahedron	Octahedron	Dodecahedron	Icosahedron
$\frac{R}{a}$	$\frac{\sqrt{6}}{4}$	$\frac{\sqrt{3}}{2}$	$\frac{\sqrt{2}}{2}$	$\frac{\sqrt{3}\,(1+\sqrt{5})}{4}$	$\frac{\sqrt{10+2\,\sqrt{5}}}{4}$
$\frac{r}{a}$	$\frac{\sqrt{6}}{12}$	$\frac{1}{2}$	$\frac{\sqrt{6}}{6}$	$\frac{\sqrt{250+110\,\sqrt{5}}}{20}$	$\frac{\sqrt{3}\,(3+\sqrt{5})}{12}$
$\frac{A}{a^2}$	$\sqrt{3}$	6	$2\,\sqrt{3}$	$3\,\sqrt{25+10\,\sqrt{5}}$	$5\,\sqrt{3}$
$\frac{V}{a^3}$	$\frac{\sqrt{2}}{12}$	1	$\frac{\sqrt{2}}{3}$	$\frac{(15+7\,\sqrt{5})}{4}$	$\frac{5\,(3+\sqrt{5})}{12}$

See Also

MuPAD Functions
`plot`

MuPAD Graphical Primitives
`plot::Box` | `plot::Cone` | `plot::Cylinder` | `plot::Dodecahedron` | `plot::Hexahedron` | `plot::Icosahedron` | `plot::Parallelogram3d` | `plot::Sphere` | `plot::Tetrahedron` | `plot::Transform3d`

plot::Dodecahedron

Regular Dodecahedra

Syntax

```
plot::Dodecahedron(<a = a_min .. a_max>, options)
```

Description

`plot::Dodecahedron()` creates regular polyhedra.

Per default, all polyhedra are centered at the origin. The attribute `Center` allows to choose a different center. This is helpful to align the polyhedra relative to other objects in the graphical scene. Cf. “Example 1” on page 24-1005.

All polyhedra fit into a box extending from -1 to 1 in all coordinate directions. Their size can be changed by the attribute `Radius`. In case of a hexahedron (a box), this attribute represents the radius of the inscribed sphere. For the other polyhedra, it is the radius of the circumscribed sphere.

The default value of `Radius` is 1 for all polyhedra.

Further to the attributes `Center` and `Radius`, you can modify the polyhedra by applying transformation objects of type `plot::Rotate3d`, `plot::Scale3d`, `plot::Translate3d` , and `plot::Transform3d`. Cf. “Example 3” on page 24-1007.

User-defined color functions (`LineColorFunction`, `FillColorFunction`) are called with the index of the current facet as its first parameter, followed by the *x*, *y*, and *z* coordinate of the current point, followed by the current value of the animation parameter (if animated). Cf. “Example 4” on page 24-1008.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`

Attribute	Purpose	Default Value
`Center`	center of objects, rotation center	[0, 0, 0]
`CenterX`	center of objects, rotation center, x-component	0
`CenterY`	center of objects, rotation center, y-component	0
`CenterZ`	center of objects, rotation center, z-component	0
`Color`	the main color	`RGB::Red`
`Filled`	filled or transparent areas and surfaces	`TRUE`
`FillColor`	color of areas and surfaces	`RGB::Red`
`FillColor2`	second color of areas and surfaces for color blends	`RGB::CornflowerBlue`
`FillColorType`	surface filling types	`Dichromatic`
`FillColorFunction`	functional area/surface coloring	
`FillColorDirection`	the direction of color transitions on surfaces	[0, 0, 1]
`FillColorDirectionX`	x-component of the direction of color transitions on surfaces	0
`FillColorDirectionY`	y-component of the direction of color transitions on surfaces	0
`FillColorDirectionZ`	z-component of the direction of color transitions on surfaces	1
`Frames`	the number of frames in an animation	50
`Legend`	makes a legend entry	

Attribute	Purpose	Default Value
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Black.[0.25]
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	0
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5

Attribute	Purpose	Default Value
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE
Radius	radius of circles, spheres etc.	1
Shading	smooth color blend of surfaces	Smooth
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE

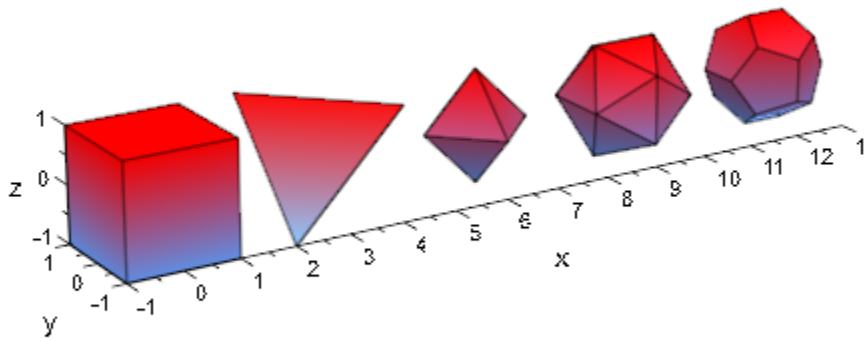
Attribute	Purpose	Default Value
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

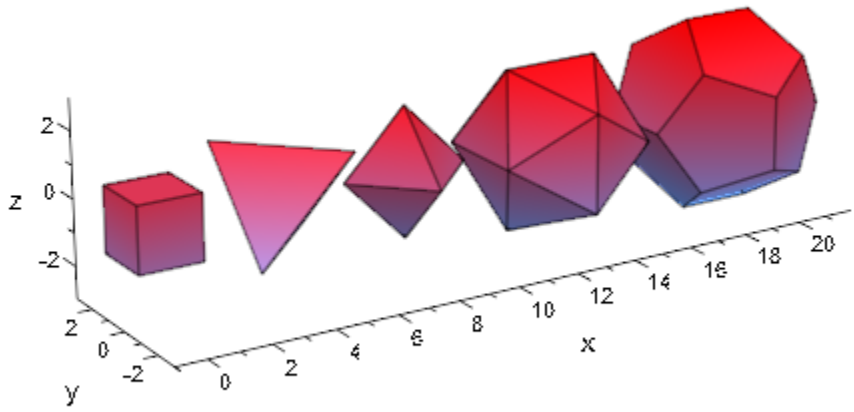
Using different `Center`s, the five regular polyhedra are placed side by side:

```
plot(plot::Hexahedron  (Center = [0, 0, 0]),
     plot::Tetrahedron (Center = [3, 0, 0]),
     plot::Octahedron  (Center = [6, 0, 0]),
     plot::Icosahedron (Center = [9, 0, 0]),
     plot::Dodecahedron(Center = [12, 0, 0]),
     Axes = Frame);
```



With the attribut `Radius`, the size of the polyhydra can be changed:

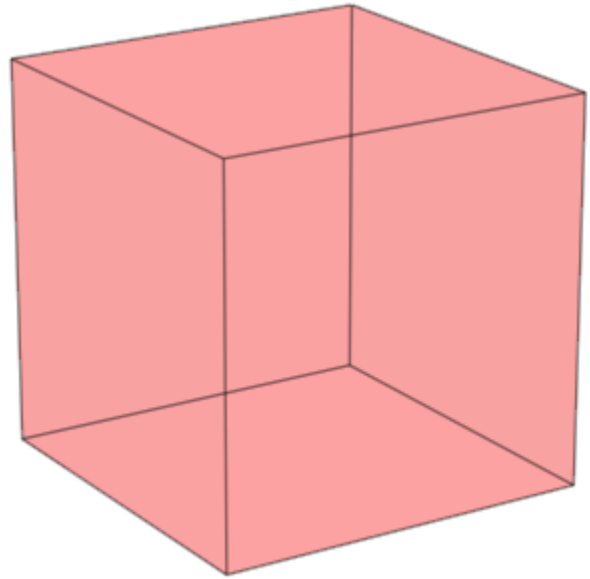
```
plot(plot::Hexahedron  (Radius = 1.0, Center = [0, 0, 0]),
     plot::Tetrahedron (Radius = 1.5, Center = [4, 0, 0]),
     plot::Octahedron  (Radius = 2.0, Center = [8, 0, 0]),
     plot::Icosahedron (Radius = 2.5, Center = [13, 0, 0]),
     plot::Dodecahedron(Radius = 3.0, Center = [19, 0, 0]),
     Axes = Frame);
```



Example 2

A tetrahedron and an octahedron are embedded in a hexahedron:

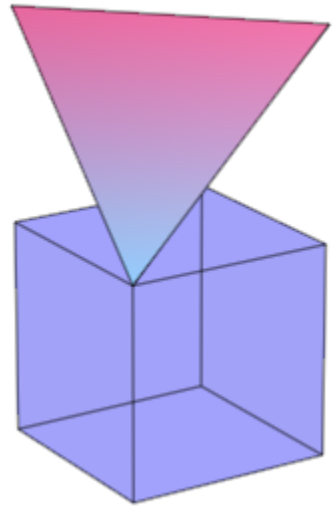
```
plot(plot::Hexahedron (FillColorFunction = RGB::Red.[0.2],
                       VisibleFromTo = 0..8),
     plot::Tetrahedron(FillColorFunction = RGB::Green.[0.2],
                       VisibleFromTo = 1..5),
     plot::Octahedron (FillColorFunction = RGB::Blue.[0.2],
                       VisibleFromTo = 3..7),
     Axes = None)
```



Example 3

Transformation objects can be applied to polyhedra as demonstrated below:

```
H := plot::Hexahedron(Color = RGB::Blue.[0.2],
                      FillColorType = Flat):
T := plot::Tetrahedron(Color = RGB::Red):
plot(plot::Rotate3d(a, [0, 0, 0], [0, 0, 1], a = 0..2*PI,
                    H,
                    plot::Translate3d([0, 0, a], T, a = 0..2)
                   ), Axes = None)
```

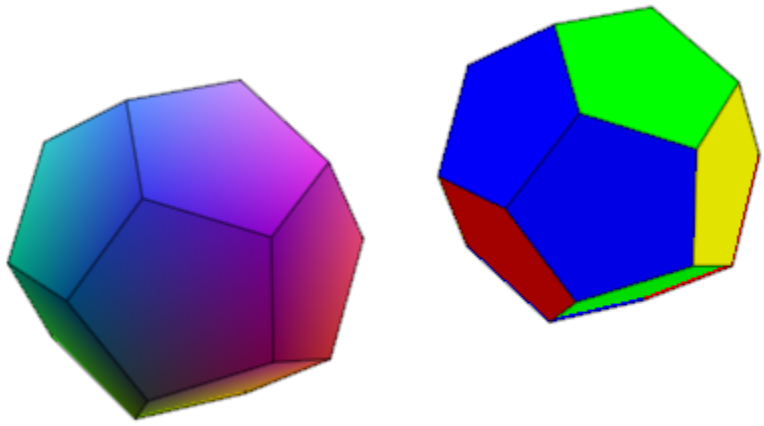


```
delete T, H:
```

Example 4

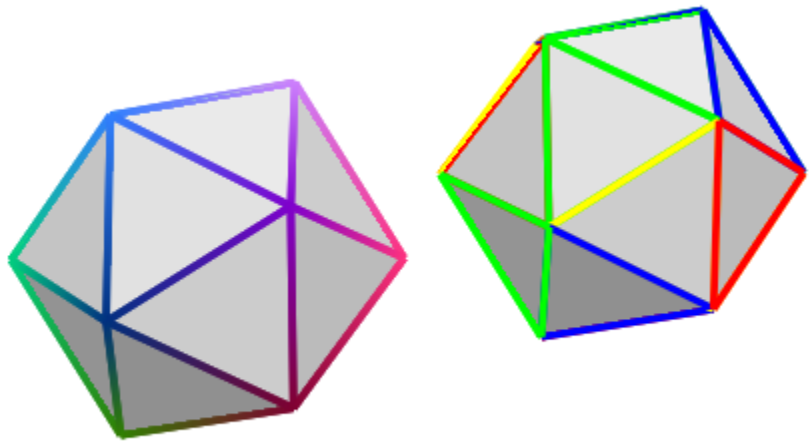
A `FillColorFunction` can be specified. This will be called with the index of the current facet as its first parameter, followed by the *x*-, *y*- and *z*-coordinate of the current point:

```
mycolorlist := [RGB::Red, RGB::Blue, RGB::Green, RGB::Yellow]:
plot(plot::Dodecahedron(Center = [0, 0, 0],
                        FillColorFunction =
                          proc(n, x, y, z) begin
                            [(1 + x)/2, (1 + y)/2, (1 + z)/2]
                          end_proc),
     plot::Dodecahedron(Center = [3, 0, 0],
                        FillColorFunction =
                          proc(n, x, y, z) begin
                             mycolorlist[(n mod 4)+1]
                          end_proc),
     Axes = None):
```



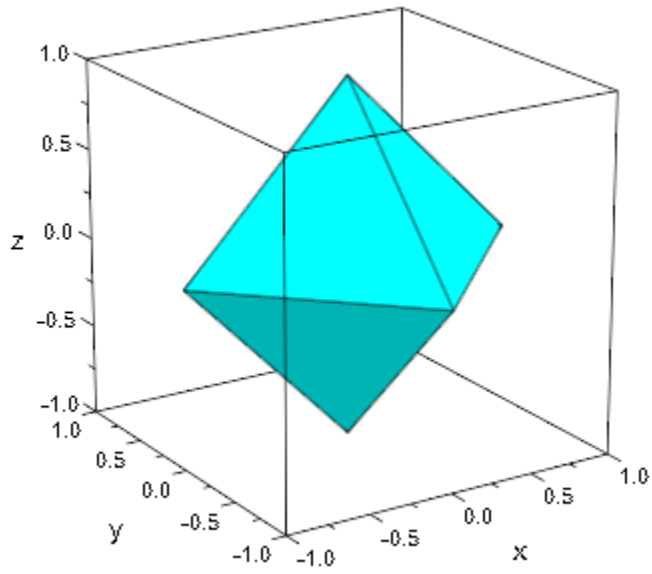
The same holds true for a `LineColorFunction`:

```
plot(plot::Icosahedron(Center = [0, 0, 0],
                       LineColorFunction =
                          proc(n, x, y, z) begin
                            [(1 + x)/2, (1 + y)/2, (1 + z)/2]
                          end_proc),
     plot::Icosahedron(Center = [3, 0, 0],
                       LineColorFunction =
                           proc(n, x, y, z) begin
                              mycolorlist[(n mod 4)+1]
                           end_proc),
     Axes = None, LineWidth = 1.0*unit::mm,
     FillColor = RGB::Grey80, FillColorType = Flat):
```



If the polyhedron is animated, the color functions are called with an additional argument: the current value of the animation parameter:

```
plot(plot::Octahedron(FillColorFunction =
                        proc(n, x, y, z, a)
                        begin
                          [sin(n*a)^2, cos(n*a)^2, 1]:
                        end_proc,
                      a = 0..2*PI))
```




```
delete mycolorlist:
```

Algorithms

A polyhedron is called regular if all its facets consist of the same regular polygon and each vertex has the same number of coincidence polygons.

Since Plato we know that only five regular polyhedrons exist:

- the tetrahedron with 4 (greek *tetra*) triangles,
- the hexahedron with 6 (greek *hexa*) squares,
- the octahedron with 8 (greek *okta*) triangles,
- the dodecahedron with 12 (greek *dodeka*) pentagons and
- the icosahedron with 20 (greek *eikosi*) triangles.

The following table lists some important geometrical data of the polyhedra with the edge length a. Where R is the radius of the outer sphere, r the radius of the inner sphere, A the surface area, and V the volume:

Ratio	Tetrahedron	Hexahedron	Octahedron	Dodecahedron	Icosahedron
$\frac{R}{a}$	$\frac{\sqrt{6}}{4}$	$\frac{\sqrt{3}}{2}$	$\frac{\sqrt{2}}{2}$	$\frac{\sqrt{3}\,(1+\sqrt{5})}{4}$	$\frac{\sqrt{10+2\,\sqrt{5}}}{4}$
$\frac{r}{a}$	$\frac{\sqrt{6}}{12}$	$\frac{1}{2}$	$\frac{\sqrt{6}}{6}$	$\frac{\sqrt{250+110\,\sqrt{5}}}{20}$	$\frac{\sqrt{3}\,(3+\sqrt{5})}{12}$
$\frac{A}{a^2}$	$\sqrt{3}$	6	$2\,\sqrt{3}$	$3\,\sqrt{25+10\,\sqrt{5}}$	$5\,\sqrt{3}$
$\frac{V}{a^3}$	$\frac{\sqrt{2}}{12}$	1	$\frac{\sqrt{2}}{3}$	$\frac{(15+7\,\sqrt{5})}{4}$	$\frac{5\,(3+\sqrt{5})}{12}$

See Also

MuPAD Functions
`plot`

MuPAD Graphical Primitives
`plot::Box` | `plot::Cone` | `plot::Cylinder` | `plot::Hexahedron` | `plot::Icosahedron` | `plot::Octahedron` | `plot::Parallelogram3d` | `plot::Sphere` | `plot::Tetrahedron` | `plot::Transform3d`

plot::Icosahedron

Regular Icosahedra

Syntax

```
plot::Icosahedron(<a = a_min .. a_max>, options)
```

Description

`plot::Icosahedron()` creates regular polyhedra.

Per default, all polyhedra are centered at the origin. The attribute `Center` allows to choose a different center. This is helpful to align the polyhedra relative to other objects in the graphical scene. Cf. “Example 1” on page 24-1017.

All polyhedra fit into a box extending from -1 to 1 in all coordinate directions. Their size can be changed by the attribute `Radius`. In case of a hexahedron (a box), this attribute represents the radius of the inscribed sphere. For the other polyhedra, it is the radius of the circumscribed sphere.

The default value of `Radius` is 1 for all polyhedra.

Further to the attributes `Center` and `Radius`, you can modify the polyhedra by applying transformation objects of type `plot::Rotate3d`, `plot::Scale3d`, `plot::Translate3d` , and `plot::Transform3d`. Cf. “Example 3” on page 24-1019.

User-defined color functions (`LineColorFunction`, `FillColorFunction`) are called with the index of the current facet as its first parameter, followed by the *x*, *y*, and *z* coordinate of the current point, followed by the current value of the animation parameter (if animated). Cf. “Example 4” on page 24-1020.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`

Attribute	Purpose	Default Value
`Center`	center of objects, rotation center	[0, 0, 0]
`CenterX`	center of objects, rotation center, x-component	0
`CenterY`	center of objects, rotation center, y-component	0
`CenterZ`	center of objects, rotation center, z-component	0
`Color`	the main color	`RGB::Red`
`Filled`	filled or transparent areas and surfaces	`TRUE`
`FillColor`	color of areas and surfaces	`RGB::Red`
`FillColor2`	second color of areas and surfaces for color blends	`RGB::CornflowerBlue`
`FillColorType`	surface filling types	`Dichromatic`
`FillColorFunction`	functional area/surface coloring	
`FillColorDirection`	the direction of color transitions on surfaces	[0, 0, 1]
`FillColorDirectionX`	x-component of the direction of color transitions on surfaces	0
`FillColorDirectionY`	y-component of the direction of color transitions on surfaces	0
`FillColorDirectionZ`	z-component of the direction of color transitions on surfaces	1
`Frames`	the number of frames in an animation	50
`Legend`	makes a legend entry	

Attribute	Purpose	Default Value
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Black.[0.25]
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	0
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5

Attribute	Purpose	Default Value
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE
Radius	radius of circles, spheres etc.	1
Shading	smooth color blend of surfaces	Smooth
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE

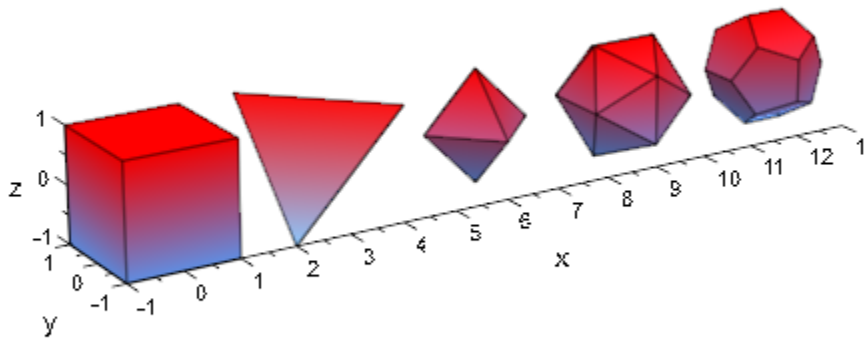
Attribute	Purpose	Default Value
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

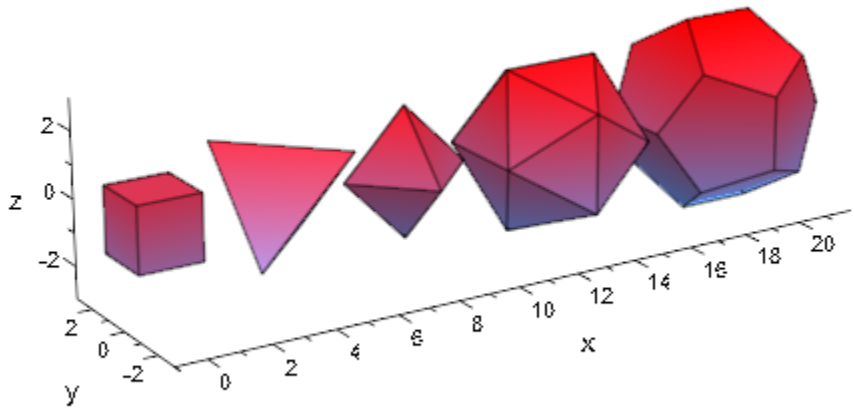
Using different `Center`s, the five regular polyhedra are placed side by side:

```
plot(plot::Hexahedron  (Center = [0, 0, 0]),
     plot::Tetrahedron (Center = [3, 0, 0]),
     plot::Octahedron  (Center = [6, 0, 0]),
     plot::Icosahedron (Center = [9, 0, 0]),
     plot::Dodecahedron(Center = [12, 0, 0]),
     Axes = Frame);
```



With the attribut `Radius`, the size of the polyhydra can be changed:

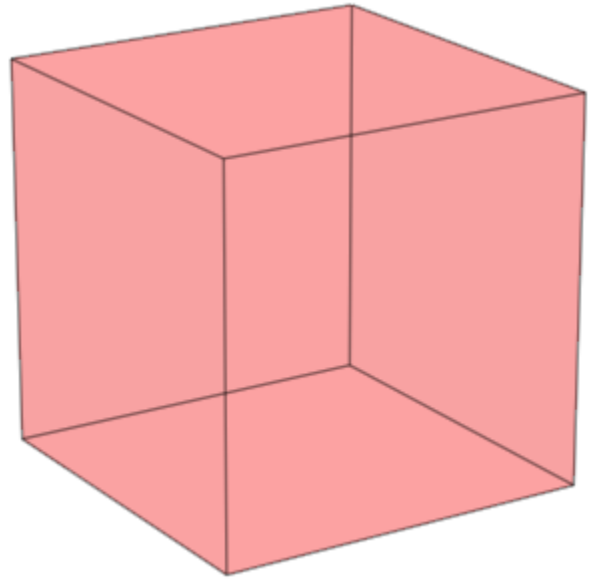
```
plot(plot::Hexahedron  (Radius = 1.0, Center = [0, 0, 0]),
     plot::Tetrahedron (Radius = 1.5, Center = [4, 0, 0]),
     plot::Octahedron  (Radius = 2.0, Center = [8, 0, 0]),
     plot::Icosahedron (Radius = 2.5, Center = [13, 0, 0]),
     plot::Dodecahedron(Radius = 3.0, Center = [19, 0, 0]),
     Axes = Frame);
```



Example 2

A tetrahedron and an octahedron are embedded in a hexahedron:

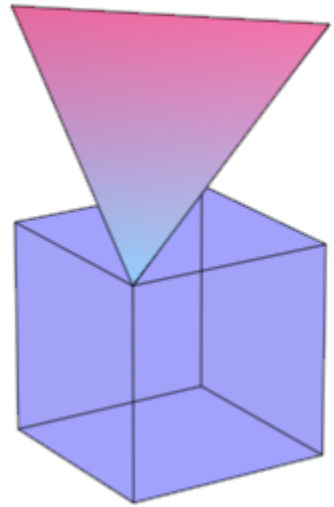
```
plot(plot::Hexahedron (FillColorFunction = RGB::Red.[0.2],
                       VisibleFromTo = 0..8),
     plot::Tetrahedron(FillColorFunction = RGB::Green.[0.2],
                       VisibleFromTo = 1..5),
     plot::Octahedron (FillColorFunction = RGB::Blue.[0.2],
                       VisibleFromTo = 3..7),
     Axes = None)
```



Example 3

Transformation objects can be applied to polyhedra as demonstrated below:

```
H := plot::Hexahedron(Color = RGB::Blue.[0.2],
                      FillColorType = Flat):
T := plot::Tetrahedron(Color = RGB::Red):
plot(plot::Rotate3d(a, [0, 0, 0], [0, 0, 1], a = 0..2*PI,
                    H,
                    plot::Translate3d([0, 0, a], T, a = 0..2)
                   ), Axes = None)
```

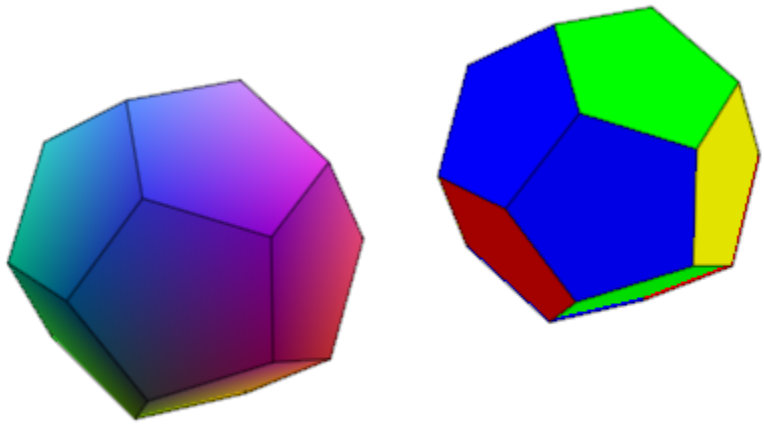


```
delete T, H:
```

Example 4

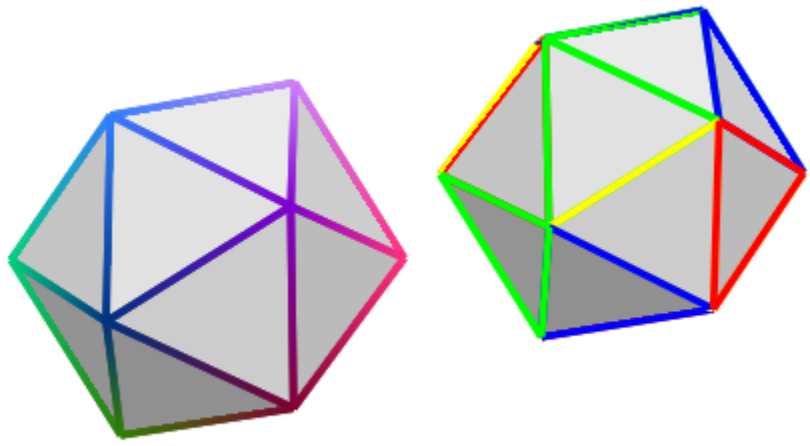
A `FillColorFunction` can be specified. This will be called with the index of the current facet as its first parameter, followed by the *x*-, *y*- and *z*-coordinate of the current point:

```
mycolorlist := [RGB::Red, RGB::Blue, RGB::Green, RGB::Yellow]:
plot(plot::Dodecahedron(Center = [0, 0, 0],
                        FillColorFunction =
                          proc(n, x, y, z) begin
                            [(1 + x)/2, (1 + y)/2, (1 + z)/2]
                          end_proc),
     plot::Dodecahedron(Center = [3, 0, 0],
                        FillColorFunction =
                          proc(n, x, y, z) begin
                             mycolorlist[(n mod 4)+1]
                          end_proc),
     Axes = None):
```



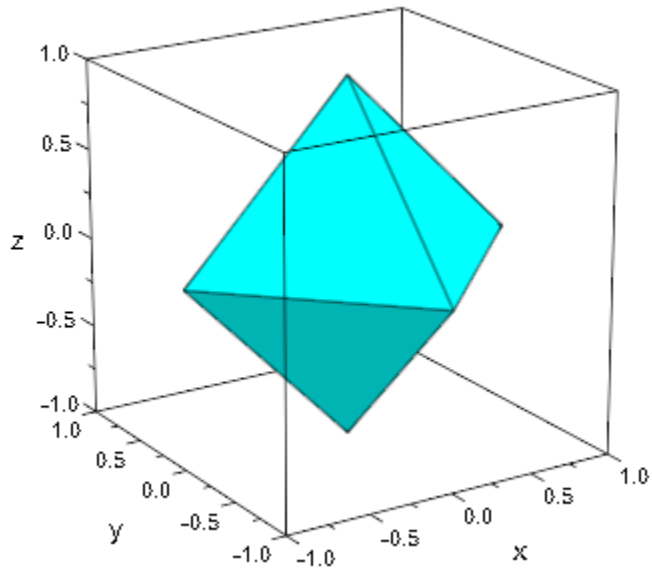
The same holds true for a `LineColorFunction`:

```
plot(plot::Icosahedron(Center = [0, 0, 0],
                       LineColorFunction =
                          proc(n, x, y, z) begin
                            [(1 + x)/2, (1 + y)/2, (1 + z)/2]
                          end_proc),
     plot::Icosahedron(Center = [3, 0, 0],
                       LineColorFunction =
                           proc(n, x, y, z) begin
                              mycolorlist[(n mod 4)+1]
                           end_proc),
     Axes = None, LineWidth = 1.0*unit::mm,
     FillColor = RGB::Grey80, FillColorType = Flat):
```



If the polyhedron is animated, the color functions are called with an additional argument: the current value of the animation parameter:

```
plot(plot::Octahedron(FillColorFunction =
                        proc(n, x, y, z, a)
                        begin
                          [sin(n*a)^2, cos(n*a)^2, 1]:
                        end_proc,
                      a = 0..2*PI))
```




```
delete mycolorlist:
```

Algorithms

A polyhedron is called regular if all its facets consist of the same regular polygon and each vertex has the same number of coincidence polygons.

Since Plato we know that only five regular polyhedrons exist:

- the tetrahedron with 4 (greek *tetra*) triangles,
- the hexahedron with 6 (greek *hexa*) squares,
- the octahedron with 8 (greek *okta*) triangles,
- the dodecahedron with 12 (greek *dodeka*) pentagons and
- the icosahedron with 20 (greek *eikosi*) triangles.

The following table lists some important geometrical data of the polyhedra with the edge length a. Where R is the radius of the outer sphere, r the radius of the inner sphere, A the surface area and V the volume:

Ratio	Tetrahedron	Hexahedron	Octahedron	Dodecahedron	Icosahedron
$\frac{R}{a}$	$\frac{\sqrt{6}}{4}$	$\frac{\sqrt{3}}{2}$	$\frac{\sqrt{2}}{2}$	$\frac{\sqrt{3}\,(1+\sqrt{5})}{4}$	$\frac{\sqrt{10+2\,\sqrt{5}}}{4}$
$\frac{r}{a}$	$\frac{\sqrt{6}}{12}$	$\frac{1}{2}$	$\frac{\sqrt{6}}{6}$	$\frac{\sqrt{250+110\,\sqrt{5}}}{20}$	$\frac{\sqrt{3}\,(3+\sqrt{5})}{12}$
$\frac{A}{a^2}$	$\sqrt{3}$	6	$2\,\sqrt{3}$	$3\,\sqrt{25+10\,\sqrt{5}}$	$5\,\sqrt{3}$
$\frac{V}{a^3}$	$\frac{\sqrt{2}}{12}$	1	$\frac{\sqrt{2}}{3}$	$\frac{(15+7\,\sqrt{5})}{4}$	$\frac{5\,(3+\sqrt{5})}{12}$

See Also

MuPAD Functions
`plot`

MuPAD Graphical Primitives
`plot::Box` | `plot::Cone` | `plot::Cylinder` | `plot::Dodecahedron` | `plot::Hexahedron` | `plot::Octahedron` | `plot::Parallelogram3d` | `plot::Sphere` | `plot::Tetrahedron` | `plot::Transform3d`

plot::Text2d

2D text

Syntax

`plot::Text2d(text, [x, y], <a = a_min .. a_max>, options)`

Description

`plot::Text2d` draws a text at a given position (x, y) (the "anchor point"). The attributes `VerticalAlignment` and `HorizontalAlignment` determine the alignment of the text w.r.t. its anchor.

Size, text color, font type etc. are controlled by the attribute `TextFont`.

A text may consist of several lines. The newline character in MuPAD strings ist `\n`. For example: `"first line\nsecond line"`.

The attribute `TextRotation` allows to rotate the text on the screen.

The text of a text object can be animated if it is passed as a procedure that returns the text string during runtime. Cf. "Example 5" on page 24-1031.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Frames`	the number of frames in an animation	`50`
`HorizontalAlignment`	horizontal alignment of text objects w.r.t. their coordinates	`Left`
`Legend`	makes a legend entry	

Attribute	Purpose	Default Value
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
Position	positions of cameras, lights, and text objects	
PositionX	x-positions of cameras, lights, and text objects	
PositionY	y-positions of cameras, lights, and text objects	
Text	the text of a text object	
TextFont	font of text objects	[" sans-serif ", 11]
TextRotation	rotation of a 2D text	0
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	

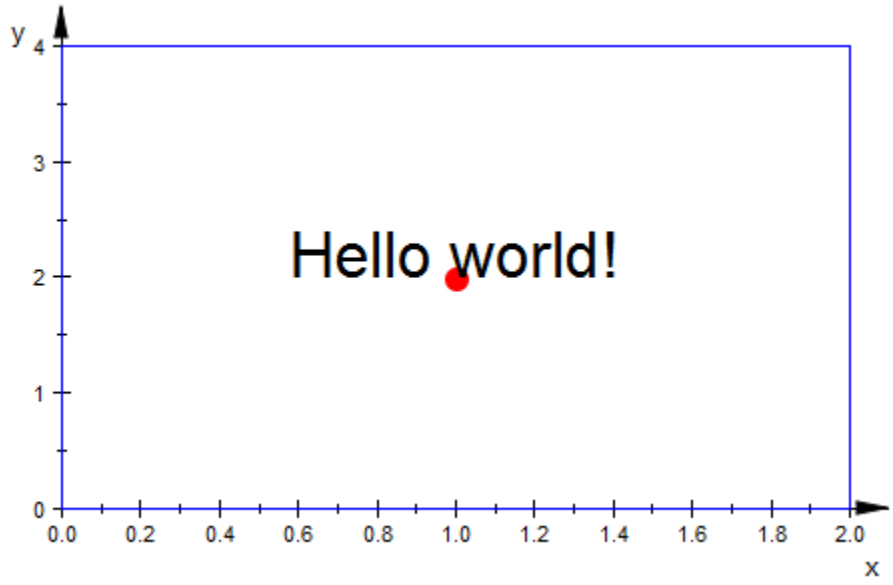
Attribute	Purpose	Default Value
`TitleAlignment`	horizontal alignment of titles w.r.t. their coordinates	`Center`
`TitlePositionX`	position of object titles, x component	
`TitlePositionY`	position of object titles, y component	
`VerticalAlignment`	vertical alignment of text objects w.r.t. their coordinates	`BaseLine`
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`

Examples

Example 1

We draw the text string `Hello world!' at the anchor point (1, 2) which is indicated by a red dot:

```
plot(plot::Rectangle(0..2, 0..4),
     plot::Point2d([1, 2]),
     plot::Text2d("Hello world!", [1, 2],
                  HorizontalAlignment = Center),
     Axes = Frame, TextFont = [24],
     PointColor = RGB::Red, PointSize = 3*unit::mm)
```

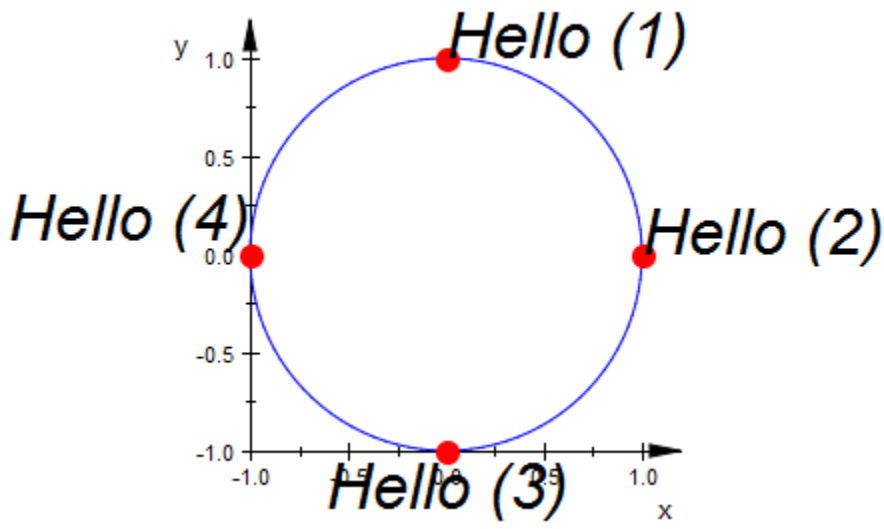


Example 2

We animate the anchor points of the following texts and demonstrate various alignment possibilities:

```
plot(plot::Circle2d(1),
     plot::Point2d([sin(a), cos(a)], a = 0..2*PI),
     plot::Point2d([cos(a), -sin(a)], a = 0..2*PI),
     plot::Point2d([-sin(a), -cos(a)], a = 0..2*PI),
     plot::Point2d([-cos(a), sin(a)], a = 0..2*PI),
     PointColor = RGB::Red, PointSize = 3*unit::mm,
     plot::Text2d("Hello (1)", [sin(a), cos(a)], a = 0..2*PI),
     plot::Text2d("Hello (2)", [cos(a), -sin(a)], a = 0..2*PI,
                  HorizontalAlignment = Left,
                  VerticalAlignment = BaseLine),
     plot::Text2d("Hello (3)", [-sin(a), -cos(a)], a = 0..2*PI,
                  HorizontalAlignment = Center,
                  VerticalAlignment = Top),
     plot::Text2d("Hello (4)", [-cos(a), sin(a)], a = 0..2*PI,
                  HorizontalAlignment = Right,
                  VerticalAlignment = Bottom),
```

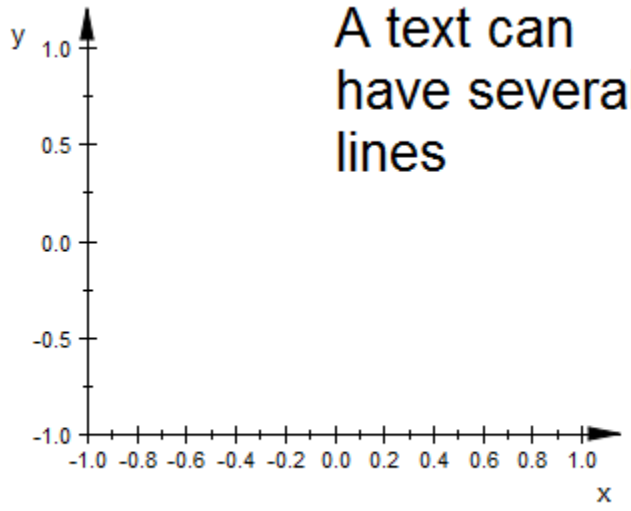
```
     TextFont = [Italic, 24], Axes = Frame)
```



Example 3

A text may consist of several lines. The newline character in MuPAD strings is `\n`:

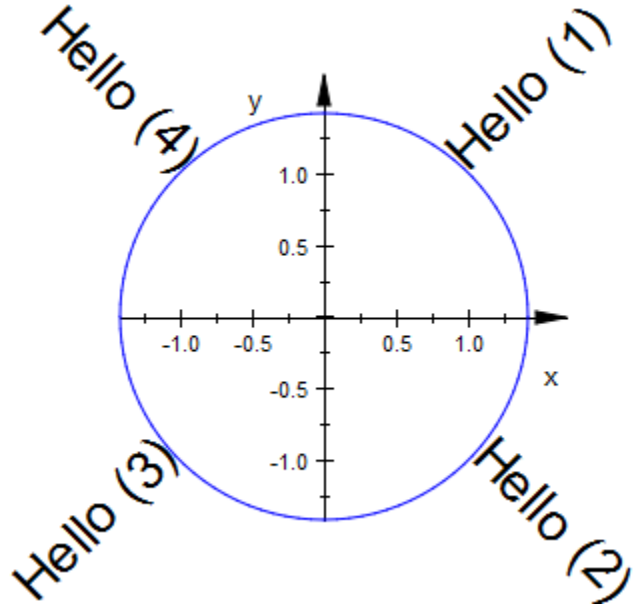
```
plot(plot::Text2d("A text can\nhave several\nlines",
                  [sin(a), cos(a)], a = 0..2*PI),
     Axes = Frame, TextFont = [20])
```



Example 4

The attribute `TextRotation` allows to rotate a 2D text on the screen:

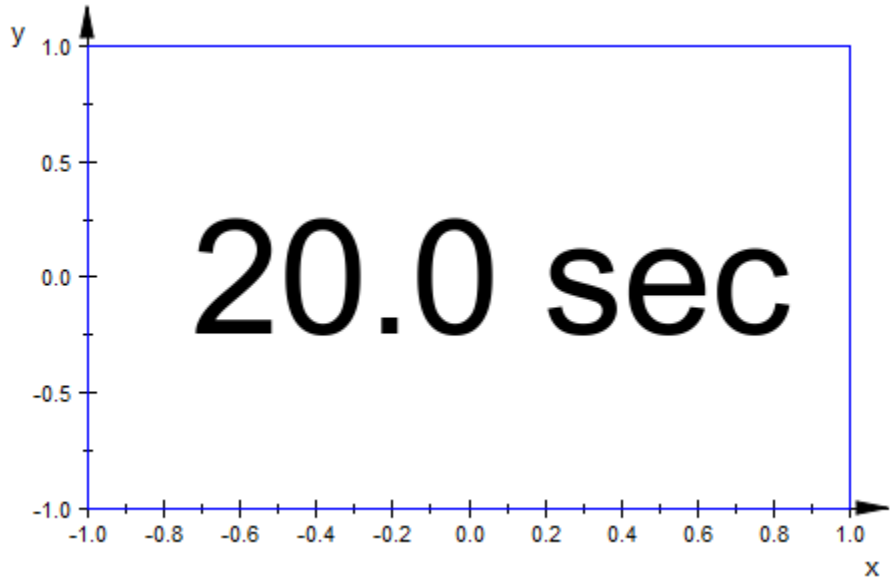
```
plot(plot::Circle2d(sqrt(2)),
     plot::Text2d("Hello (1)", [ 1, 1],
                  HorizontalAlignment = Left,
                  TextRotation = PI/4),
     plot::Text2d("Hello (2)", [ 1,-1],
                  HorizontalAlignment = Left,
                  TextRotation = -PI/4),
     plot::Text2d("Hello (3)", [-1,-1],
                  HorizontalAlignment = Right,
                  TextRotation = PI/4),
     plot::Text2d("Hello (4)", [-1, 1],
                  HorizontalAlignment = Right,
                  TextRotation = -PI/4),
     HorizontalAlignment = Left, TextFont = [20])
```



Example 5

The text of a text object can be animated if the text string is provided by a procedure. We use `stringlib::formatf` to format the animation parameter that is passed to the procedure as a floating-point number for each frame of the animation:

```
plot(plot::Rectangle(-1..1, -1..1),
     plot::Text2d(a -> stringlib::formatf(a, 2, 5)." sec",
                  [0, 0], a = 0..20),
     TextFont = [60],
     HorizontalAlignment = Center, VerticalAlignment = Center,
     Axes = Frame, Frames = 201, TimeRange = 0..20)
```



Parameters

`text`

The text: a string. Alternatively, a procedure that accepts one input parameter *a* (the animation parameter) and returns a string.

text is equivalent to the attribute `Text`.

`x, y`

The position of the text. The coordinates `x` and `y` must be real numerical values or arithmetical expressions of the animation parameter `a`.

x, y are equivalent to the attributes `Position`, `PositionX`, `PositionY`.

`a`

Animation parameter, specified as a = $a_{min}..a_{max}$, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy` | `stringlib::formatf`

MuPAD Graphical Primitives
`plot::Text3d`

plot::Text3d

3D text

Syntax

```
plot::Text3d(text, [x, y, z], <a = a_min .. a_max>, options)
```

Description

`plot::Text3d` draws a text at a given position (*x*, *y*, *z*) (the "anchor point"). The attributes `VerticalAlignment` and `HorizontalAlignment` determine the alignment of the text w.r.t. its anchor.

Size, text color, font type etc. are controlled by the attribute `TextFont`.

In contrast to `plot::Text2d`, a 3D text cannot consist of several lines. The newline character `\n` in MuPAD strings does not have an effect.

By default, a 3D text uses `Billboarding` = `TRUE`, i.e., the text is automatically oriented such that it is readable by the observer. When setting `Billboarding` = `FALSE`, the attribute `TextOrientation` allows to fix the orientation of the text arbitrarily in space. See the help page of `TextOrientation` for details.

The text of a text object can be animated if it is passed as a procedure that returns the text string during runtime. Cf. "Example 5" on page 24-1041.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Billboarding`	text orientation in space or towards observer	`TRUE`
`Frames`	the number of frames in an animation	`50`

Attribute	Purpose	Default Value
HorizontalAlignment	horizontal alignment of text objects w.r.t. their coordinates	Left
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
Position	positions of cameras, lights, and text objects	
PositionX	x-positions of cameras, lights, and text objects	
PositionY	y-positions of cameras, lights, and text objects	
PositionZ	z-positions of cameras, lights, and text objects	
Text	the text of a text object	
TextFont	font of text objects	[" sans-serif ", 11]
TextOrientation	orientation of a 3D text	[1, 0, 0, 0, 0, 1]
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0

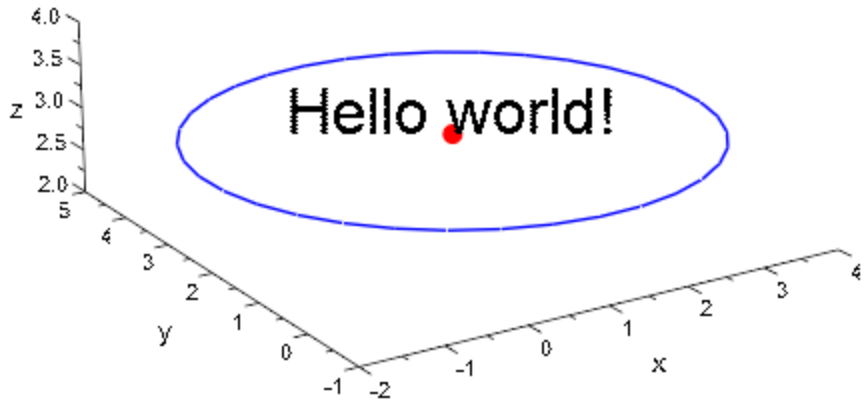
Attribute	Purpose	Default Value
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
VerticalAlignment	vertical alignment of text objects w.r.t. their coordinates	BaseLine
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE

Examples

Example 1

We draw the text string `Hello world' at the anchor point (1, 2, 3) which is indicated by a red dot:

```
plot(plot::Circle3d(3, [1, 2, 3], [0, 0, 1]),
     plot::Point3d([1, 2, 3]),
     plot::Text3d("Hello world!", [1, 2, 3],
                  HorizontalAlignment = Center),
     Axes = Frame, TextFont = [24],
     PointColor = RGB::Red, PointSize = 3*unit::mm)
```

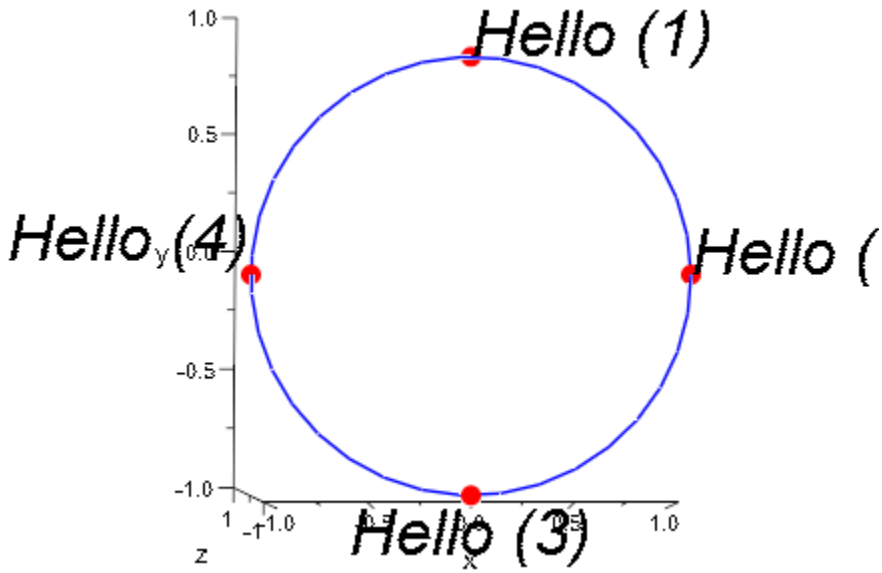


Example 2

We animate the anchor points of the following texts and demonstrate various alignment possibilities:

```
plot(plot::Circle3d(1, [0, 0, 0], [0, 0, 1]),
     plot::Point3d([sin(a), cos(a), 0], a = 0..2*PI),
```

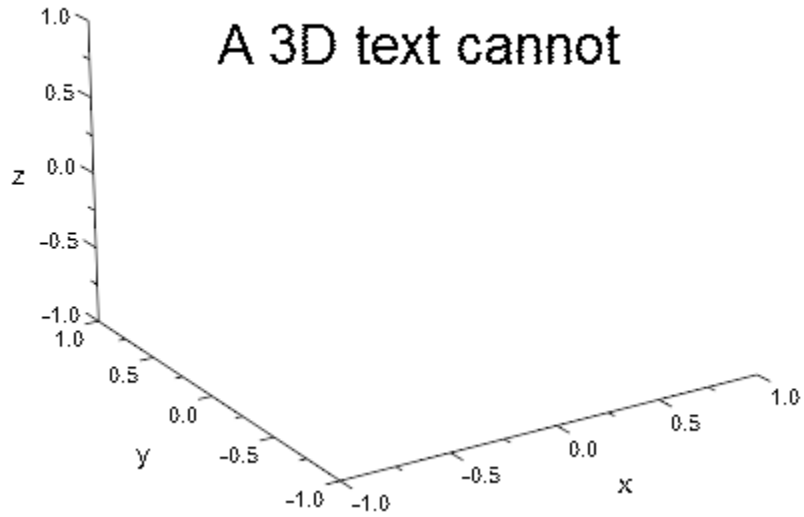
```
plot::Point3d([cos(a), -sin(a), 0], a = 0..2*PI),
plot::Point3d([-sin(a), -cos(a), 0], a = 0..2*PI),
plot::Point3d([-cos(a), sin(a), 0], a = 0..2*PI),
PointColor = RGB::Red, PointSize = 3*unit::mm,
plot::Text3d("Hello (1)", [sin(a), cos(a), 0], a = 0..2*PI),
plot::Text3d("Hello (2)", [cos(a), -sin(a), 0], a = 0..2*PI,
             HorizontalAlignment = Left,
             VerticalAlignment = BaseLine),
plot::Text3d("Hello (3)", [-sin(a), -cos(a), 0], a = 0..2*PI,
             HorizontalAlignment = Center,
             VerticalAlignment = Top),
plot::Text3d("Hello (4)", [-cos(a), sin(a), 0], a = 0..2*PI,
             HorizontalAlignment = Right,
             VerticalAlignment = Bottom),
TextFont = [Italic, 24], Axes = Frame,
CameraDirection = [0, -1, 10])
```



Example 3

In contrast to `plot::Text2d`, a 3D text may not consist of several lines. The newline character `\n` in MuPAD strings does not have any effect:

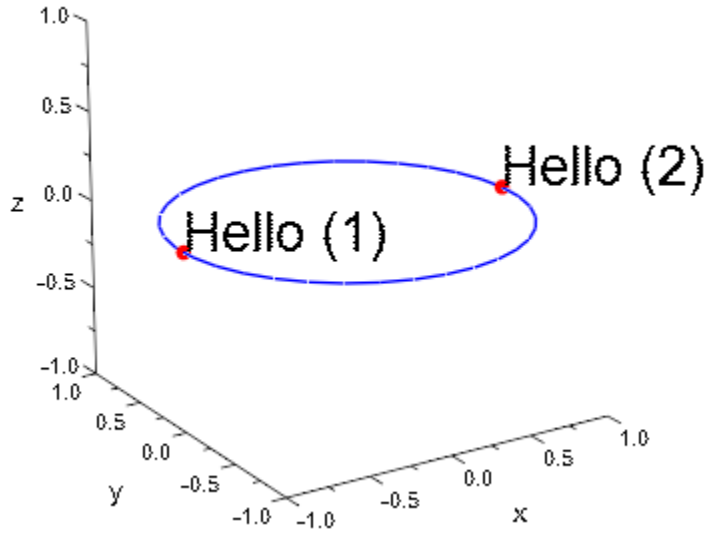
```
plot(plot::Text3d("A 3D text cannot \nhave several\nlines",
                  HorizontalAlignment = Center,
                  [0, 0, 0]),
     Axes = Frame, TextFont = [20])
```



Example 4

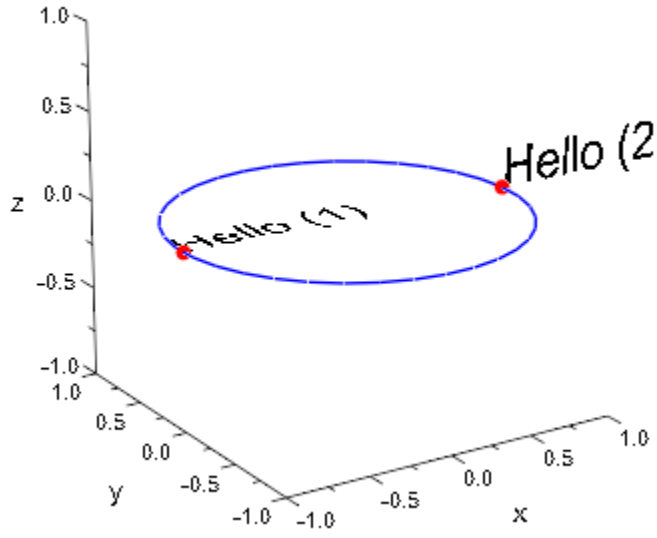
Per default, the attribute `Billboarding` = `TRUE` is set. The text always faces the observer:

```
plot(plot::Circle3d(1, [0, 0, 0], [0, 0, 1]),
     plot::Point3d([-cos(a), -sin(a), 0], a = 0 .. 2*PI),
     plot::Point3d([cos(a), sin(a), 0], a = 0 .. 2*PI),
     plot::Text3d("Hello (1)", [-cos(a), -sin(a), 0],
                  a = 0 .. 2*PI),
     plot::Text3d("Hello (2)", [cos(a), sin(a), 0],
                  a = 0 .. 2*PI),
     Axes = Frame, TextFont = [20],
     PointColor = RGB::Red, PointSize = 2*unit::mm)
```



We use `TextOrientation` to fix the orientation of the texts in space. The first text lies in a plane parallel to the *x*-*y* plane, the second text is parallel to the *x*-*z* plane. Note that we have to specify `Billboarding` = `FALSE` for `TextOrientation` to have an effect:

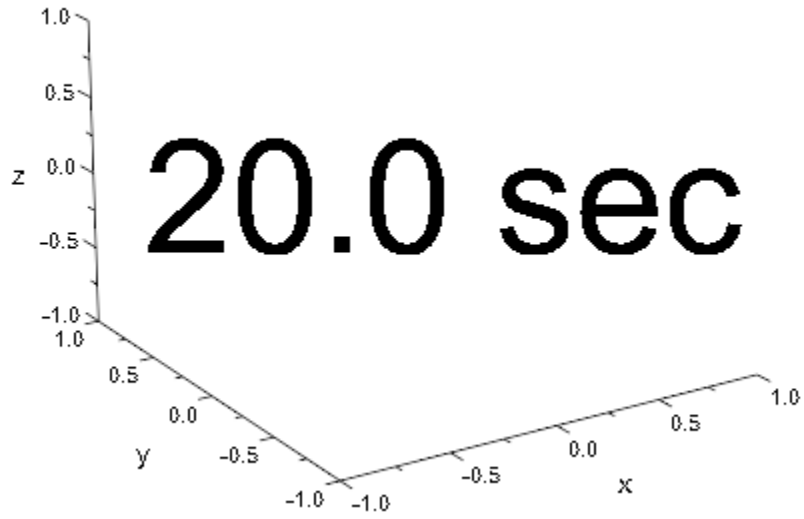
```
plot(plot::Circle3d(1, [0, 0, 0], [0, 0, 1]),
     plot::Point3d([-cos(a), -sin(a), 0], a = 0 .. 2*PI),
     plot::Point3d([cos(a), sin(a), 0], a = 0 .. 2*PI),
     PointColor = RGB::Red, PointSize = 2*unit::mm,
     plot::Text3d("Hello (1)", [-cos(a), -sin(a), 0],
                  a = 0 .. 2*PI,
                  TextOrientation = [1, 0, 0, 0, 1, 0]),
     plot::Text3d("Hello (2)", [cos(a), sin(a), 0],
                  a = 0 .. 2*PI,
                  TextOrientation = [1, 0, 0, 0, 0, 1]),
     Billboarding = FALSE, TextFont = [20], Axes = Frame)
```



Example 5

The text of a text object can be animated if the text string is provided by a procedure. We use `stringlib::formatf` to format the animation parameter that is passed to the procedure as a floating-point number for each frame of the animation:

```
plot(plot::Text3d(a -> stringlib::formatf(a, 2, 5)." sec",
                  [0, 0, 0], a = 0..20),
     TextFont = [60],
     HorizontalAlignment = Center, VerticalAlignment = Center,
     Axes = Frame, Frames = 201, TimeRange = 0..20)
```



Parameters

`text`

The text: a string. Alternatively, a procedure that accepts one input parameter *a* (the animation parameter) and returns a string.

text is equivalent to the attribute `Text`.

`x, y, z`

The position of the text. The coordinates `x`, `y`, `z` must be real numerical values or arithmetical expressions of the animation parameter `a`.

x, y, z are equivalent to the attributes `Position`, `PositionX`, `PositionY`, `PositionZ`.

`a`

Animation parameter, specified as a = $a_{min}..a_{max}$, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy` | `stringlib::formatf`

MuPAD Graphical Primitives
`plot::Text2d`

plot::Tube

Generalized tubular plots (canal surfaces)

Syntax

```
plot::Tube([x, y, z], <r>, t = t_min .. t_max, <a = a_min .. a_max>, options)
```

Description

`plot::Tube` creates generalized tubular plots, known as “canal surfaces”, with special cases known as “tube surface”, “pipe surface” or “tubular surfaces.”

Intuitively, canal surfaces are space curves with thickness. More formally, a canal surface `plot::Tube([x(t), y(t), z(t)], r(t), t = t_min..t_max)` is the envelope of spheres with center $[x(t), y(t), z(t)]$ and radius $r(t)$, i.e., the thickness of the curve can vary with the curve parameter t.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AngleEnd`	end of angle range	`2*PI`
`AngleBegin`	begin of angle range	`0`
`AngleRange`	angle range	`0 .. 2*PI`
`Filled`	filled or transparent areas and surfaces	`TRUE`
`FillColor`	color of areas and surfaces	`RGB::Red`
`FillColor2`	second color of areas and surfaces for color blends	`RGB::CornflowerBlue`
`FillColorType`	surface filling types	`Dichromatic`

Attribute	Purpose	Default Value
FillColorFunction	functional area/surface coloring	
FillColorDirection	the direction of color transitions on surfaces	[0, 0, 1]
FillColorDirectionX	x-component of the direction of color transitions on surfaces	0
FillColorDirectionY	y-component of the direction of color transitions on surfaces	0
FillColorDirectionZ	z-component of the direction of color transitions on surfaces	1
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	TRUE
LineColor	color of lines	RGB::Black.[0.25]
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::DeepPink
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	0

Attribute	Purpose	Default Value
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Mesh	number of sample points	[60, 11]
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointsVisible	visibility of mesh points	FALSE
RadiusFunction	radius of a tube plot	1/10
Shading	smooth color blend of surfaces	Smooth
Submesh	density of submesh (additional sample points)	[0, 1]
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	

Attribute	Purpose	Default Value
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
ULinesVisible	visibility of parameter lines (u lines)	TRUE
UMax	final value of parameter “u”	
UMesh	number of sample points for parameter “u”	60
UMin	initial value of parameter “u”	
UName	name of parameter “u”	
URange	range of parameter “u”	
USubmesh	density of additional sample points for parameter “u”	0
VLinesVisible	visibility of parameter lines (v lines)	TRUE
VMesh	number of sample points for parameter “v”	11
VSubmesh	density of additional sample points for parameter “v”	1
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE

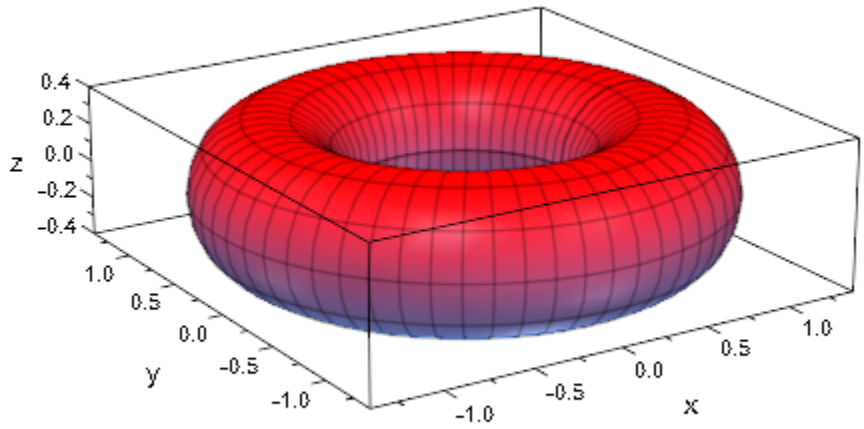
Attribute	Purpose	Default Value
XFunction	function for x values	
YFunction	function for y values	
ZFunction	function for z values	

Examples

Example 1

A torus can be drawn as a tube around a circle:

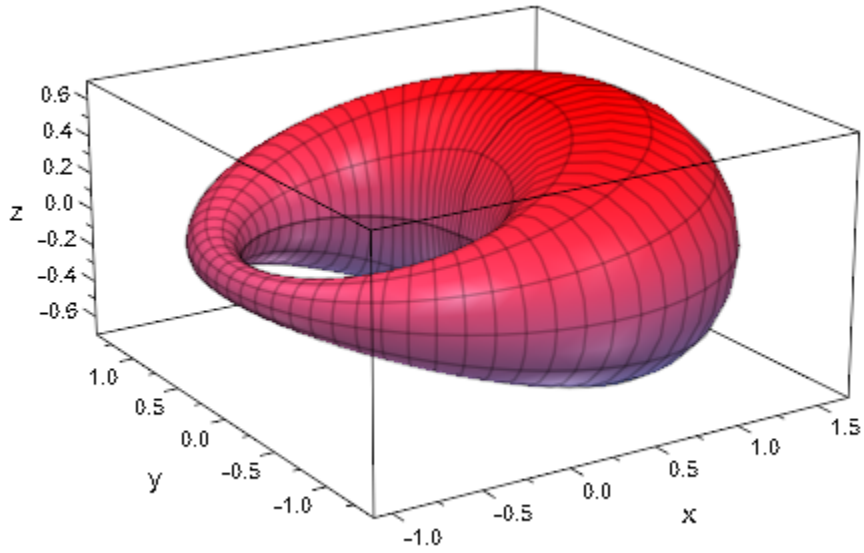
```
plot(plot::Tube([cos(t), sin(t), 0], 0.4,
     t = 0..2*PI))
```



Varying the diameter of the tube, we deform the torus into a cyclide:

```
plot(plot::Tube([cos(t), sin(t), 0], 0.4 + 0.3*cos(t),
```

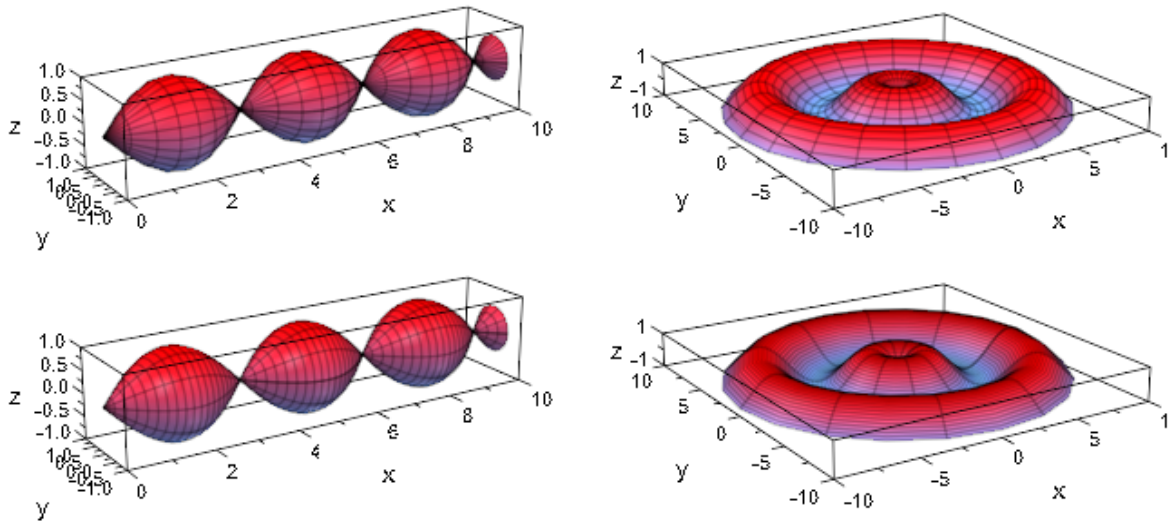
```
     t = 0..2*PI))
```



Example 2

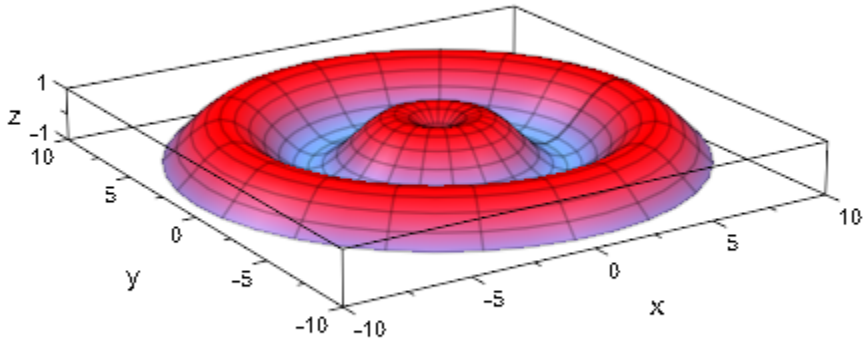
All surfaces of revolution are special cases of canal surfaces:

```
plot(plot::Scene3d(plot::XRotate(sin(u), u = 0..10)),
     plot::Scene3d(plot::ZRotate(sin(u), u = 0..10)),
     plot::Scene3d(plot::Tube([u, 0, 0], sin(u), u = 0..10)),
     plot::Scene3d(plot::Tube([0, 0, sin(u)], u, u = 0..10)),
     Width = 180 * unit::mm)
```



The last image shows that the defaults for the mesh are not always adequate and should be changed:

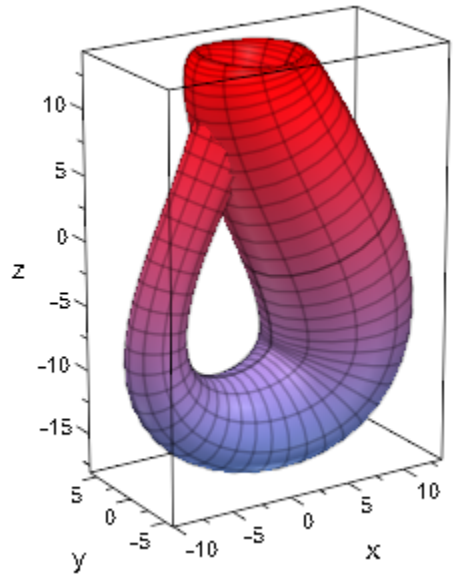
```
plot(plot::Tube([0, 0, sin(u)], u, u = 0..10,
                Mesh = [20, 20]))
```



Example 3

The famous Klein bottle can be obtained from a “drop silhouette” by using an appropriate radius parametrization:

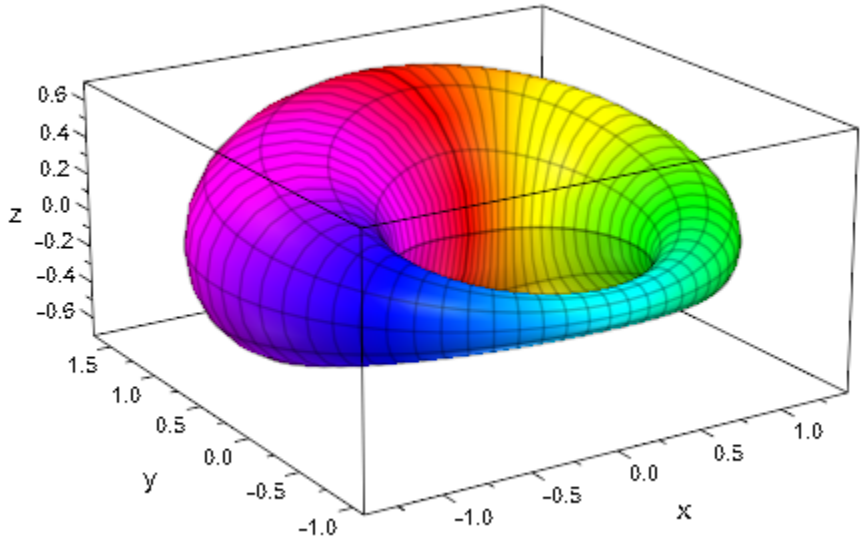
```
plot(plot::Tube([6*cos(u)*(sin(u) - 1), 0, 14*sin(u)],
                4 - 2*cos(u), u = -PI..PI))
```



Example 4

Re-using the cyclide from above, we demonstrate coloring a canal surface:

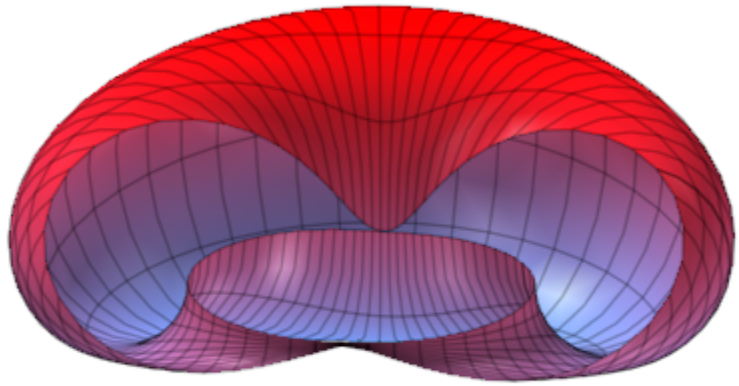
```
color := (t, phi) -> RGB::fromHSV([(t+sin(4*phi))*180/PI, 1, 1]):
plot(plot::Tube([sin(t), cos(t), 0], 0.4 + 0.3*cos(t), t=0..2*PI,
                FillColorFunction = color))
```



Example 5

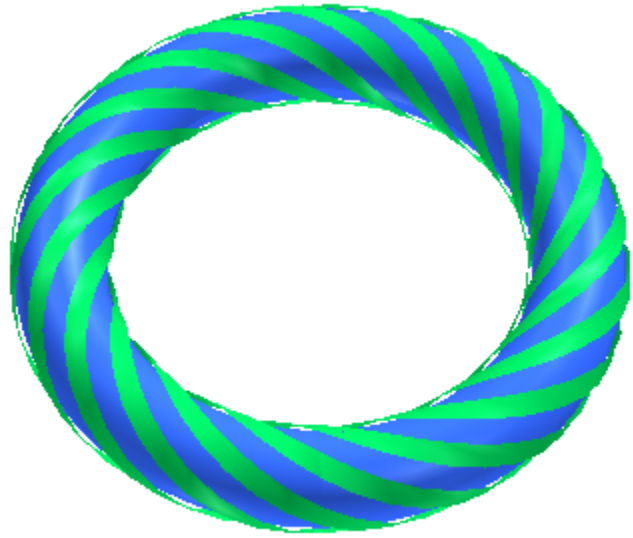
Yet another variation of the cyclide, we use a non-constant `AngleRange` to "slice" it:

```
plot(plot::Tube([sin(t), cos(t), 0], 0.4 - 0.3*sin(t), t=0..2*PI,
                AngleRange = 0 .. 2*PI*sin(abs(t-PI/2)/2)),
     Axes = None, CameraDirection = [14, 1, 5])
```



Combining more than one tubular plot with identical spine curves but different angle ranges, we can achieve a braid-like effect:

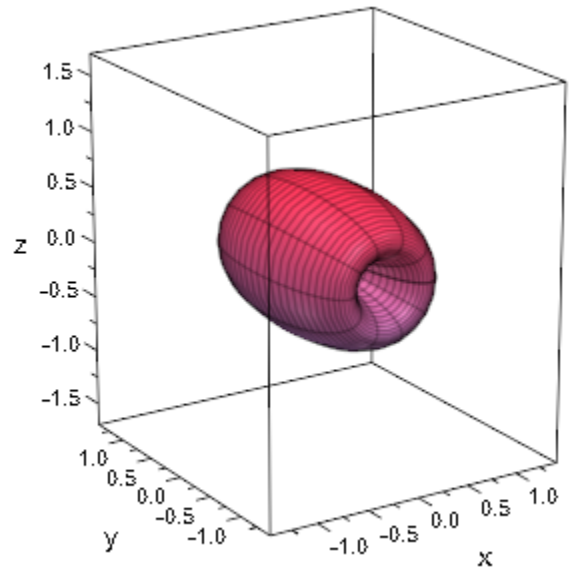
```
braid := i ->
   plot::Tube([sin(u), cos(u), 0], 0.2, u=0..2*PI,
              AngleRange = i*PI/3 + 3*u .. i*PI/3 + 3*u + 1/2,
              Color = RGB::EmeraldGreen, Mesh = [60, 2]):
torus := plot::Tube([sin(u), cos(u), 0], 0.18, u=0..2*PI,
                    Color = RGB::BlueLight,
                    Name = "Torus"):
plot(braid(i) $ i = 0..5,
     torus,
     ULinesVisible = FALSE, VLinesVisible = FALSE,
     FillColorType = Flat,
     Axes = None, CameraDirection = [0, 7, 10])
```



Example 6

The spine curve, the radius function, color functions etc. can be animated as usual:

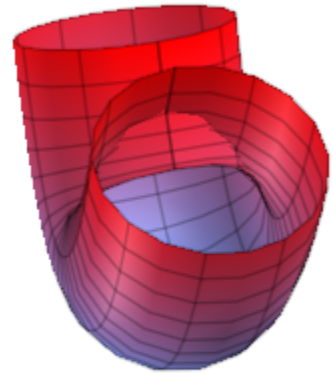
```
plot(plot::Tube([sin(t)*sin(a), cos(t)*cos(a), sin(a)],
                0.4 - 0.3*sin(t-a),
                t = 0..2*PI, a = 0..2*PI,
                Frames = 20, TimeRange = 0..5))
```



Example 7

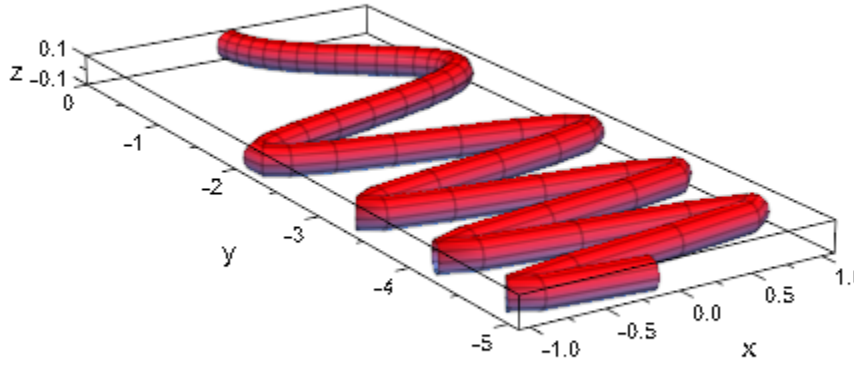
Note that in the presence of a sharp bend (in relation to the tube diameter), the surface plotted my `plot::Tube` may self-intersect:

```
plot(plot::Tube([x, 0, x^2], 1.2, x = -1.4..1.4,
                Mesh = [20, 10]),
     Axes = None, CameraDirection = [-3, 1, 2])
```



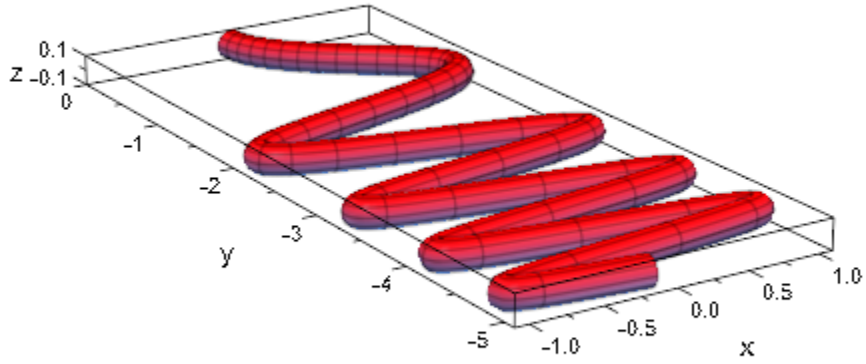
This effect is unavoidable. Sharp bends also cause another effect which can be avoided by increasing the mesh density: The tube might not follow the curve quickly enough:

```
plot(plot::Tube([sin(x^2), x, 0], x = -5..0))
```



In this situation, you can set `USubmesh` to a positive value to request additional function evaluations:

```
plot(plot::Tube([sin(x^2), x, 0], x = -5..0, USubmesh = 5))
```



Parameters

x, y, z

The spine curve coordinates: real-valued expressions in `t` and the animation parameter.

x, y, z are equivalent to the attributes `XFunction`, `YFunction`, `ZFunction`.

r

The tube radius: a real-valued expression in `t` and the animation parameter. Default is the constant $\frac{1}{10}$.

r is equivalent to the attribute `RadiusFunction`.

t

The curve parameter: an (indexed) identifier.

t is equivalent to the attribute `UName`.

t_{min} .. t_{max}

The range of the curve parameter: real-valued expressions in the animation parameter.

t_{min} .. t_{max} is equivalent to the attributes `URange`, `UMin`, `UMax`.

a

Animation parameter, specified as a = a_{min}..a_{max}, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Curve3d` | `plot::Surface`

plot::Turtle

"turtle graphics" (imperative drawings)

Syntax

```
plot::Turtle(commands, <a = a_min .. a_max>, options)
```

Description

Turtle graphics define a line drawing by a sequence of commands to an abstract robot.

`plot::Turtle` defines a graphic by sending movement commands to an abstract robot. This robot starts heading up and standing at the origin, with its pen ready for drawing ("down") and the line color taken from the attribute `LineColor`.

The following commands are known to the robot:

- `Left(α)`

  Turn left by the angle α (in radians).
- `Right(α)`

  Turn right by the angle α (in radians).
- `Forward(d)`

  Move forward distance d.
- `Up`

  Lift the "pen", i.e., subsequent movement commands do not draw lines.
- `Down`

  Lower the "pen", i.e., subsequent movement commands do draw lines.
- `Push`

  Remember the current state (position, angle, line color).
- `Pop`

Restore the last remembered state and remove it from the list of remembered states.

- `Noop`

  This command is ignored.

- `LineColor(c)`

  Set the line color to the color`c`.

The commands not taking an argument may also be entered with empty parentheses `()` after, e.g., `Push()`.

A `plot::Turtle`-object can be manipulated dynamically by calling its methods `left`, `right`, `forward`, `penUp`, `penDown`, `push`, `pop`, and `setLineColor`, with the obvious connections to the commands above. These methods append a new command to the end of the list. Cf. “Example 3” on page 24-1069.

Note: For long command sequences, it is highly recommended to give the commands directly using the syntax above or by setting the `CommandList` attribute directly.

Both angles and distances can be animated. Colors can not.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`
`Color`	the main color	`RGB::Blue`
`CommandList`	turtle movement commands	[]
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	

Attribute	Purpose	Default Value
`LegendEntry`	add this object to the legend?	`FALSE`
`LineColor`	color of lines	`RGB::Blue`
`LineWidth`	width of lines	`0.35`
`LineStyle`	solid, dashed or dotted lines?	`Solid`
`LinesVisible`	visibility of lines	`TRUE`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`PointSize`	the size of points	`1.5`
`PointStyle`	the presentation style of points	`FilledCircles`
`PointsVisible`	visibility of mesh points	`FALSE`
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`
`Title`	object title	
`TitleFont`	font of object titles	`[" sans-serif ", 11]`
`TitlePosition`	position of object titles	
`TitleAlignment`	horizontal alignment of titles w.r.t. their coordinates	`Center`

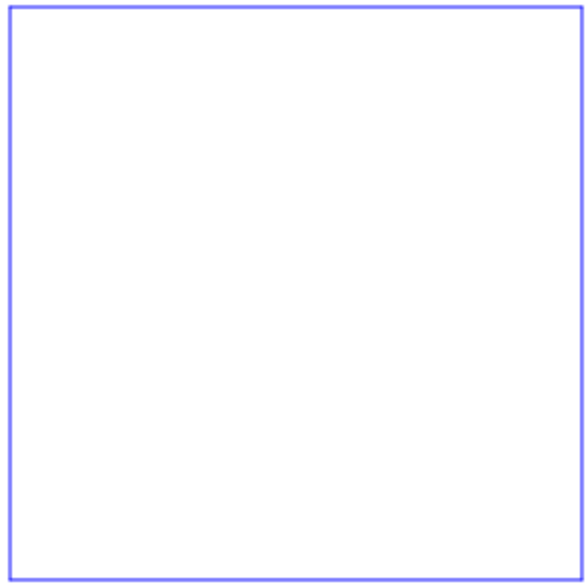
Attribute	Purpose	Default Value
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE

Examples

Example 1

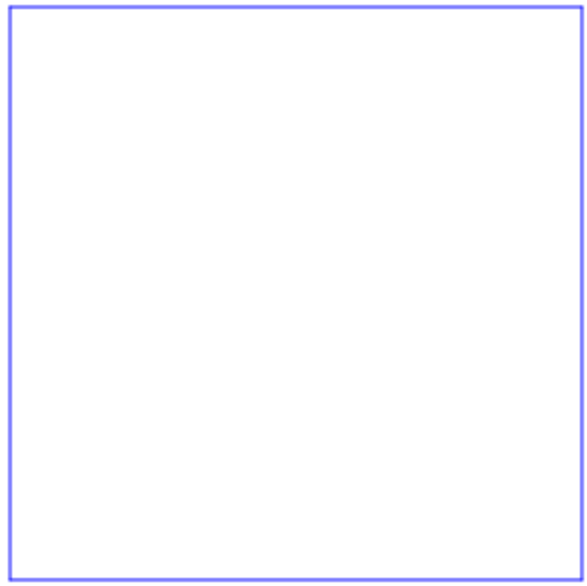
A square can be drawn by four times moving forward, each time turning right 90°:

```
plot(plot::Turtle([Forward(1), Right(PI/2),
                   Forward(1), Right(PI/2),
                   Forward(1), Right(PI/2),
                   Forward(1), Right(PI/2)]))
```



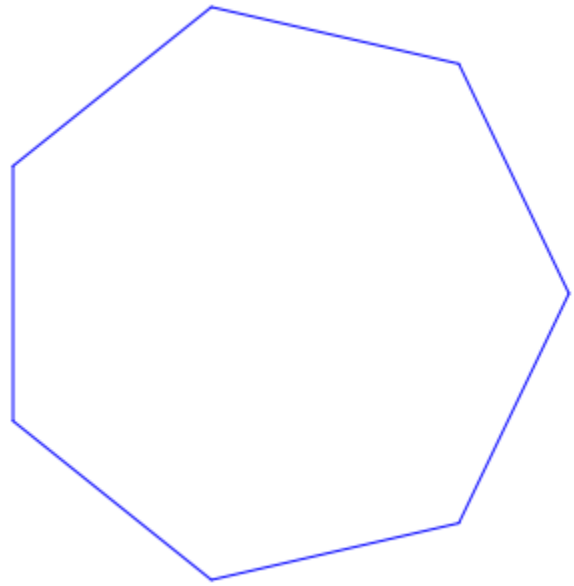
Using the $ operator, this command list can be written much shorter:

```
plot(plot::Turtle([(Forward(1), Right(PI/2))$4]))
```



In the same fashion, we can draw any regular n-sided polygon:

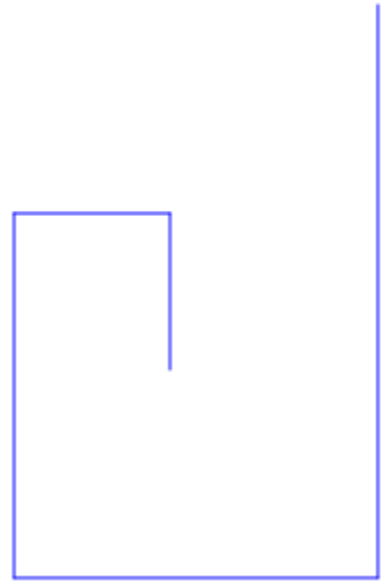
```
n := 7:
plot(plot::Turtle([(Forward(1), Right(2*PI/n)) $ n]))
```



Example 2

The distance to move may contain an animation parameter:

```
plot(plot::Turtle([Forward(1+a), Right(PI/2),
                   Forward(1-2*a), Right(PI/2),
                   Forward(1+3*a), Right(PI/2),
                   Forward(1-4*a), Right(PI/2),
                   Forward(1+5*a)], a=0..2))
```



Likewise, the angle can be animated:

```
plot(plot::Turtle([(Forward(1), Right(a))$10],
                  a = 0.25..2.5))
```

Example 3

It is also possible to successively append commands to the list:

```
t := plot::Turtle()
```

```
plot::Turtle([])
```

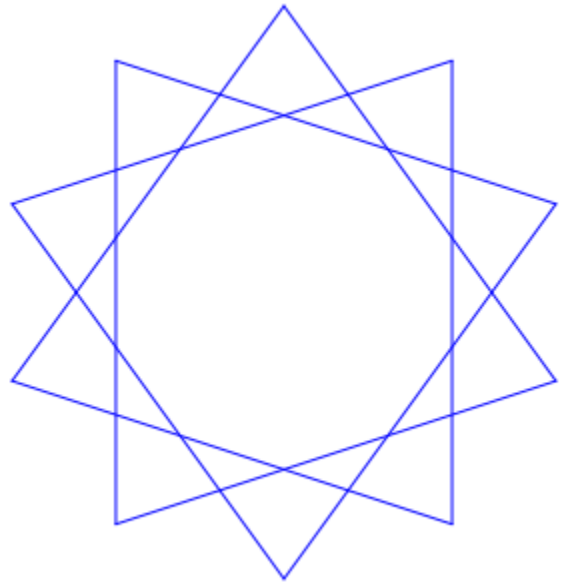
```
t::forward(1)
```

```
plot::Turtle([Forward(1)])
```

```
for i from 1 to 9 do
  t::left(3*PI/5);
  t::forward(1);
end_for
```

```
plot::Turtle([Forward(1), Left((3*PI)/...5), Forward(1)])
```

```
plot(t)
```



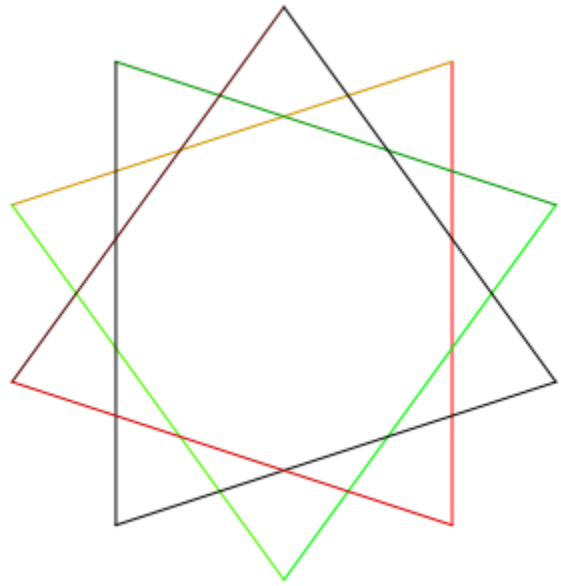
Example 4

As an extension to the original turtle model, the line color may be changed while plotting:

```
t := plot::Turtle():
t::setLineColor(RGB::Red):
t::forward(1):
p := float(PI/5):
for i from 1 to 9 do
  t::left(108*PI/180);
  t::setLineColor([cos(i*p), sin(i*p), 0.0]);
  t::forward(1);
end_for;
```

```
plot::Turtle([LineColor([1.0, 0.0, 0.0...]), Forward(1)])
```

```
plot(t)
```

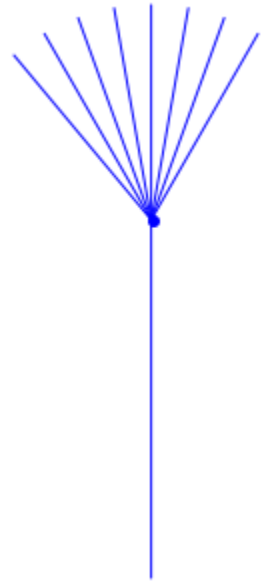


Note that the color within one line segment is constant.

Example 5

Another extension to the turtle model is that `plot::Turtle` supports a stack of saved states, enabling the robot to return to previous positions:

```
t := plot::Turtle():
t::forward(5):
for i from -3 to 4 do
  t::push();
    t::left(PI/18*i);
    t::forward(3);
  t::pop();
end_for:
plot(t)
```



Example 6

Using small steps, it is possible to create appealing curves with `plot::Turtle`:

```
t := plot::Turtle(LineColor = RGB::Green):
t::forward(2):
for dir in [-1, 1] do
  t::push();
    t::left(dir*PI/30);
    for i from 1 to 10 do
      t::forward(0.2);
      t::left(dir*PI/30);
    end_for;
    t::left(dir*2/3*PI);
    for i from 1 to 10 do
      t::forward(0.2);
      t::left(dir*PI/30);
    end_for;
  t::pop()
end_for:
t::forward(3):
t::setLineColor(RGB::Red):
```

```
for dir from -5 to 5 do
  t::push();
    t::left(dir*2*PI/11);
    for i from 1 to 10 do
      t::forward(0.1);
      t::left(PI/30);
    end_for;
    t::left(2*PI/3);
    for i from 1 to 10 do
      t::forward(0.1);
      t::left(PI/30);
    end_for;
  t::pop()
end_for:
plot(t)
```

Parameters

commands

A list of commands. See below for command definitions.

commands is equivalent to the attribute `CommandList`.

a

Animation parameter, specified as a `= a`$_{min}$`..a`$_{max}$, where `a`$_{min}$ is the initial parameter value, and `a`$_{max}$ is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Lsys`

plot::VectorField2d

2D vector field

Syntax

```
plot::VectorField2d([v_1, v_2], x = x_{min} .. x_{max}, y = y_{min} .. y_{max}, <a = a_{min} .. a_{max}>, options
```

```
plot::VectorField2d(v_1, v_2, x = x_{min} .. x_{max}, y = y_{min} .. y_{max}, <a = a_{min} .. a_{max}>, options)
```

Description

`plot::VectorField2d([v_1, v_2], x = `x_{min}`..`x_{max}`, y = `y_{min}`..`y_{max}`)` represents a plot of the vector field defined by $(x, y) \to (v_1(x, y), v_2(x, y))$ with $(x, y) \in [x_min, x_max] \times [y_min, y_max]$.

A vector field is defined by a function $f: \mathbb{R}^2 \to \mathbb{R}^2$. `plot::VectorField2d` displays a vector field by placing arrows at regular intervals with the arrow at (x, y) pointing in direction $f(x, y)$.

The length of the arrows depend on $|f(x, y)|$ and the setting of the attribute `ArrowLength`: By default, arrow lengths are proportional to the magnitude of f, but can be set to be of fixed length or to scale logarithmically.

The density of arrows placed can be controlled with the attributes `XMesh`, `YMesh`, and `Mesh`. See the examples below.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AntiAliased`	antialiased lines and points?	`TRUE`
`ArrowLength`	scaling of arrows in a vector field	`Proportional`
`Color`	the main color	`RGB::Blue`

Attribute	Purpose	Default Value
`Frames`	the number of frames in an animation	`50`
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`FALSE`
`LineColor`	color of lines	`RGB::Blue`
`LineWidth`	width of lines	`0.35`
`LineColor2`	color of lines	`RGB::DeepPink`
`LineColorType`	line coloring types	`Flat`
`LineColorFunction`	functional line coloring	
`LineColorDirection`	the direction of color transitions on lines	[0, 1]
`LineColorDirectionX`	x-component of the direction of color transitions on lines	`0`
`LineColorDirectionY`	y-component of the direction of color transitions on lines	`1`
`Mesh`	number of sample points	[11, 11]
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`

Attribute	Purpose	Default Value
TimeRange	the real time span of an animation	0.0 .. 10.0
TipAngle	opening angle of arrow heads	0.6283185307
TipStyle	presentation style of arrow heads	Open
TipLength	length of arrow heads	1.5
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XFunction	function for x values	
XMax	final value of parameter “x”	
XMesh	number of sample points for parameter “x”	11

Attribute	Purpose	Default Value
XMin	initial value of parameter "x"	
XName	name of parameter "x"	
XRange	range of parameter "x"	
YFunction	function for y values	
YMax	final value of parameter "y"	
YMesh	number of sample points for parameter "y"	11
YMin	initial value of parameter "y"	
YName	name of parameter "y"	
YRange	range of parameter "y"	

Examples

Example 1

We demonstrate a plot of the vector field $v(x, y) = (1, sin(x) + cos(y))$:

```
field := plot::VectorField2d([1, sin(x) + cos(y)],
                             x = 0..6, y = 0..2.5,
                             Mesh = [31, 26]):
```

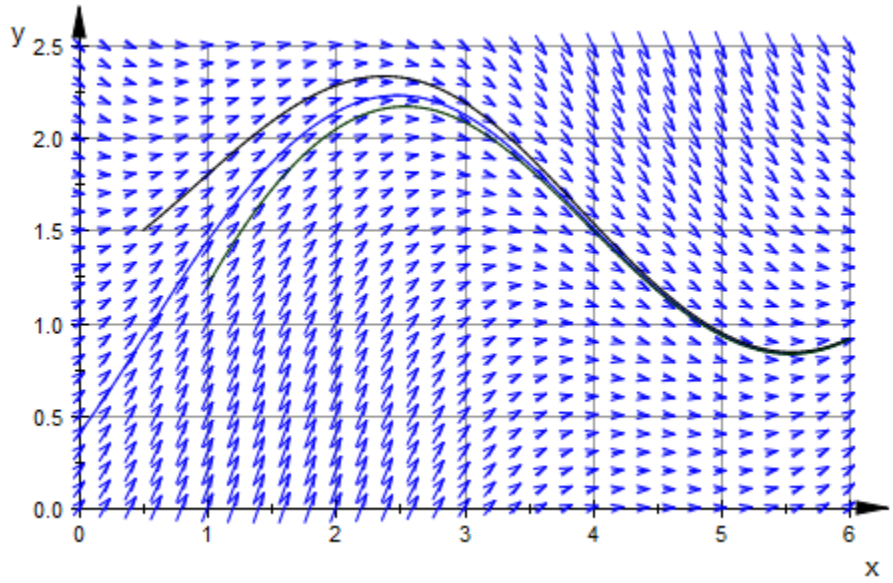
It is the directional field associated with the ode $y'(x) = \sin(x) + \cos(y)$. We insert curves representing numerical solutions of this ode into this plot. We use `numeric::odesolve2` to compute the numerical solutions for the initial values $y(0) = 0.4$, $y(0.5) = 1.5$, and $y(1) = 1.2$:

```
f := (x, y) -> [sin(x) + cos(y[1])]:
solution1 := numeric::odesolve2(f, 0, [0.4]):
curve1 := plot::Function2d(solution1(x)[1], x = 0 .. 6,
                           LineColor = RGB::Blue):
solution2 := numeric::odesolve2(f, 0.5, [1.5]):
curve2 := plot::Function2d(solution2(x)[1], x = 0.5 .. 6,
                           LineColor = RGB::Black):
```

```
solution3 := numeric::odesolve2(f, 1, [1.2]):
curve3 := plot::Function2d(solution3(x)[1], x = 1 .. 6,
                           LineColor = RGB::GreenDark):
```

We plot the three objects in a single graphical scene:

```
plot(field, curve1, curve2, curve3, GridVisible = TRUE):
```



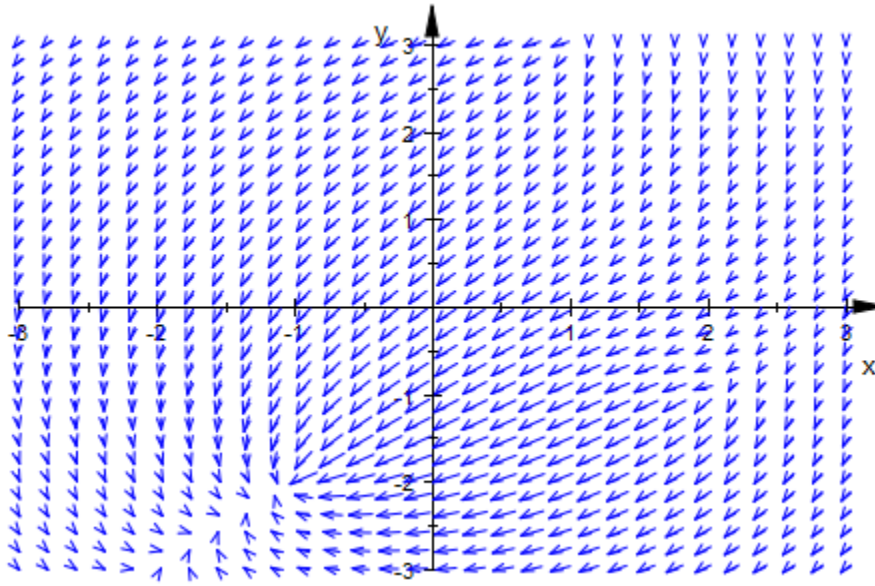

```
delete field, curve1, curve2, curve3:
```

Example 2

Assume you want to plot an electrostatic potential field. The following routine generates the necessary formula in a format accepted by `plot::VectorField2d`:

```
potentialE :=
  proc(l)
    local p, x0, y0, f0, fx, fy, dist;
  begin
    fx := 0; fy := 0;
    for p in [args()] do
```

```
      [x0, y0, f0] := p;
      dist := sqrt((x-x0)^2 + (y-y0)^2);
      fx := fx + f0*(x-x0)/dist;
      fy := fy + f0*(y-y0)/dist;
    end_for;
    [fx, fy];
  end_proc:

plot(plot::VectorField2d(potentialE([-1, -2, -1 ],
                                    [ 1,  3, 0.5],
                                    [ 2, -1, 0.5]),
                         x = -3..3, y = -3..3,
                         XMesh = 30, YMesh = 30)):
```

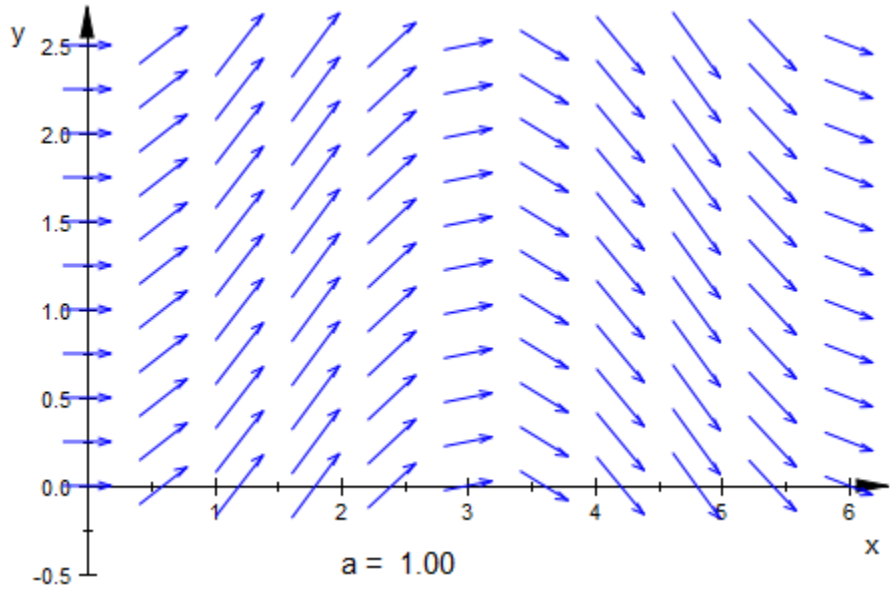


Example 3

Like most other objects, `plot::VectorField2d` can be animated by supplying an extra parameter:

```
field := plot::VectorField2d([1, a*sin(x) + (a-1)*cos(y)],
                             x = 0..6, y = 0..2.5, a=-1..1):
text := plot::Text2d(a -> "a = ".stringlib::formatf(a, 2, 5), [2, -0.5],
```

```
                    a = -1..1, HorizontalAlignment = Left):
plot(field, text)
```




```
delete field, text:
```

Parameters

v_1, v_2

The *x*- and *y*-component of the vector field: arithmetical expressions in `x`, `y`, and, possibly, the animation parameter `a`.

v_1, v_2 are equivalent to the attributes `XFunction`, `YFunction`.

x, y

Identifiers.

x, y are equivalent to the attributes `XName`, `YName`.

x_{min} .. x_{max}, y_{min} .. y_{max}

Real numerical values.

x_{min} .. x_{max}, y_{min} .. y_{max} are equivalent to the attributes `XRange`, `YRange`, `XMin`, `XMax`, `YMin`, `YMax`.

a

Animation parameter, specified as a = a_{min}..a_{max}, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Ode2d` | `plot::Ode3d` | `plot::Streamlines2d` | `plot::VectorField3d`

plot::VectorField3d

3D vector field

Syntax

```
plot::VectorField3d([v1, v2, v3], x = xmin .. xmax, y = ymin .. ymax, z = zmin .. zmax, <a = a
plot::VectorField3d(v1, v2, v3, x = xmin .. xmax, y = ymin .. ymax, z = zmin .. zmax, <a = amin
```

Description

`plot::VectorField3d([v_1, v_2, v_3], x = `x_{min}`..`x_{max}`, y = `y_{min}`..`y_{max}`, z = `z_{min}`..`z_{max}`)` represents a plot of the vector field defined by

$$(x, y, z) \to \left(v_1(x, y, z), v_2(x, y, z), v_3(x, y, z)\right)$$

with $(x, y, z) \in [x_min, x_max] \times [y_min, y_max] \times [z_min, z_max]$.

A vector field is defined by a function $f: \mathbb{R}^3 \to \mathbb{R}^3$. `plot::VectorField3d` displays a vector field by placing arrows at regular intervals with the arrow at (x, y, z) pointing in the direction $f(x, y, z)$.

The length of the arrows depend on $|f(x, y, z)|$ and the setting of the attribute `ArrowLength`: By default, arrow lengths are proportional to the magnitude of f, but can be set to be of fixed length or to scale logarithmically.

The density of arrows placed can be controlled with the attributes `XMesh`, `YMesh`, `ZMesh`, and `Mesh`. See the examples below.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`

Attribute	Purpose	Default Value
ArrowLength	scaling of arrows in a vector field	Proportional
Color	the main color	RGB::Blue
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Blue
LineWidth	width of lines	0.1
LineColor2	color of lines	RGB::DeepPink
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	0
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Mesh	number of sample points	[7, 7, 7]
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	

Attribute	Purpose	Default Value
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	TRUE
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	
VisibleFromTo	object visible during this time range	

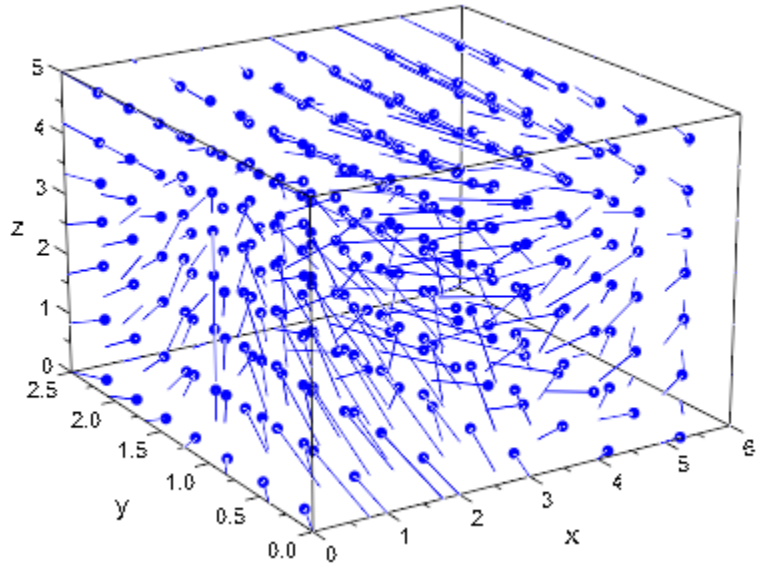
Attribute	Purpose	Default Value
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE
XFunction	function for x values	
XMax	final value of parameter “x”	
XMesh	number of sample points for parameter “x”	7
XMin	initial value of parameter “x”	
XName	name of parameter “x”	
XRange	range of parameter “x”	
YFunction	function for y values	
YMax	final value of parameter “y”	
YMesh	number of sample points for parameter “y”	7
YMin	initial value of parameter “y”	
YName	name of parameter “y”	
YRange	range of parameter “y”	
ZFunction	function for z values	
ZMax	final value of parameter “z”	
ZMesh	number of sample points for parameter “z”	7
ZMin	initial value of parameter “z”	
ZName	name of parameter “z”	
ZRange	range of parameter “z”	

Examples

Example 1

We demonstrate a plot of the vector field $v(x, y, z) = (1, sin(x) + cos(y), sin(z))$:

```
field := plot::VectorField3d([1, sin(x) + cos(y), sin(z)],
                             x = 0..6, y = 0..2.5, z = 0..5,
                             Mesh = [7, 7, 7]):
plot(field):
```



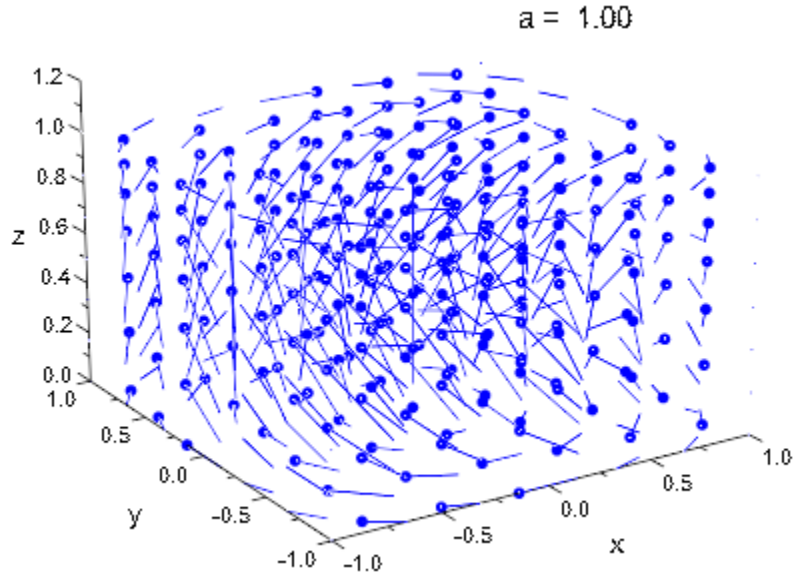

```
delete field:
```

Example 2

Like most other objects, `plot::VectorField3d` can be animated by supplying an extra parameter:

```
mycolor := (x, y, z, vx, vy, vz, a) -> [a, a*z, 1 - a]:
```

```
field := plot::VectorField3d([ a*y + (1-a)*x ,
                               -a*x + (1-a)*y,
                                a*sin(PI*z)],
                              x = -1..1, y = -1..1, z = 0..1,
                              LineColorFunction = mycolor,
                              Mesh = [7, 7, 7], a = 0..1):
text := plot::Text3d(a -> "a = ".stringlib::formatf(a, 2, 5),
                     [1, 0.7, 1.2], a = 0..1):
plot(field, text, Axes = Frame)
```




```
delete field, text:
```

Parameters

v_1, v_2, v_3

The x-, y-, and z-component of the vector field: arithmetical expressions in `x`, `y`, `z` and, possibly, the animation parameter `a`.

v_1, v_2, v_3 are equivalent to the attributes `XFunction`, `YFunction`, `ZFunction`.

x, y, z

Identifiers.

x, y, z are equivalent to the attributes `XName`, `YName`, `ZName`.

x_{min} .. x_{max}, y_{min} .. y_{max}, z_{min} .. z_{max}

Real numerical values.

x_{min} .. x_{max}, y_{min} .. y_{max}, z_{min} .. z_{max} are equivalent to the attributes `XRange`, `YRange`, `ZRange`, `XMin`, `XMax`, `YMin`, `YMax`, `ZMin`, `ZMax`.

a

Animation parameter, specified as a = $a_{min}..a_{max}$, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions

`plot` | `plot::copy`

MuPAD Graphical Primitives

`plot::Ode2d` | `plot::Ode3d` | `plot::VectorField2d`

plot::Waterman

Waterman polyhedra

Syntax

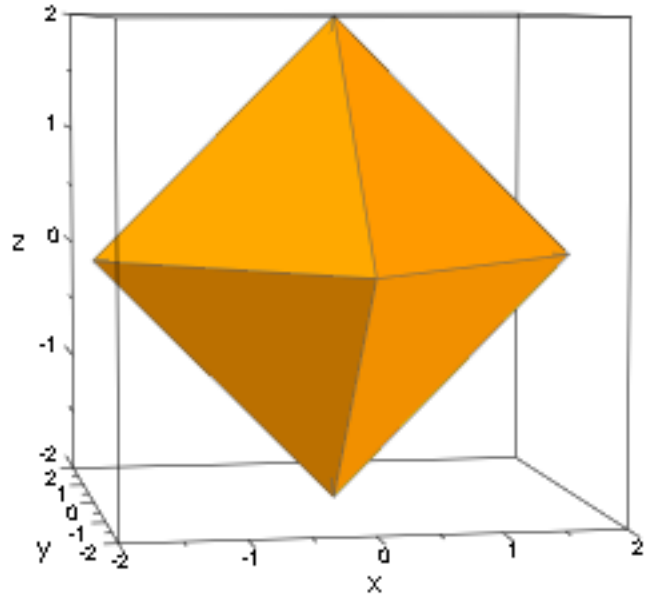
```
plot::Waterman(r, <a = a_min .. a_max>, options)
```

Description

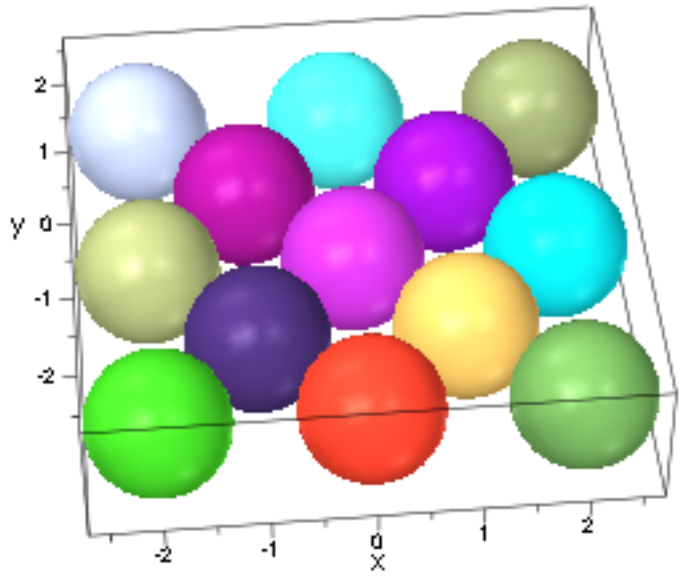
`plot::Waterman(r)` creates the Waterman polyhedron of radius *r*.

Waterman polyhedra, invented around 1990 by Steve Waterman, form a vast family of polyhedra. Some of them have a number of nice properties like multiple symmetries, or very interesting and regular shapes. Some other are just a bunch of faces formed out of irregular convex polygons.

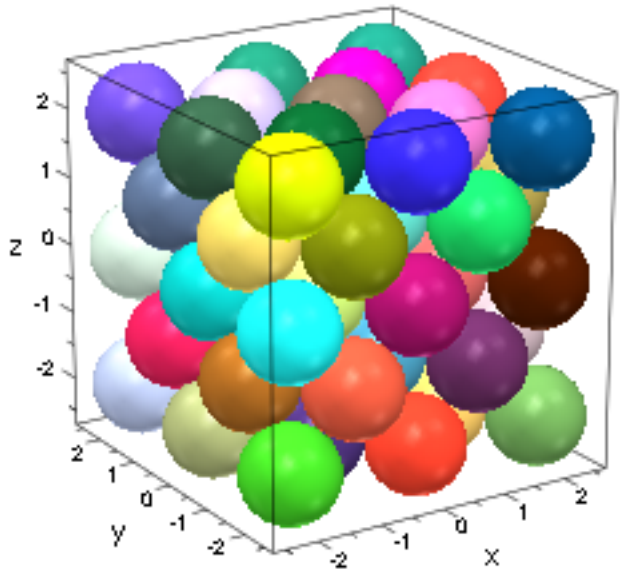
Waterman polyhedra result from the examination of balls in face-centered cubic close packing (which is one of the two densest packings of equally sized balls in 3D space, according to the Kepler Conjecture, proofed by Hales and Ferguson, 1997-2005). A single layer of spheres (of radius $\frac{1}{\sqrt{2}}$) in this packing looks like this:



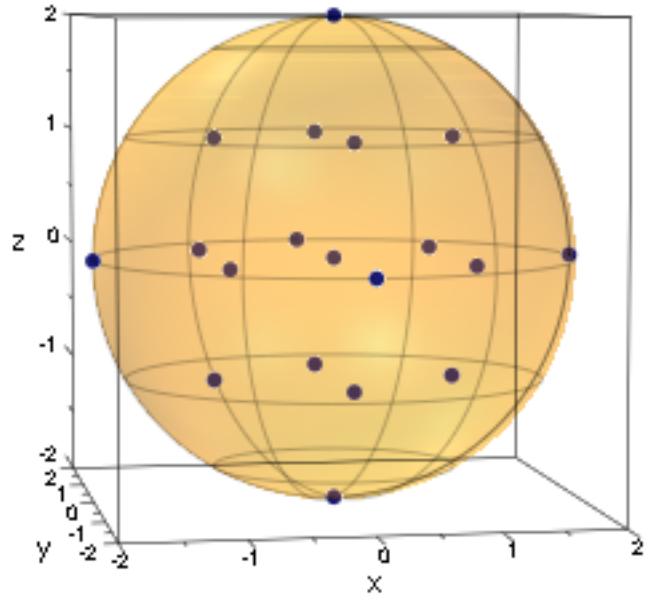
The close packing results from placing several of these layers over one another, shifted to optimally fill the gaps (in very much the same way your grocery store puts apples and oranges on display):

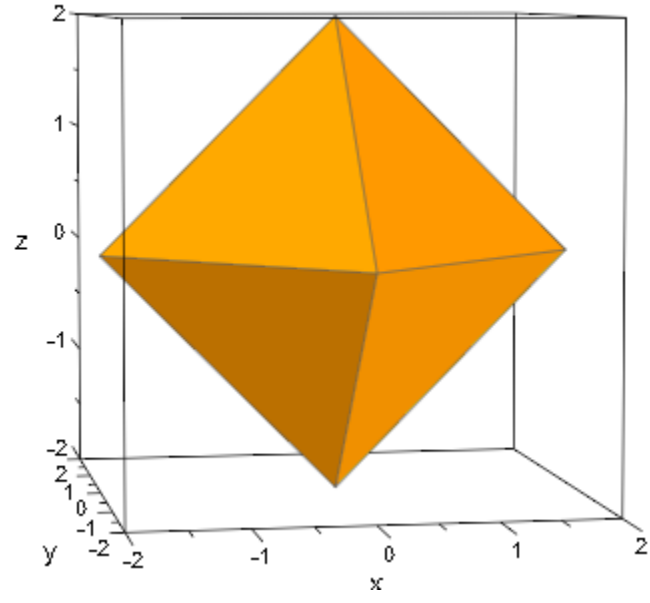


Given a radius *r* and a center *c* (which we let default to [0, 0, 0]), now consider all those centers of spheres in this packing which fall into the sphere of radius *r* around *c*:



The convex hull of these points is the Waterman polyhedron of the given radius and center:


2
1
0
-1
-2
z
2
1
0
-1
-2
y
-2
-1
0
1
2
X


2
1
0
-1
-2
z
2
1
0
-1
-2
y
-2
-1
0
1
2
x

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Center`	center of objects, rotation center	[0, 0, 0]
`CenterX`	center of objects, rotation center, x-component	0
`CenterY`	center of objects, rotation center, y-component	0
`CenterZ`	center of objects, rotation center, z-component	0
`Color`	the main color	`RGB::SafetyOrange`
`Filled`	filled or transparent areas and surfaces	`TRUE`
`FillColor`	color of areas and surfaces	`RGB::SafetyOrange`
`FillColor2`	second color of areas and surfaces for color blends	`RGB::CornflowerBlue`
`FillColorType`	surface filling types	`Flat`
`FillColorFunction`	functional area/surface coloring	
`FillColorDirection`	the direction of color transitions on surfaces	[0, 0, 1]
`FillColorDirectionX`	x-component of the direction of color transitions on surfaces	0
`FillColorDirectionY`	y-component of the direction of color transitions on surfaces	0
`FillColorDirectionZ`	z-component of the direction of color transitions on surfaces	1

Attribute	Purpose	Default Value
Frames	the number of frames in an animation	50
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	FALSE
LineColor	color of lines	RGB::Grey40.[0.4]
LineWidth	width of lines	0.25
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LinesVisible	visibility of lines	TRUE
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 1, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	1
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	

Attribute	Purpose	Default Value
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE
Radius	radius of circles, spheres etc.	
Shading	smooth color blend of surfaces	Smooth
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
Visible	visibility	TRUE
VisibleAfter	object visible after this time value	
VisibleBefore	object visible until this time value	

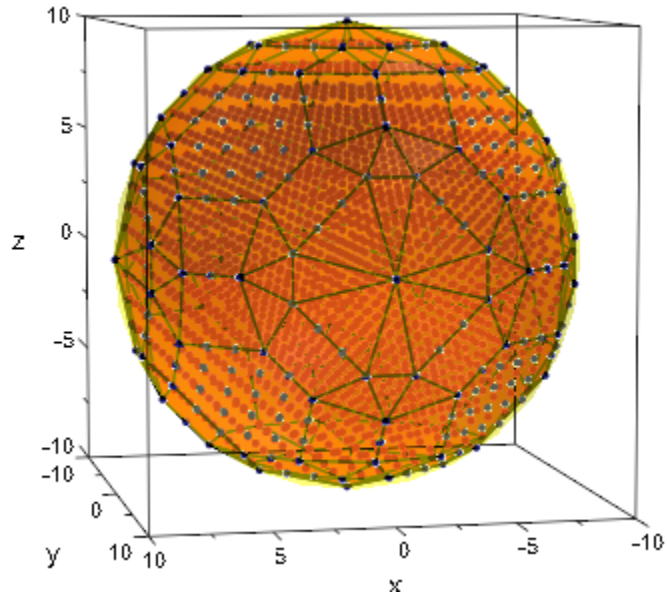
Attribute	Purpose	Default Value
VisibleFromTo	object visible during this time range	
VisibleAfterEnd	object visible after its animation time ended?	TRUE
VisibleBeforeBegin	object visible before its animation time starts?	TRUE

Examples

Example 1

With increasing radius, Waterman polyhedra get ever closer to spheres:

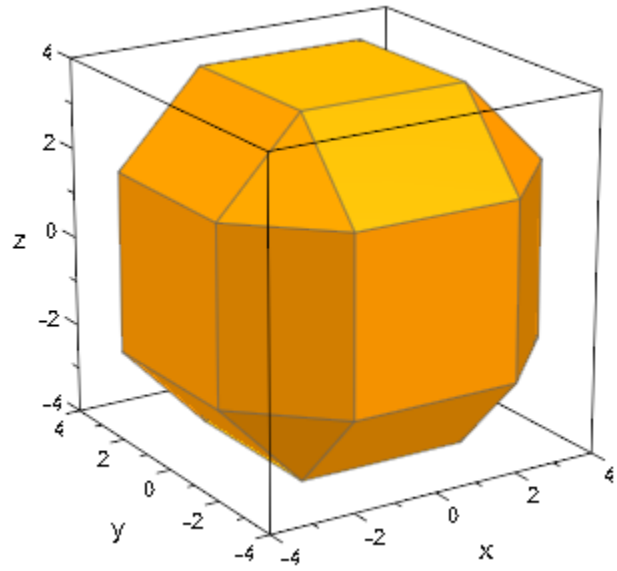
```
plot(plot::Waterman(r, r=0..10,
                    PointsVisible, PointSize=1,
                    LineColor=RGB::Black,
                    Color=RGB::Red.[0.75]),
     plot::Sphere(r, [0,0,0], r=0..10,
                  Color=RGB::Yellow.[0.3]),
     CameraDirection=[2,10,1])
```



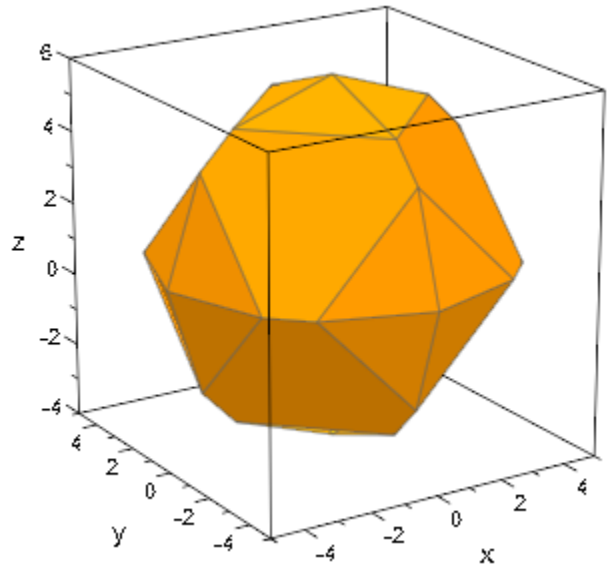
Example 2

Waterman polyhedra have a rather general definition and can be made from spheres centered anywhere:

```
plot(plot::Waterman(5, Center=[0,0,0]))
```

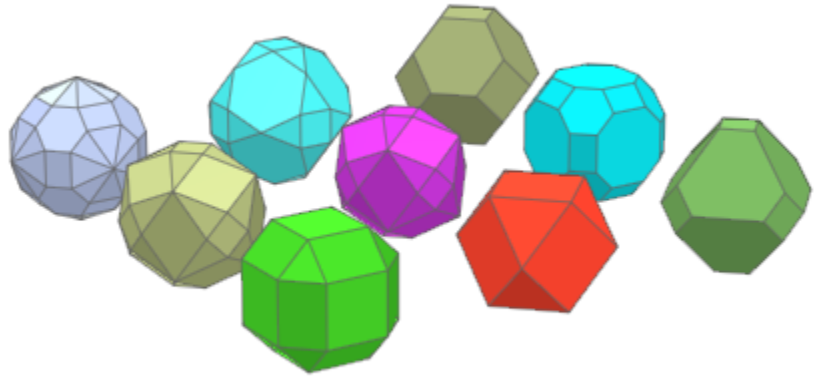



```
plot(plot::Waterman(5, Center=[0,0,1]))
```



To translate or scale a Waterman polyhedron, use `plot::Translate3d` and `plot::Scale3d`:

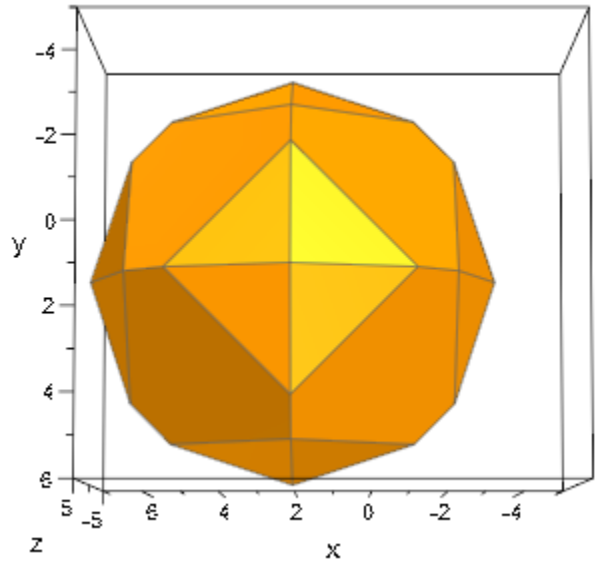
```
n := 3:
r := i -> 3/2+sqrt(i+1):
plot(plot::Translate3d([i mod n, i div n, 0],
               plot::Scale3d([1/(3*r(i)) $ 3],
                  plot::Waterman(r(i), Color=RGB::random())))
     $ i = 0..n^2-1, Axes=None)
```



Example 3

As usual, many attributes can be animated, although by the nature of Waterman polyhedra, the resulting animation will not be smooth:

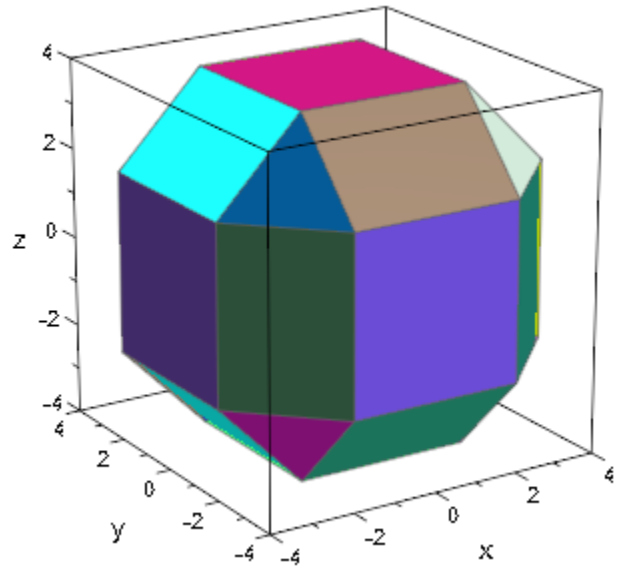
```
plot(plot::Waterman(5, Center=[a/PI, cos(a), 0],
                    a=0..2*PI),
     AnimationStyle=BackAndForth, CameraDirection=[0,0.1,1])
```



Example 4

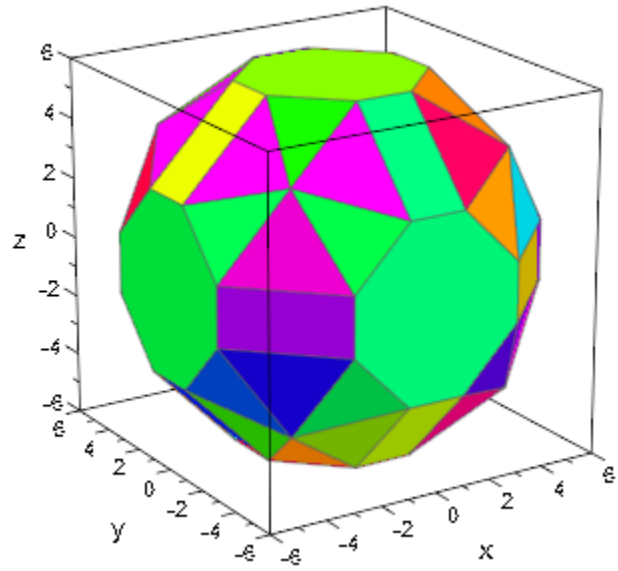
The `LineColorFunction` and `FillColorFunction` attributes can be set to functions which get indices of the currently painted surfacepolygon and its current vertex as fourth and fifth argument, repectively. This allows to color the polygons individually:

```
colors := [RGB::random() $ i = 1..42]:
plot(plot::Waterman(5, FillColorFunction=((x,y,z,i) -> colors[i])))
```



Another way of getting random colors which remain constant for each polygon is to use a procedure with `option remember`:

```
col :=
proc(n)
  option remember;
begin
  RGB::fromHSV([360*frandom(), 1, 1]);
end:
plot(plot::Waterman(7, FillColorFunction=((x,y,z,i) -> col(i))))
```



Parameters

r

An arithmetical expression: the radius of the polyhedron (see below for details).

r is equivalent to the attribute `Radius`.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

Algorithms

`plot::Waterman` uses `plot::hull` (and therefore, the Geometry Centre's qhull code) to compute the convex hull of the coordinates. Most of the remaining code has been contributed by Mirek Majewski.

See Also

MuPAD Functions
`plot`

MuPAD Graphical Primitives
`plot::Dodecahedron` | `plot::Hexahedron` | `plot::Icosahedron` | `plot::Octahedron` | `plot::Sphere` | `plot::Tetrahedron`

plot::XRotate

Surfaces of revolution around x-axis

Syntax

```
plot::XRotate(f, x = x_min .. x_max, <a = a_min .. a_max>, options)
```

Description

`plot::XRotate(f, x = `x_{min}`..xmax)` creates a surface of revolution by rotating the function graph $y = f(x)$ with $x \in [x_min, x_max]$ around the x-axis.

`plot::XRotate` rotates the graph of the function $y = f(x)$ around the x-axis, creating a surface of revolution. The slice of the surface parallel to the y-z plane at a point x is a circle of radius $f(x)$.

The range of the rotation can be restricted with the attributes `AngleBegin`, `AngleEnd`, `AngleRange`. The surface of revolution will only span over the given range of the rotation angle.

Surfaces of revolution are parametrized surfaces. The first surface parameter is x, the second is the rotation angle. Surfaces of revolution react to most of the graphical attributes that surfaces of type `plot::Surface` react to. For example, use `Mesh`, `Submesh` to control the numerical mesh or use `ULinesVisible`, `VLinesVisible` to switch the parameter lines on/off.

Attributes

Attribute	Purpose	Default Value
`AdaptiveMesh`	adaptive sampling	`0`
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AngleEnd`	end of angle range	`2*PI`
`AngleBegin`	begin of angle range	`0`
`AngleRange`	angle range	`0 .. 2*PI`

Attribute	Purpose	Default Value
`Color`	the main color	`RGB::Red`
`Filled`	filled or transparent areas and surfaces	`TRUE`
`FillColor`	color of areas and surfaces	`RGB::Red`
`FillColor2`	second color of areas and surfaces for color blends	`RGB::CornflowerBlue`
`FillColorType`	surface filling types	`Dichromatic`
`FillColorFunction`	functional area/surface coloring	
`FillColorDirection`	the direction of color transitions on surfaces	`[0, 0, 1]`
`FillColorDirectionX`	x-component of the direction of color transitions on surfaces	`0`
`FillColorDirectionY`	y-component of the direction of color transitions on surfaces	`0`
`FillColorDirectionZ`	z-component of the direction of color transitions on surfaces	`1`
`Frames`	the number of frames in an animation	`50`
`Function`	function expression or procedure	
`Legend`	makes a legend entry	
`LegendText`	short explanatory text for legend	
`LegendEntry`	add this object to the legend?	`TRUE`
`LineColor`	color of lines	`RGB::Black.[0.25]`
`LineWidth`	width of lines	`0.35`

Attribute	Purpose	Default Value
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	0
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Mesh	number of sample points	[25, 25]
MeshVisible	visibility of irregular mesh lines in 3D	FALSE
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE

Attribute	Purpose	Default Value
Shading	smooth color blend of surfaces	Smooth
Submesh	density of submesh (additional sample points)	[0, 0]
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
ULinesVisible	visibility of parameter lines (u lines)	TRUE
UMesh	number of sample points for parameter “u”	25
USubmesh	density of additional sample points for parameter “u”	0
VLinesVisible	visibility of parameter lines (v lines)	TRUE
VMesh	number of sample points for parameter “v”	25
VSubmesh	density of additional sample points for parameter “v”	0

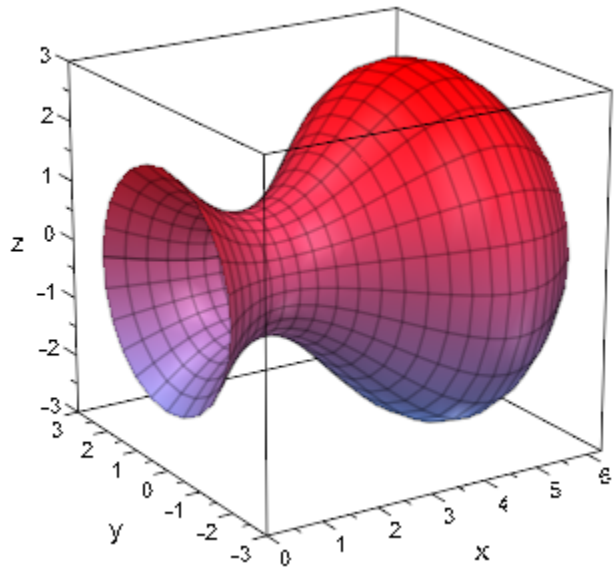
Attribute	Purpose	Default Value
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`
`XContours`	contour lines at constant x values	[]
`XMax`	final value of parameter “x”	`5`
`XMin`	initial value of parameter “x”	`-5`
`XName`	name of parameter “x”	
`XRange`	range of parameter “x”	`-5 .. 5`
`YContours`	contour lines at constant y values	[]
`ZContours`	contour lines at constant z values	[]

Examples

Example 1

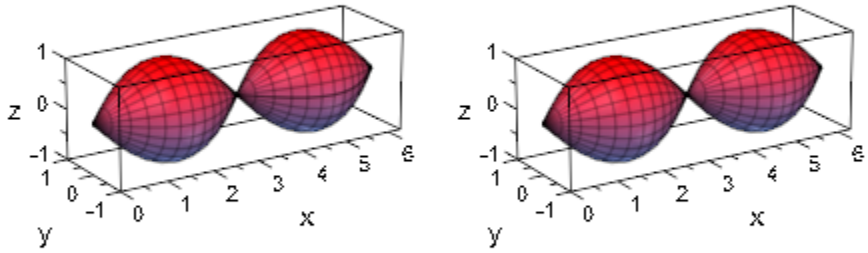
By default, `plot::XRotate` displays a complete revolution, just as if an object was created on a lathe:

```
plot(plot::XRotate(2 - sin(x), x = 0..2*PI))
```



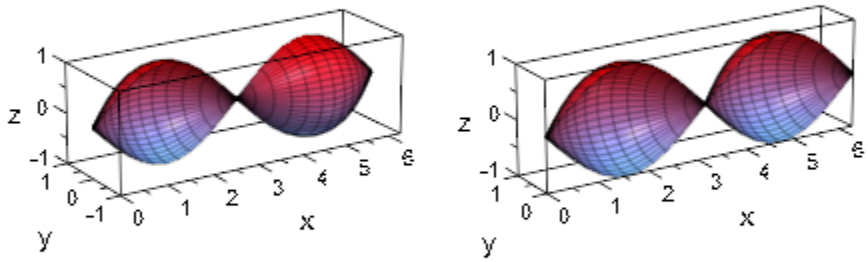
This rotation is insensitive to negative values: The surfaces of revolution of $f(x)$ and $|f(x)|$ are identical:

```
plot(plot::Scene3d(plot::XRotate(sin(x), x = 0..2*PI)),
     plot::Scene3d(plot::XRotate(abs(sin(x)), x = 0..2*PI)),
     Layout = Horizontal)
```



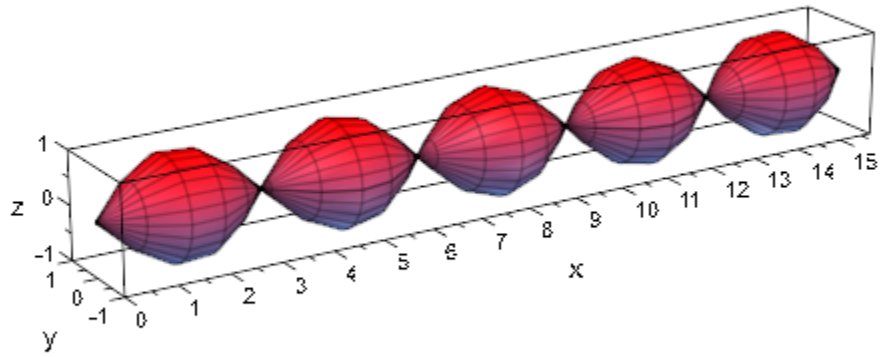
This symmetry is broken when not performing a whole revolution:

```
plot(plot::Scene3d(plot::XRotate(sin(x), x = 0..2*PI,
                                 AngleRange = -PI/2..PI/2)),
     plot::Scene3d(plot::XRotate(abs(sin(x)), x = 0..2*PI,
                                 AngleRange = -PI/2..PI/2)),
     Layout = Horizontal)
```



`plot::XRotate` can be animated, like almost every plot object:

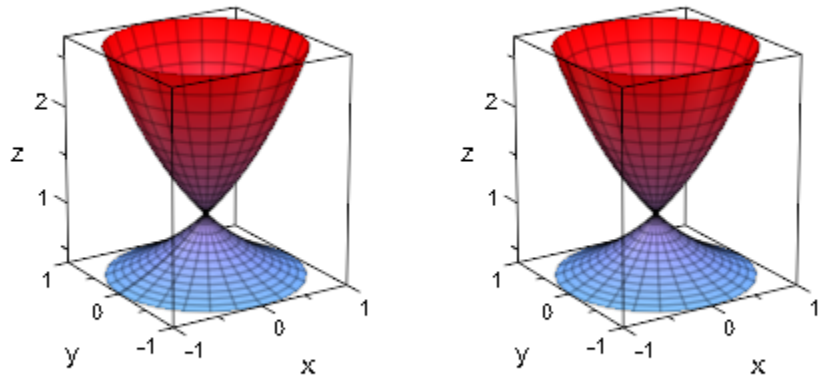
```
plot(plot::XRotate(sin(x + a), x = 0 .. 2*a + PI,
                   AngleRange = 0 .. PI + a/2,
                   a = 0..2*PI))
```



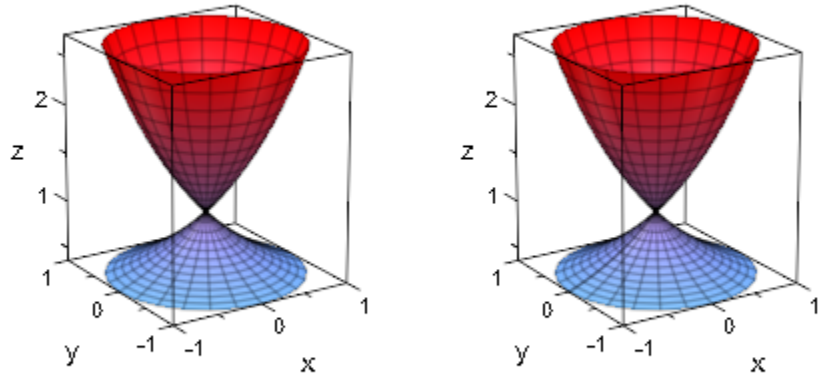
Example 2

As for `plot::ZRotate`, most of the points from above hold here, too. Obviously, the symmetry for a whole revolution is now with respect to the *x* values, not the function values:

```
plot(plot::Scene3d(plot::ZRotate(exp(x), x = -1..1)),
     plot::Scene3d(plot::ZRotate(exp(-x), x = -1..1)),
     Layout = Horizontal)
```




```
plot(plot::Scene3d(plot::ZRotate(exp(x), x = -1..1,
                                 AngleRange = -a..a,
                                 a = 0..PI)),
     plot::Scene3d(plot::ZRotate(exp(-x), x = -1..1,
                                 AngleRange = -a..a,
                                 a = 0..PI)),
     Layout = Horizontal)
```



Parameters

`f`

The function: an arithmetical expression or a `piecewise` object in the independent variable x and the animation parameter a. Alternatively, a procedure that accepts 1 input parameter x or 2 input parameters x, a and returns a real numerical value when the input parameters are numerical.

f is equivalent to the attribute `Function`.

`x`

The independent variable: an identifier or an indexed identifier.

x is equivalent to the attribute `XName`.

x_{min} .. x_{max}

The plot range: x_{min}, x_{max} must be numerical real values or expressions of the animation parameter a.

x_{min} .. x_{max} is equivalent to the attributes `XRange`, `XMin`, `XMax`.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Function2d` | `plot::Function3d` | `plot::Surface` | `plot::ZRotate`

plot::ZRotate

Surfaces of revolution around z-axis

Syntax

```
plot::ZRotate(f, x = x_min .. x_max, <a = a_min .. a_max>, options)
```

Description

`plot::ZRotate(f, x = `x_{min}`..xmax)` creates a surface of revolution by rotating the function graph $z = f(x)$ with $x \in [x_min, x_max]$ around the z-axis.

`plot::ZRotate` rotates the graph of the given function $z = f(x)$ around the z-axis, creating another surface of revolution. The slice of the surface parallel to the x-y plane at a point z consists of circles with radii $|x_i|$ given by the solutions of $f(x) = z$.

The range of the rotation can be restricted with the attributes `AngleBegin`, `AngleEnd`, `AngleRange`. The surface of revolution will only span over the given range of the rotation angle.

Surfaces of revolution are parametrized surfaces. The first surface parameter is x, the second is the rotation angle. Surfaces of revolution react to most of the graphical attributes that surfaces of type `plot::Surface` react to. For example, use `Mesh`, `Submesh` to control the numerical mesh or use `ULinesVisible`, `VLinesVisible` to switch the parameter lines on/off.

Attributes

Attribute	Purpose	Default Value
`AdaptiveMesh`	adaptive sampling	`0`
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`AngleEnd`	end of angle range	`2*PI`
`AngleBegin`	begin of angle range	`0`

Attribute	Purpose	Default Value
AngleRange	angle range	0 .. 2*PI
Color	the main color	RGB::Red
Filled	filled or transparent areas and surfaces	TRUE
FillColor	color of areas and surfaces	RGB::Red
FillColor2	second color of areas and surfaces for color blends	RGB::CornflowerBlue
FillColorType	surface filling types	Dichromatic
FillColorFunction	functional area/surface coloring	
FillColorDirection	the direction of color transitions on surfaces	[0, 0, 1]
FillColorDirectionX	x-component of the direction of color transitions on surfaces	0
FillColorDirectionY	y-component of the direction of color transitions on surfaces	0
FillColorDirectionZ	z-component of the direction of color transitions on surfaces	1
Frames	the number of frames in an animation	50
Function	function expression or procedure	
Legend	makes a legend entry	
LegendText	short explanatory text for legend	
LegendEntry	add this object to the legend?	TRUE
LineColor	color of lines	RGB::Black.[0.25]

Attribute	Purpose	Default Value
LineWidth	width of lines	0.35
LineColor2	color of lines	RGB::DeepPink
LineStyle	solid, dashed or dotted lines?	Solid
LineColorType	line coloring types	Flat
LineColorFunction	functional line coloring	
LineColorDirection	the direction of color transitions on lines	[0, 0, 1]
LineColorDirectionX	x-component of the direction of color transitions on lines	0
LineColorDirectionY	y-component of the direction of color transitions on lines	0
LineColorDirectionZ	z-component of the direction of color transitions on lines	1
Mesh	number of sample points	[25, 25]
MeshVisible	visibility of irregular mesh lines in 3D	FALSE
Name	the name of a plot object (for browser and legend)	
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
PointSize	the size of points	1.5
PointStyle	the presentation style of points	FilledCircles
PointsVisible	visibility of mesh points	FALSE

Attribute	Purpose	Default Value
Shading	smooth color blend of surfaces	Smooth
Submesh	density of submesh (additional sample points)	[0, 0]
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Title	object title	
TitleFont	font of object titles	[" sans-serif ", 11]
TitlePosition	position of object titles	
TitleAlignment	horizontal alignment of titles w.r.t. their coordinates	Center
TitlePositionX	position of object titles, x component	
TitlePositionY	position of object titles, y component	
TitlePositionZ	position of object titles, z component	
ULinesVisible	visibility of parameter lines (u lines)	TRUE
UMesh	number of sample points for parameter “u”	25
USubmesh	density of additional sample points for parameter “u”	0
VLinesVisible	visibility of parameter lines (v lines)	TRUE
VMesh	number of sample points for parameter “v”	25
VSubmesh	density of additional sample points for parameter “v”	0

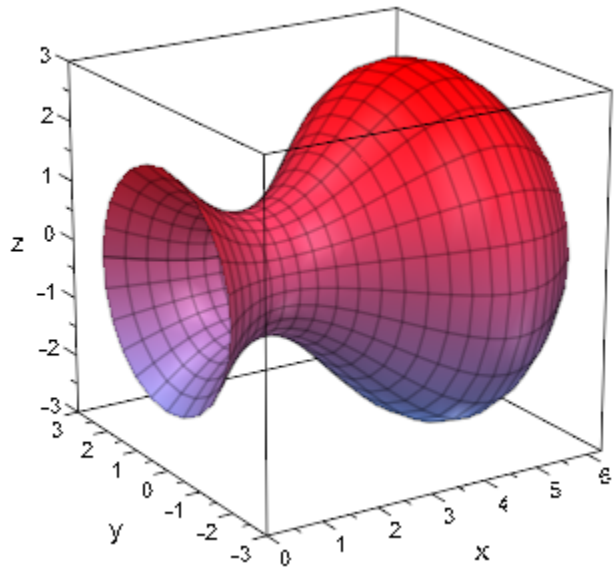
Attribute	Purpose	Default Value
`Visible`	visibility	`TRUE`
`VisibleAfter`	object visible after this time value	
`VisibleBefore`	object visible until this time value	
`VisibleFromTo`	object visible during this time range	
`VisibleAfterEnd`	object visible after its animation time ended?	`TRUE`
`VisibleBeforeBegin`	object visible before its animation time starts?	`TRUE`
`XContours`	contour lines at constant x values	[]
`XMax`	final value of parameter “x”	5
`XMin`	initial value of parameter “x”	0
`XName`	name of parameter “x”	
`XRange`	range of parameter “x”	0 .. 5
`YContours`	contour lines at constant y values	[]
`ZContours`	contour lines at constant z values	[]

Examples

Example 1

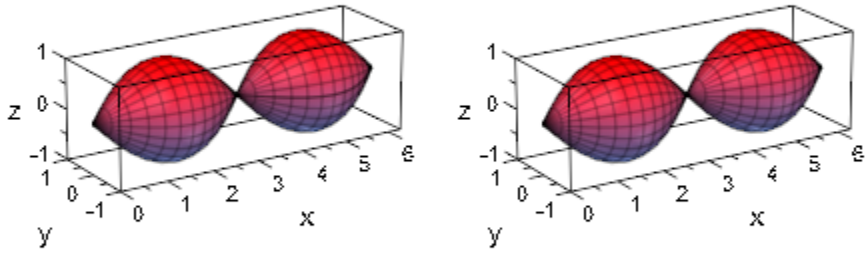
By default, `plot::XRotate` displays a complete revolution, just as if an object was created on a lathe:

```
plot(plot::XRotate(2 - sin(x), x = 0..2*PI))
```



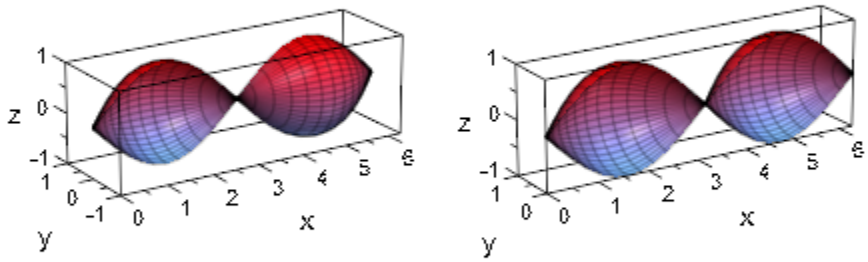
This rotation is insensitive to negative values: The surfaces of revolution of $f(x)$ and $|f(x)|$ are identical:

```
plot(plot::Scene3d(plot::XRotate(sin(x), x = 0..2*PI)),
     plot::Scene3d(plot::XRotate(abs(sin(x)), x = 0..2*PI)),
     Layout = Horizontal)
```



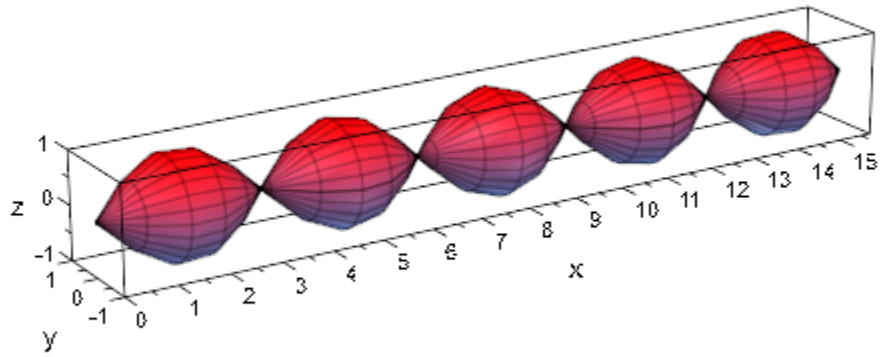
This symmetry is broken when not performing a whole revolution:

```
plot(plot::Scene3d(plot::XRotate(sin(x), x = 0..2*PI,
                                 AngleRange = -PI/2..PI/2)),
     plot::Scene3d(plot::XRotate(abs(sin(x)), x = 0..2*PI,
                                 AngleRange = -PI/2..PI/2)),
     Layout = Horizontal)
```



`plot::XRotate` can be animated, like almost every plot object:

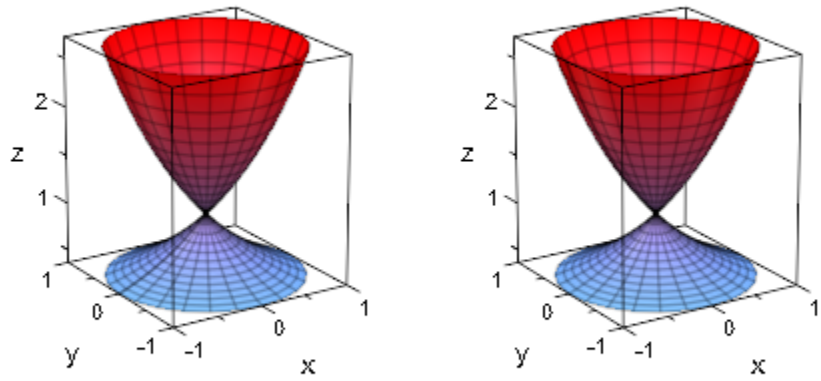
```
plot(plot::XRotate(sin(x + a), x = 0 .. 2*a + PI,
                   AngleRange = 0 .. PI + a/2,
                   a = 0..2*PI))
```



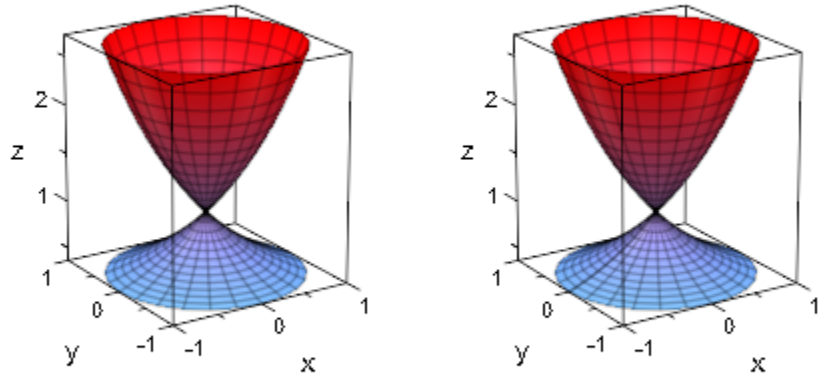
Example 2

As for `plot::ZRotate`, most of the points from above hold here, too. Obviously, the symmetry for a whole revolution is now with respect to the *x* values, not the function values:

```
plot(plot::Scene3d(plot::ZRotate(exp(x), x = -1..1)),
     plot::Scene3d(plot::ZRotate(exp(-x), x = -1..1)),
     Layout = Horizontal)
```




```
plot(plot::Scene3d(plot::ZRotate(exp(x), x = -1..1,
                                 AngleRange = -a..a,
                                 a = 0..PI)),
     plot::Scene3d(plot::ZRotate(exp(-x), x = -1..1,
                                 AngleRange = -a..a,
                                 a = 0..PI)),
     Layout = Horizontal)
```



Parameters

`f`

The function: an arithmetical expression or a `piecewise` object in the independent variable x and the animation parameter a. Alternatively, a procedure that accepts 1 input parameter x or 2 input parameters x, a and returns a real numerical value when the input parameters are numerical.

f is equivalent to the attribute `Function`.

`x`

The independent variable: an identifier or an indexed identifier.

x is equivalent to the attribute `XName`.

x_{min} .. x_{max}

The plot range: x_{min}, x_{max} must be numerical real values or expressions of the animation parameter a.

x_{min} .. x_{max} is equivalent to the attributes `XRange`, `XMin`, `XMax`.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Function2d` | `plot::Function3d` | `plot::Surface` | `plot::XRotate`

plot::Canvas

Drawing area

Syntax

```
plot::Canvas(object1, object2, …, <a = amin .. amax>, options)
```

Description

`plot::Canvas` is the top level element of the hierachy of graphical objects. It represents the 2 dimensional drawing area into which 2D and 3D plots are painted.

The user does not need to create a canvas object explicitly, because a `plot` command such as `plot(object1, object2, …)` implicitly creates a default canvas object to display the graphical objects in.

Strictly speaking, a canvas object is a container for scenes of type `plot::Scene2d` or `plot::Scene3d`, respectively. The user, however, does not have to bother about this technicality, because a suitable default scene is created internally, when graphical primitives are passed to `plot::Canvas`.

A canvas can display several scenes simultaneously. However, all scenes must be of the same dimension. A mixture of 2D and 3D is not supported!

See the help page of the canvas attribute `Layout` for details on the layout of a canvas containing several scenes.

The canvas object is always visible in the interactive object browser of the MuPAD graphics tool (see section Viewer, Browser, and Inspector: Interactive Manipulation of this document). It can contain one or more scenes as its children. When the canvas object is selected, it provides access to a variety of attributes that are associated with the canves. The canvas attributes allow to

- set `Height` and `Width` of the plot,
- set a `Header` and/or a `Footer`,

- control the layout (`Layout`, `Rows`, `Columns`),
- set various style parameters such as `BorderWidth`, `BorderColor`, `BackgroundColor` etc.

A complete listing of the attributes associated with a canvas is given below. Follow the links to the help pages of the attributes to find more detailed information.

Apart from these attributes of the canvas object, also attributes for scenes, coordinate systems and graphical objects inside the canvas can be specified when generating a canvas object. These attribute values are inherited to the objects inside the canvas as new default values.

Attributes

Attribute	Purpose	Default Value
`AnimationStyle`	behaviour of the animation toolbar	`RunOnce`
`AutoPlay`	start animations automatically	`TRUE`
`BackgroundColor`	background color	`RGB::White`
`BorderColor`	color of frame/border around canvas and scenes	`RGB::Grey50`
`BorderWidth`	width of frame/border around canvas and scenes	`0`
`BottomMargin`	bottom margin width	`1`
`Columns`	number of columns of scenes	`0`
`Footer`	footer text	
`FooterFont`	font of footers (scene and canvas)	`[" sans-serif ", 12]`
`FooterAlignment`	alignment of footer of canvas and scenes	`Center`
`Header`	header text	
`HeaderFont`	font of headers (scene and canvas)	`[" sans-serif ", 12]`

Attribute	Purpose	Default Value
HeaderAlignment	alignment of header of canvas and scenes	Center
Height	heights of canvas/scenes	80
InitialTime	initial time of the animation slider	
Layout	arrangement/layout of several scenes in a canvas	Tabular
LeftMargin	left margin width	1
Margin	margins around canvas and scenes	1
Name	the name of a plot object (for browser and legend)	
OutputUnits	the physical length unit used by the inspector	unit::mm
RightMargin	right margin width	1
Rows	number of rows of scenes	0
Spacing	space between scenes	1.0
TopMargin	top margin width	1
Width	widths of canvas/scenes	120

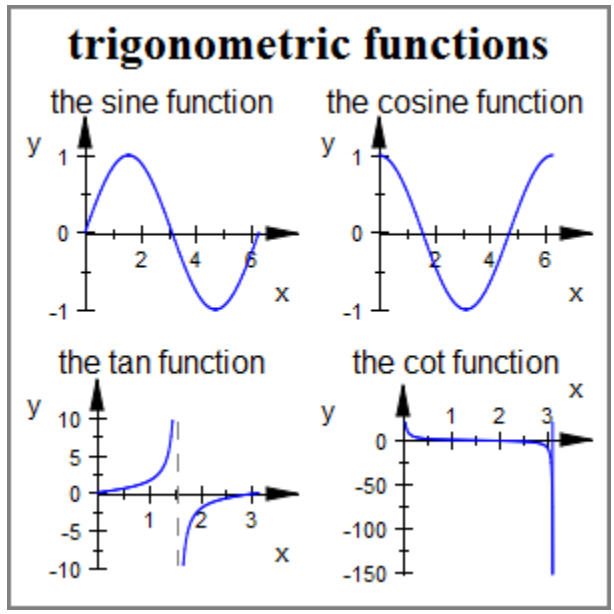
Examples

Example 1

We diplay several scenes in a canvas. Various canvas attributes are passed when creating the canvas object:

```
S1 := plot::Scene2d(plot::Function2d(sin(x), x = 0..2*PI),
                    Header = "the sine function"):
S2 := plot::Scene2d(plot::Function2d(cos(x), x = 0..2*PI),
                    Header = "the cosine function"):
S3 := plot::Scene2d(plot::Function2d(tan(x), x = 0..PI),
                    Header = "the tan function"):
```

```
S4 := plot::Scene2d(plot::Function2d(cot(x), x = 0..PI),
                    Header = "the cot function"):
C := plot::Canvas(S1, S2, S3, S4,
                  Width = 80*unit::mm, Height = 80*unit::mm,
                  BorderWidth = 0.5*unit::mm,
                  Header = "trigonometric functions",
                  HeaderFont = ["Times New Roman", Bold, 18]):
plot(C)
```




```
delete S1, S2, S3, S4, C:
```

Parameters

object$_1$, object$_2$, …

Graphical objects

See Also

MuPAD Functions

plot | plot::copy

MuPAD Graphical Primitives

plot::CoordinateSystem2d | plot::CoordinateSystem3d | plot::Scene2d | plot::Scene3d

plot::CoordinateSystem2d

Coordinate system to display 2D objects in

Syntax

`plot::CoordinateSystem2d(`$object_1$`, `$object_2$`, …, <a = `a_{min}` .. `a_{max}`>, options)`

Description

`plot::CoordinateSystem2d` is a container to display graphical 2D objects within. Usually, the user does not need to create such an object explicitly, because a `plot` command such as `plot(`$object_1$`, `$object_2$`, …)` creates a default object of type `plot::CoordinateSystem2d` implicitly to display the graphical objects in.

The `plot::CoordinateSystem2d` object is always visible in the interactive object browser of the MuPAD graphics tool (see Viewer, Browser, and Inspector: Interactive Manipulation of this document). It contains the graphical objects as its children. When the coordinate system object is selected, it provides access to a variety of attributes that are associated with the coordinate system. These attributes allow to manipulate:

- the `CoordinateType` (linear vs. logarithmic coordinates),
- the `ViewingBox` (visibility range),
- the coordinate axes (axes titles, visibility, alignment, type, tips etc.),
- the ticks along the coodinate axes (number of tick marks, visibility, tick labels etc.),
- the coordinate grid (visibility, color, line width etc.),
- the scaling ratios of the coordinate directions (`Constrained` vs. `UnConstrained`).

A complete listing of the attributes associated with the coordinate system is given below. Follow the links to the help pages of the attributes to find more detailed information.

Apart from these attributes of the coordinate system, also attributes for the graphical objects inside the coordinate system can be specified when generating an object of type `plot::CoordinateSystem2d`. These attribute values are inherited to the graphical objects as new default values.

A graphical scene may contain more than one coordinate system. Each coordinate system provides separate coordinate axes, ticks, grid lines etc.

In such a case, separate `plot::CoordinateSystem2d` containers must be created explicitly by the user and passed to a `plot` command (or inserted into a scene of type `plot::Scene2d`). Cf. "Example 2" on page 24-1142.

Attributes

Attribute	Purpose	Default Value
`Axes`	type of the coordinate axes	`Automatic`
`AxesTips`	arrow tips at the coordinate axes?	`TRUE`
`AxesOrigin`	crosspoint of the coordinate axes	`[0, 0]`
`AxesTitles`	titles for the coordinate axes	`[" x ", " y "]`
`AxesInFront`	coordinate axes in front of or behind graphical objects?	`FALSE`
`AxesOriginX`	crosspoint of the coordinate axes, x-coordinate	`0`
`AxesOriginY`	crosspoint of the coordinate axes, y-coordinate	`0`
`AxesVisible`	display coordinate axes?	`TRUE`
`AxesLineColor`	color of the coordinate axes	`RGB::Black`
`AxesLineWidth`	width of the coordinate axes	`0.18`
`AxesTitleFont`	font of axes titles	`[" sans-serif ", 10]`
`AxesTitleAlignment`	alignment of axes titles	`End`
`CoordinateType`	linear versus logarithmic plots in 2D	`LinLin`
`GridInFront`	coordinate grid in front of or behind graphical objects?	`FALSE`
`GridVisible`	display a coordinate grid?	`FALSE`

Attribute	Purpose	Default Value
GridLineColor	line color of the coordinate grid	RGB::Grey60
GridLineWidth	width of coordinate grid lines	0.1
GridLineStyle	line style of the coordinate grid	Solid
Name	the name of a plot object (for browser and legend)	
Scaling	scaling ratios	Unconstrained
SubgridVisible	display a coordinate subgrid?	FALSE
SubgridLineColor	line color of the coordinate subgrid	RGB::Grey80
SubgridLineWidth	width of coordinate subgrid lines	0.1
SubgridLineStyle	line style of the coordinate subgrid	Solid
TicksAt	special axes tick marks	
TicksAnchor	user defined start of axes tick marks	0
TicksLength	length of axes tick marks	2
TicksNumber	number of axes tick marks	Normal
TicksBetween	number of minor (unlabeled) axes tick marks between major (labeled) axes tick marks	1
TicksVisible	display axes tick marks?	TRUE
TicksDistance	user defined axes tick mark distance	0
TicksLabelFont	font of tick labels	[" sans-serif ", 8]
TicksLabelStyle	display style of axes tick labels	Horizontal

Attribute	Purpose	Default Value
TicksLabelsVisible	display axes tick labels?	TRUE
ViewingBox	the visible coordinate range	[Automatic .. Automatic, Automatic .. Automatic]
ViewingBoxXMin	the smallest visible x-values	Automatic
ViewingBoxYMin	the smallest visible y-values	Automatic
ViewingBoxXMax	the largest visible x-values	Automatic
ViewingBoxYMax	the largest visible y-values	Automatic
ViewingBoxXRange	the range of x-values visible	Automatic .. Automatic
ViewingBoxYRange	the range of y-values visible	Automatic .. Automatic
XAxisTitle	title for the x axis	" x "
XAxisVisible	display x axis?	TRUE
XAxisTitleAlignment	alignment of x axis title	End
XGridVisible	display a coordinate grid in x-direction?	FALSE
XSubgridVisible	display a coordinate subgrid in x-direction?	FALSE
XTicksAt	special x axis tick marks	
XTicksAnchor	user defined start of x axis tick marks	0
XTicksNumber	number of x axis tick marks	Normal
XTicksBetween	number of minor (unlabeled) x axis tick marks between major (labeled) x axis tick marks	1
XTicksVisible	display x axis tick marks?	TRUE
XTicksDistance	distance of tick marks on x axis	0
XTicksLabelStyle	display style of x axis tick labels	Horizontal
XTicksLabelsVisible	display x axis tick labels?	TRUE

Attribute	Purpose	Default Value
YAxisTitle	title for the y axis	" y "
YAxisVisible	display y axis?	TRUE
YAxisTitleAlignment	alignment of y axis title	End
YAxisTitleOrientation	orientation of the vertical axis title in 2D	Horizontal
YGridVisible	display a coordinate grid in y-direction?	FALSE
YSubgridVisible	display a coordinate subgrid in y-direction?	FALSE
YTicksAt	special y axis tick marks	
YTicksAnchor	user defined start of y axis tick marks	0
YTicksNumber	number of y axis tick marks	Normal
YTicksBetween	number of minor (unlabeled) y axis tick marks between major (labeled) y axis tick marks	1
YTicksVisible	display y axis tick marks?	TRUE
YTicksDistance	distance of tick marks on y axis	0
YTicksLabelStyle	display style of y axis tick labels	Horizontal
YTicksLabelsVisible	display y axis tick labels?	TRUE

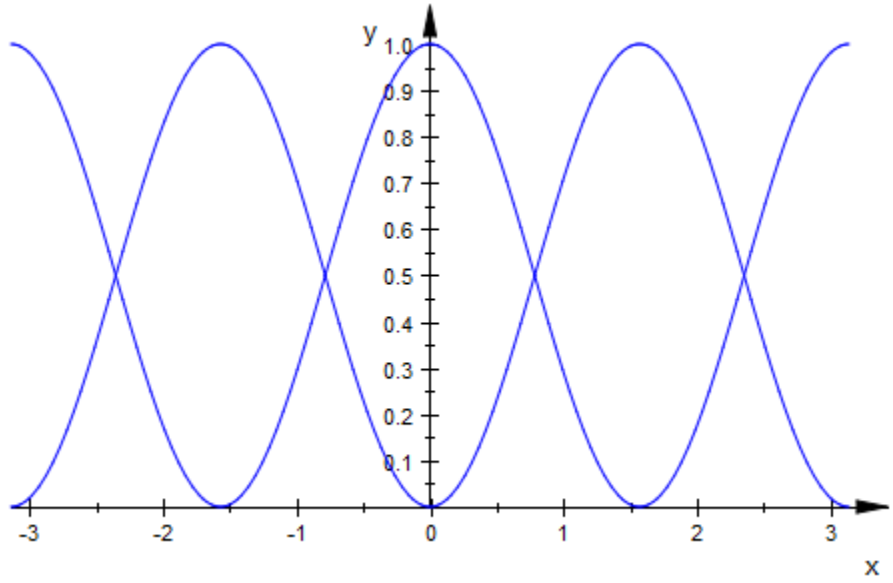
Examples

Example 1

When executing a plot command, a default `plot::CoordinateSystem2d` is created implicitly which contains the specified graphical objects:

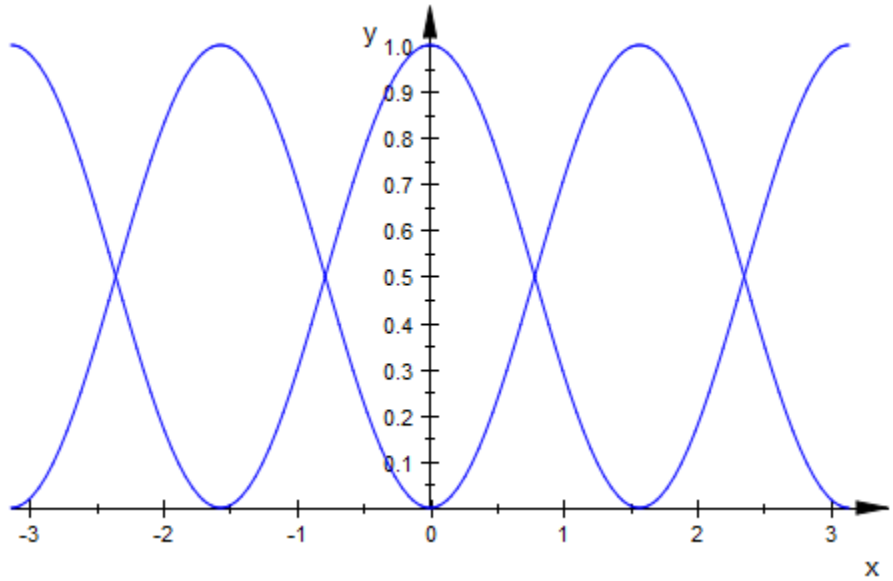
```
f := plot::Function2d(sin(x)^2, x = -PI..PI):
```

```
g := plot::Function2d(cos(x)^2, x = -PI..PI):
plot(f, g)
```



We can also create the coordinate system explicitly. The result is the same:

```
plot(plot::CoordinateSystem2d(f, g))
```



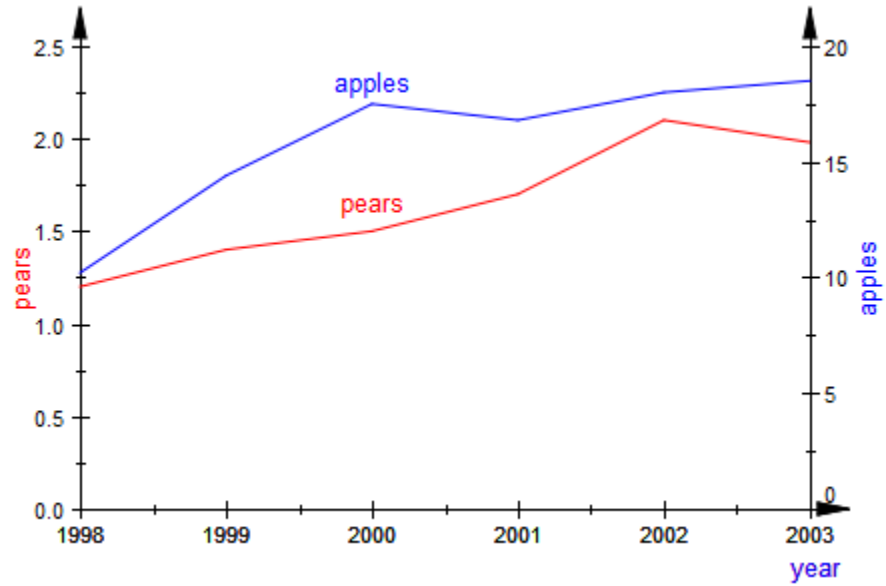

```
delete f, g:
```

Example 2

We present the yearly sales of pears and apples in one scene. Different coordinate systems are used to obtain separate axes. We set various attributes to determine the positioning of the axes and their titles:

```
pears := plot::Polygon2d(
           [[1998, 1.2], [1999, 1.4], [2000, 1.5],
            [2001, 1.7], [2002, 2.1], [2003, 1.98]],
           Color = RGB::Red, Title = "pears",
           TitlePosition = [2000, 1.6],
           TitleFont = [RGB::Red]):
apples := plot::Polygon2d(
            [[1998, 10.2], [1999, 14.4], [2000, 17.5],
             [2001, 16.8], [2002, 18.0], [2003, 18.5]],
            Color = RGB::Blue, Title = "apples",
            TitlePosition = [2000, 18.0],
            TitleFont = [RGB::Blue]):

CS1 := plot::CoordinateSystem2d(pears):
```

```
CS1::AxesOriginX := 1998:
CS1::ViewingBox := [1998..2003, 0..2.5]:
CS1::AxesTitleFont := [RGB::Red]:
CS1::XAxisTitle := "year":
CS1::YAxisTitle := "pears":

CS2 := plot::CoordinateSystem2d(apples):
CS2::AxesOriginX := 2003:
CS2::ViewingBox := [1998..2003, 0..20]:
CS2::AxesTitleFont := [RGB::Blue]:
CS2::XAxisTitle := "year":
CS2::YAxisTitle := "apples":
plot(CS1, CS2, Axes = Origin, YAxisTitleAlignment = Center,
     YAxisTitleOrientation = Vertical)
```




```
delete pears, apples, CS1, CS2:
```

Parameters

object$_1$, object$_2$, …

Graphical 2D objects

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Canvas` | `plot::CoordinateSystem2d` | `plot::Scene2d`

plot::CoordinateSystem3d

Coordinate system to display 3D objects in

Syntax

`plot::CoordinateSystem3d(object`$_1$`, object`$_2$`, …, <a = a`$_{min}$ `.. a`$_{max}$`>, options)`

Description

`plot::CoordinateSystem3d` is a container to display graphical 3D objects within. Usually, the user does not need to create such an object explicitly, because a `plot` command such as `plot(object`$_1$`, object`$_2$`, …)` creates a default object of type `plot::CoordinateSystem3d` implicitly to display the graphical objects in.

The `plot::CoordinateSystem3d` object is always visible in the interactive object browser of the MuPAD graphics tool (see section Viewer, Browser, and Inspector: Interactive Manipulation of this document). It contains the graphical objects as its children. When the coordinate system object is selected, it provides access to a variety of attributes that are associated with the coordinate system. These attributes allow to manipulate:

- the `CoordinateType` (linear vs. logarithmic coordinates),
- the `ViewingBox` (visibility range),
- the coordinate axes (axes titles, visibility, alignment, type, tips etc.),
- the ticks along the coodinate axes (number of tick marks, visibility, tick labels etc.),
- the coordinate grid (visibility, color, line width etc.),
- the scaling ratios of the coordinate directions (`Constrained` vs. `UnConstrained`).

A complete listing of the attributes associated with the coordinate system is given below. Follow the links to the help pages of the attributes to find more detailed information.

Apart from these attributes of the coordinate system, also attributes for the graphical objects inside the coordinate system can be specified when generating an object of type `plot::CoordinateSystem3d`. These attribute values are inherited to the graphical objects as new default values.

A graphical scene may contain more than one coordinate system. Each coordinate system provides separate coordinate axes, ticks, grid lines etc.

In such a case, separate `plot::CoordinateSystem3d` containers must be created explicitly by the user and passed to a `plot` command (or inserted into a scene of type `plot::Scene3d`). Cf. "Example 2" on page 24-1152.

Attributes

Attribute	Purpose	Default Value
`Axes`	type of the coordinate axes	`Boxed`
`AxesTips`	arrow tips at the coordinate axes?	`FALSE`
`AxesOrigin`	crosspoint of the coordinate axes	`[0, 0, 0]`
`AxesTitles`	titles for the coordinate axes	`[" x ", " y ", " z "]`
`AxesOriginX`	crosspoint of the coordinate axes, x-coordinate	`0`
`AxesOriginY`	crosspoint of the coordinate axes, y-coordinate	`0`
`AxesOriginZ`	crosspoint of the coordinate axes, z-coordinate	`0`
`AxesVisible`	display coordinate axes?	`TRUE`
`AxesLineColor`	color of the coordinate axes	`RGB::Black`
`AxesLineWidth`	width of the coordinate axes	`0.18`
`AxesTitleFont`	font of axes titles	`[" sans-serif ", 10]`
`AxesTitleAlignment`	alignment of axes titles	`Center`
`GridVisible`	display a coordinate grid?	`FALSE`
`GridLineColor`	line color of the coordinate grid	`RGB::Grey60`
`GridLineWidth`	width of coordinate grid lines	`0.1`

Attribute	Purpose	Default Value
GridLineStyle	line style of the coordinate grid	Solid
Name	the name of a plot object (for browser and legend)	
Scaling	scaling ratios	Unconstrained
SubgridVisible	display a coordinate subgrid?	FALSE
SubgridLineColor	line color of the coordinate subgrid	RGB::Grey80
SubgridLineWidth	width of coordinate subgrid lines	0.1
SubgridLineStyle	line style of the coordinate subgrid	Solid
TicksAt	special axes tick marks	
TicksAnchor	user defined start of axes tick marks	0
TicksLength	length of axes tick marks	2
TicksNumber	number of axes tick marks	Normal
TicksBetween	number of minor (unlabeled) axes tick marks between major (labeled) axes tick marks	1
TicksVisible	display axes tick marks?	TRUE
TicksDistance	user defined axes tick mark distance	0
TicksLabelFont	font of tick labels	[" sans-serif ", 8]
TicksLabelStyle	display style of axes tick labels	Horizontal
TicksLabelsVisible	display axes tick labels?	TRUE

Attribute	Purpose	Default Value
ViewingBox	the visible coordinate range	[Automatic .. Automatic, Automatic .. Automatic, Automatic .. Automatic]
ViewingBoxXMin	the smallest visible x-values	Automatic
ViewingBoxYMin	the smallest visible y-values	Automatic
ViewingBoxXMax	the largest visible x-values	Automatic
ViewingBoxZMin	the smallest visible z-values	Automatic
ViewingBoxYMax	the largest visible y-values	Automatic
ViewingBoxZMax	the largest visible z-values	Automatic
ViewingBoxXRange	the range of x-values visible	Automatic .. Automatic
ViewingBoxYRange	the range of y-values visible	Automatic .. Automatic
ViewingBoxZRange	the range of z-values visible	Automatic .. Automatic
XAxisTitle	title for the x axis	" x "
XAxisVisible	display x axis?	TRUE
XAxisTitleAlignment	alignment of x axis title	Center
XGridVisible	display a coordinate grid in x-direction?	FALSE
XSubgridVisible	display a coordinate subgrid in x-direction?	FALSE
XTicksAt	special x axis tick marks	
XTicksAnchor	user defined start of x axis tick marks	0
XTicksNumber	number of x axis tick marks	Normal
XTicksBetween	number of minor (unlabeled) x axis tick marks between major (labeled) x axis tick marks	1
XTicksVisible	display x axis tick marks?	TRUE
XTicksDistance	distance of tick marks on x axis	0

Attribute	Purpose	Default Value
XTicksLabelStyle	display style of x axis tick labels	Horizontal
XTicksLabelsVisible	display x axis tick labels?	TRUE
YAxisTitle	title for the y axis	" y "
YAxisVisible	display y axis?	TRUE
YAxisTitleAlignment	alignment of y axis title	Center
YGridVisible	display a coordinate grid in y-direction?	FALSE
YSubgridVisible	display a coordinate subgrid in y-direction?	FALSE
YTicksAt	special y axis tick marks	
YTicksAnchor	user defined start of y axis tick marks	0
YTicksNumber	number of y axis tick marks	Normal
YTicksBetween	number of minor (unlabeled) y axis tick marks between major (labeled) y axis tick marks	1
YTicksVisible	display y axis tick marks?	TRUE
YTicksDistance	distance of tick marks on y axis	0
YTicksLabelStyle	display style of y axis tick labels	Horizontal
YTicksLabelsVisible	display y axis tick labels?	TRUE
ZAxisTitle	title for the z axis	" z "
ZAxisVisible	display z axis?	TRUE
ZAxisTitleAlignment	alignment of z axis title	Center
ZGridVisible	display a coordinate grid in z-direction?	FALSE
ZSubgridVisible	display a coordinate subgrid in z-direction?	FALSE

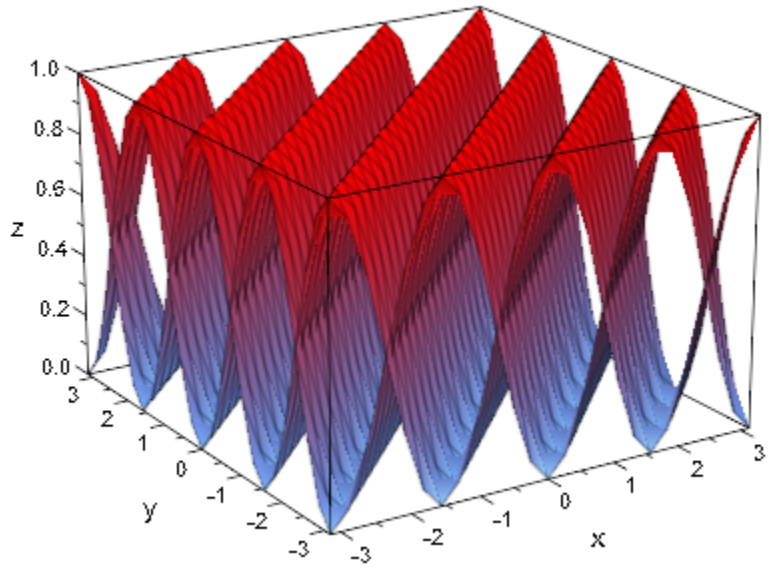
Attribute	Purpose	Default Value
ZTicksAt	special z axis tick marks	
ZTicksAnchor	user defined start of z axis tick marks	0
ZTicksNumber	number of z axis tick marks	Normal
ZTicksBetween	number of minor (unlabeled) z axis tick marks between major (labeled) z axis tick marks	1
ZTicksVisible	display z axis tick marks?	TRUE
ZTicksDistance	distance of tick marks on z axis	0
ZTicksLabelStyle	display style of z axis tick labels	Horizontal
ZTicksLabelsVisible	display z axis tick labels?	TRUE

Examples

Example 1

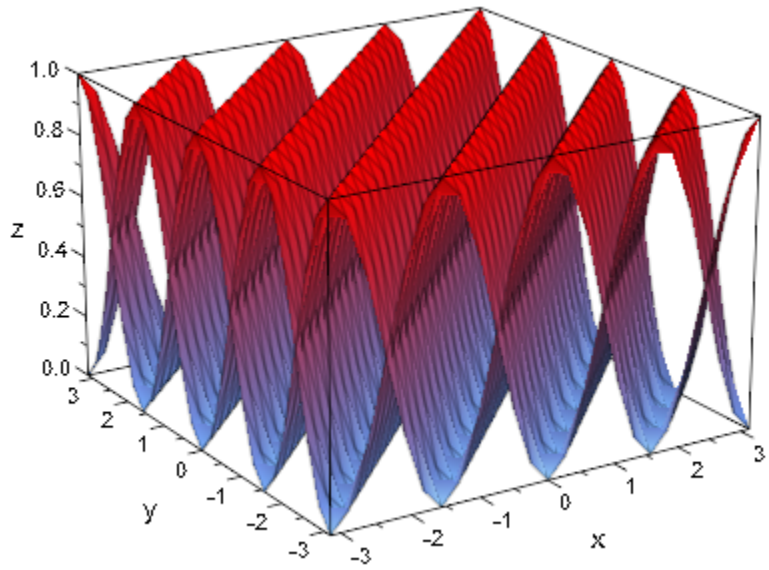
When executing a plot command, a default `plot::CoordinateSystem3d` is created implicitly which contains the specified graphical objects:

```
f := plot::Function3d(sin(x - y)^2, x = -PI..PI, y = -PI..PI):
g := plot::Function3d(cos(x - y)^2, x = -PI..PI, y = -PI..PI):
plot(f, g)
```



We can also create the coordinate system explicitly. The result is the same:

```
plot(plot::CoordinateSystem3d(f, g))
```




```
delete f, g:
```

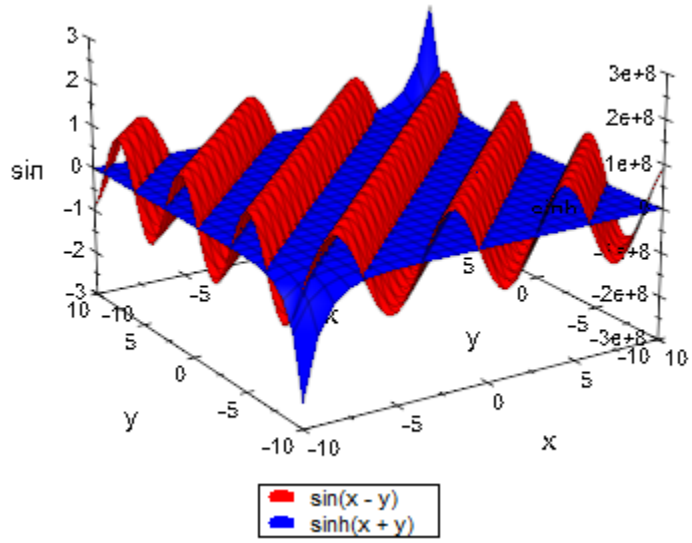
Example 2

The functions $f_1 = sin(x - y)$ and $f_2 = sin(h)(x + y)$ cannot be plotted simultaneously in one coordinate system over the range $x \in [-10, 10]$, $y \in [-10, 10]$, because they produce function values of different orders of magnitude. To plot them together, we use two different coordinate systems. We request explicit vertical ranges for the (rather different) viewing boxes by the attribute `ViewingBoxZRange`.

We set various attributes of the coordinate systems to determine the positioning of the axes and their titles:

```
f1 := plot::Function3d(sin(x - y), x = -10..10, y = -10..10,
                       Submesh = [2, 2],
                       Color = RGB::Red, FillColorType = Flat,
                       Legend = "sin(x - y)"):
CS1 := plot::CoordinateSystem3d(f1):
CS1::Axes := Origin:
CS1::AxesOrigin := [-10, 10, -3]:
CS1::ViewingBoxZRange := -3..3:
```

```
CS1::ZAxisTitle := "sin":

f2 := plot::Function3d(sinh(x + y), x = -10..10, y = -10..10,
                       Color = RGB::Blue, FillColorType = Flat,
                       Legend = "sinh(x + y)"):
CS2 := plot::CoordinateSystem3d(f2):
CS2::Axes := Origin:
CS2::AxesOrigin := [10, -10, -3*10^8]:
CS2::ViewingBoxZRange := -3*10^8..3*10^8:
CS2::ZAxisTitle := "sinh":

plot(CS1, CS2):
```




```
delete f1, CS1, f2, CS2:
```

Parameters

object$_1$, object$_2$, …

Graphical 3D objects

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Canvas` | `plot::CoordinateSystem3d` | `plot::Scene3d`

plot::Group2d

Groups of 2D objects

Syntax

`plot::Group2d(object2d`$_1$`, object2d`$_2$`, …, <a = a`$_{min}$ `.. a`$_{max}$`>, options)`

Description

`plot::Group2d` forms a group of any number of graphical 2D objects.

Grouping together a larger number of graphical objects and accessing the group as a whole simplifies their handling. In particular, the main purpose of a group is to inherit graphical attributes that are shared by all members of the group.

If you wish to change the inherited attributes interactively, you should not select the group itself in the interactive object browser of the MuPAD graphics tool (see section Viewer, Browser, and Inspector: Interactive Manipulation of this document). Underneath the group object, you find 'defaults' branches for the objects in the group. Select the 'defaults' branch for the object type that you wish to set attributes for.

A group may again contain groups.

Note: When working with groups of points, it is more efficient to use the specialized grouping constructs `plot::PointList2d` and `plot::PointList3d` instead of generic groups of points!

Attributes

Attribute	Purpose	Default Value
`Name`	the name of a plot object (for browser and legend)	

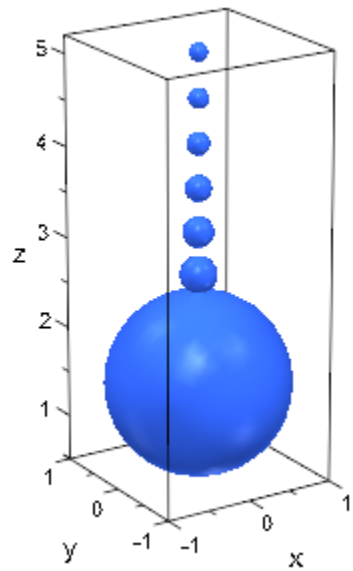
Attribute	Purpose	Default Value
`Visible`	visibility	TRUE

Examples

Example 1

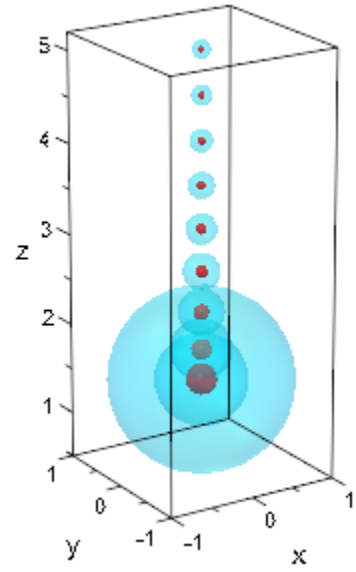
We plot two groups of bubbles. Some bubbles are not visible as they are inside larger bubbles:

```
G1 := plot::Group3d(plot::Sphere(1/n, [0, 0, n/2 + 1/n]
                                 ) $ n = 1..10):
G2 := plot::Group3d(plot::Sphere(1/(3*n), [0, 0, n/2 + 1/n]
                                 ) $ n = 2..10):
plot(G1, G2)
```



We wish to increase transparency of all bubbles in the first group, but keep the bubbles in the second group opaque. Since the bubbles are grouped, it is easy to set different attribute values for the two groups:

```
G1::Color := RGB::SkyBlue.[0.25]:
G2::Color := RGB::Red:
plot(G1, G2)
```



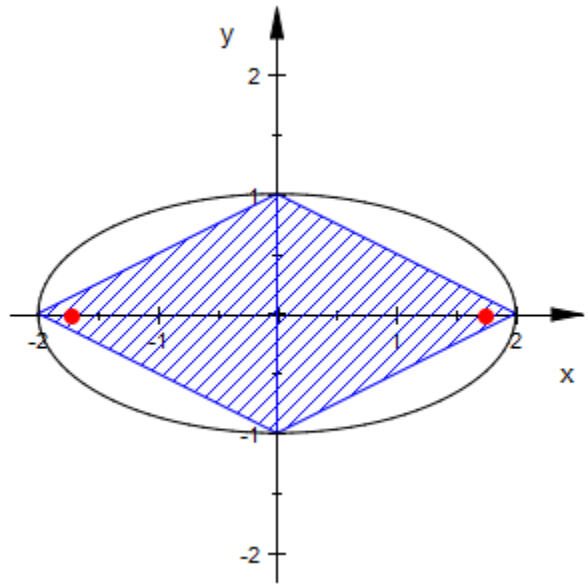

```
delete G1, G2:
```

Example 2

Consider a group `G1` consisting of two triangles inscribed in an ellipse. We form a new group `G2` consisting of the group `G1`, the ellipse and its focal points. The entire figure given by the group `G2` is rotated by an animated `plot::Rotate2d`:

```
G1 := plot::Group2d(
  plot::Polygon2d([[0, -1], [0, 1], [-2, 0]]),
  plot::Polygon2d([[0, -1], [0, 1], [ 2, 0]]),
  Closed = TRUE, Filled = TRUE, Color = RGB::Blue):
G2 := plot::Group2d(
  G1,
  plot::Ellipse2d(2, 1,[0, 0]),
  plot::PointList2d([[-sqrt(3), 0], [sqrt(3), 0]]),
  PointSize = 2*unit::mm,
```

```
  PointColor = RGB::Red,
  LineColor = RGB::Black):
plot(plot::Rotate2d(a, [0, 0], a = 0..2*PI, G2))
```




```
delete G1, G2:
```

Parameters

object2d$_1$, object2d$_2$, …

Graphical 2D objects

See Also

MuPAD Functions
plot | plot::copy

MuPAD Graphical Primitives
plot::Group3d | plot::PointList2d | plot::PointList3d

More About

- "Groups of Primitives"

plot::Group3d

Groups of 3D objects

Syntax

`plot::Group3d(object3d`$_1$`, object3d`$_2$`, …, <a = a`$_{min}$ `.. a`$_{max}$`>, options)`

Description

`plot::Group3d` forms a group of any number of graphical 3D objects.

Grouping together a larger number of graphical objects and accessing the group as a whole simplifies their handling. In particular, the main purpose of a group is to inherit graphical attributes that are shared by all members of the group.

If you wish to change the inherited attributes interactively, you should not select the group itself in the interactive object browser of the MuPAD graphics tool (see section Viewer, Browser, and Inspector: Interactive Manipulation of this document). Underneath the group object, you find 'defaults' branches for the objects in the group. Select the 'defaults' branch for the object type that you wish to set attributes for.

A group may again contain groups.

Note: When working with groups of points, it is more efficient to use the specialized grouping constructs `plot::PointList2d` and `plot::PointList3d` instead of generic groups of points!

Attributes

Attribute	Purpose	Default Value
`Name`	the name of a plot object (for browser and legend)	

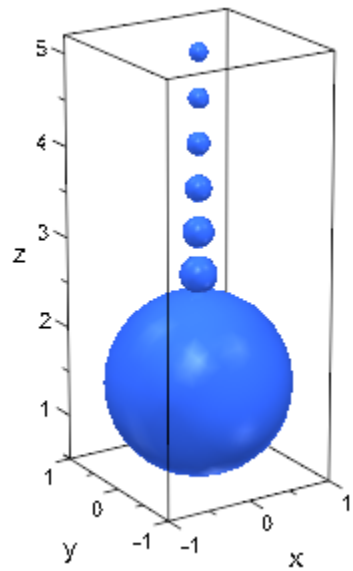
Attribute	Purpose	Default Value
`Visible`	visibility	TRUE

Examples

Example 1

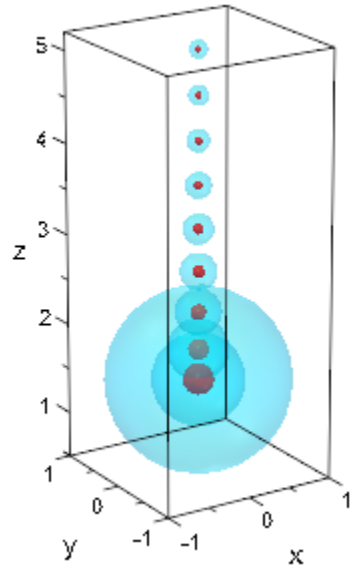
We plot two groups of bubbles. Some bubbles are not visible as they are inside larger bubbles:

```
G1 := plot::Group3d(plot::Sphere(1/n, [0, 0, n/2 + 1/n]
                                 ) $ n = 1..10):
G2 := plot::Group3d(plot::Sphere(1/(3*n), [0, 0, n/2 + 1/n]
                                 ) $ n = 2..10):
plot(G1, G2)
```



We wish to increase transparency of all bubbles in the first group, but keep the bubbles in the second group opaque. Since the bubbles are grouped, it is easy to set different attribute values for the two groups:

```
G1::Color := RGB::SkyBlue.[0.25]:
G2::Color := RGB::Red:
plot(G1, G2)
```



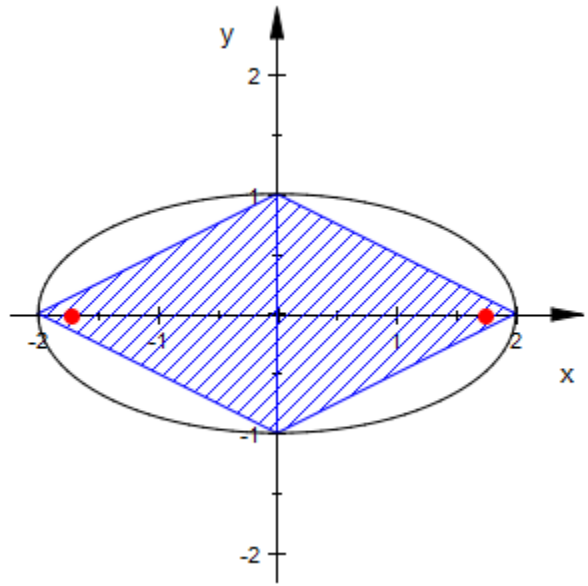

```
delete G1, G2:
```

Example 2

Consider a group `G1` consisting of two triangles inscribed in an ellipse. We form a new group `G2` consisting of the group `G1`, the ellipse and its focal points. The entire figure given by the group `G2` is rotated by an animated `plot::Rotate2d`:

```
G1 := plot::Group2d(
  plot::Polygon2d([[0, -1], [0, 1], [-2, 0]]),
  plot::Polygon2d([[0, -1], [0, 1], [ 2, 0]]),
  Closed = TRUE, Filled = TRUE, Color = RGB::Blue):
G2 := plot::Group2d(
  G1,
  plot::Ellipse2d(2, 1,[0, 0]),
  plot::PointList2d([[-sqrt(3), 0], [sqrt(3), 0]]),
  PointSize = 2*unit::mm,
```

```
  PointColor = RGB::Red,
  LineColor = RGB::Black):
plot(plot::Rotate2d(a, [0, 0], a = 0..2*PI, G2))
```




```
delete G1, G2:
```

Parameters

object3d$_1$, object3d$_2$, …

Graphical 3D objects

See Also

MuPAD Functions

plot | plot::copy

MuPAD Graphical Primitives

plot::Group2d | plot::PointList2d | plot::PointList3d

More About

- "Groups of Primitives"

plot::Scene2d

2D scenes

Syntax

`plot::Scene2d(object2d`$_1$`, object2d`$_2$`, …, <a = a`$_{min}$ `.. a`$_{max}$`>, options)`

Description

`plot::Scene2d` is a container to display one or more coordinate systems containing graphical objects. Scene objects must be created when several graphical scenes are to be displayed simultaneously in a plot.

Usually, you do not need to create a scene object explicitly, because a `plot` command such as `plot(object`$_1$`, object`$_2$`, …)` creates a default scene object implicitly to display the graphical objects in.

The need for creating scene objects explicitly arises only when several scenes are to be displayed simultaneously in one plot.

The MuPAD graphics makes a clear division between 2D and 3D. Scene objects of type `plot::Scene2d` do not accept 3D objects and `plot::Scene3d` objects do not accept 2D objects. When several scenes are displayed simultaneously in a single plot, all scenes must be of the same dimension.

Strictly speaking, a 2D scene object is a container for coordinate systems of type `plot::CoordinateSystem2d`. However, you do not have to bother about this technicality because a suitable default coordinate system is created internally, when graphical primitives are passed to `plot::Scene2d`.

Scene objects are always visible in the interactive object browser of the MuPAD graphics tool (see section Viewer, Browser, and Inspector: Interactive Manipulation of this document). Each scene contains one or more coordinate systems as its children. When the scene object is selected, it provides access to a variety of attributes that are associated with scenes. The scene attributes allow to

- set annotations (`Header`, `Footer`) and control the `Legend`,

- set layout parameters such as height and width (if the canvas attribute `Layout` is set to `Absolute` or `Relative`),
- set various style parameters such as `BackgroundColor` etc.

A complete listing of the attributes associated with a 2D scene is given below. Follow the links to the help pages of the attributes to find more detailed information.

Apart from these attributes of the scene object, also attributes for the coordinate system and the graphical objects inside the scene can be specified when generating a scene object. These attribute values are inherited to the coordinate system and the graphical objects as new default values.

A plot may contain more than one graphical scene. In such a case, separate scene objects must be created explicitly by the user and passed to a `plot` command (or inserted into an object of type `plot::Canvas`). For details on the layout of scenes inside the drawing area ("canvas"), see the help page of the canvas attribute `Layout`.

See "Example 1" on page 24-1167.

Attributes

Attribute	Purpose	Default Value
`BackgroundColor`	background color	`RGB::White`
`BackgroundTransparent`	plot a scene on a transparent background	`FALSE`
`BorderColor`	color of frame/border around canvas and scenes	`RGB::Grey50`
`BorderWidth`	width of frame/border around canvas and scenes	`0`
`Bottom`	distance of bottom of scene to bottom of canvas	`0`
`BottomMargin`	bottom margin width	`1`
`Footer`	footer text	
`FooterFont`	font of footers (scene and canvas)	`[" sans-serif ", 12]`

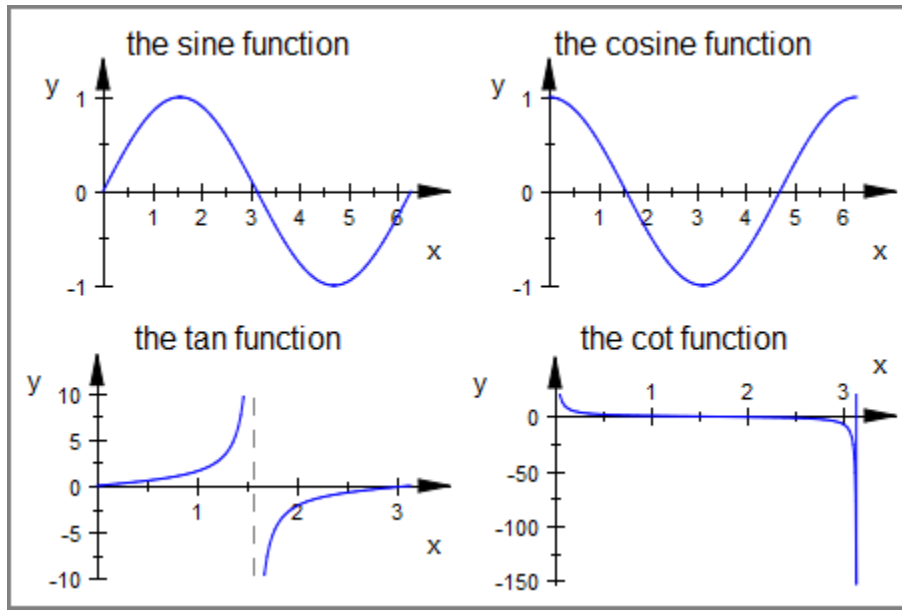
Attribute	Purpose	Default Value
FooterAlignment	alignment of footer of canvas and scenes	Center
Header	header text	
HeaderFont	font of headers (scene and canvas)	[" sans-serif ", 12]
HeaderAlignment	alignment of header of canvas and scenes	Center
Height	heights of canvas/scenes	80
Left	distance of left of scene to left of canvas	0
LeftMargin	left margin width	1
LegendFont	font of legend entries	[" sans-serif ", 8]
LegendVisible	switch legend on/off	FALSE
LegendPlacement	legend above or below	Bottom
LegendAlignment	legend at left, center, or right	Center
Margin	margins around canvas and scenes	1
Name	the name of a plot object (for browser and legend)	
RightMargin	right margin width	1
TopMargin	top margin width	1
Width	widths of canvas/scenes	120

Examples

Example 1

Scene objects have to be created explicitly only when several scenes are to be displayed simultaneously in one plot. The following call uses the automatic layout of several scenes in a canvas:

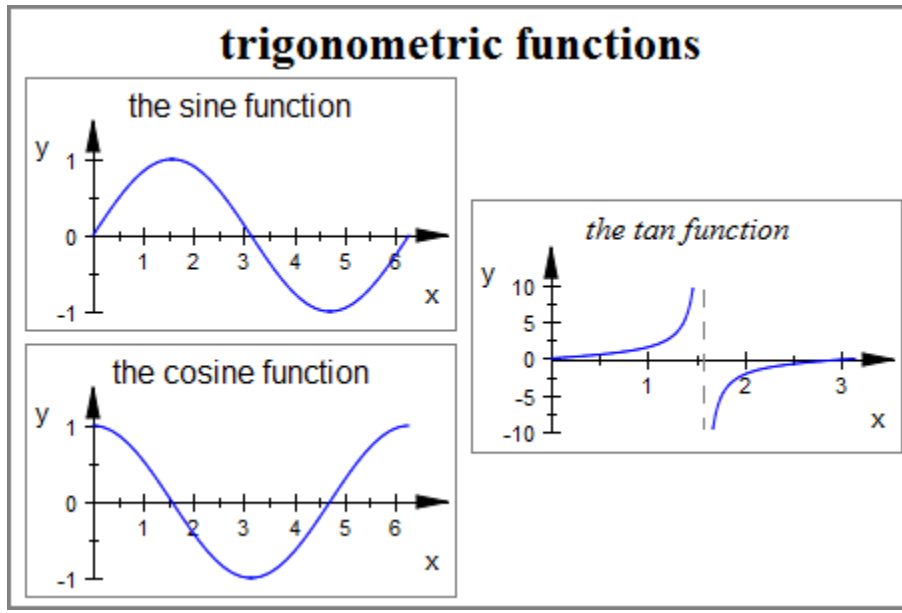
```
S1 := plot::Scene2d(plot::Function2d(sin(x), x = 0..2*PI),
                    Header = "the sine function"):
S2 := plot::Scene2d(plot::Function2d(cos(x), x = 0..2*PI),
                    Header = "the cosine function"):
S3 := plot::Scene2d(plot::Function2d(tan(x), x = 0..PI),
                    Header = "the tan function"):
S4 := plot::Scene2d(plot::Function2d(cot(x), x = 0..PI),
                    Header = "the cot function"):
plot(S1, S2, S3, S4, BorderWidth = 0.5*unit::mm)
```



We use the canvas attribute `Layout` = `Relative` to position 3 of these scenes in the canvas. The size of the scenes is set with the attributes `Width` and `Height`, specifying multiples of the canvas' width and height. The bottom left corner of each scene is positioned with the scene attributes `Bottom` and `Left`:

```
S1::Width := 0.475: S1::Height := 0.42:
S2::Width := 0.475: S2::Height := 0.42:
S3::Width := 0.475: S3::Height := 0.42:
S1::Bottom := 0.46: S1::Left := 0.02:
S2::Bottom := 0.02: S2::Left := 0.02:
S3::Bottom := 0.26: S3::Left := 0.51:
```

```
S3::HeaderFont := ["Times New Roman", Italic, 12]:
plot(S1, S2, S3, Layout = Relative,
     BorderWidth = 0.5*unit::mm,
     plot::Scene2d::BorderWidth = 0.2*unit::mm,
     Header = "trigonometric functions",
     HeaderFont = ["Times New Roman", Bold, 18]):
```

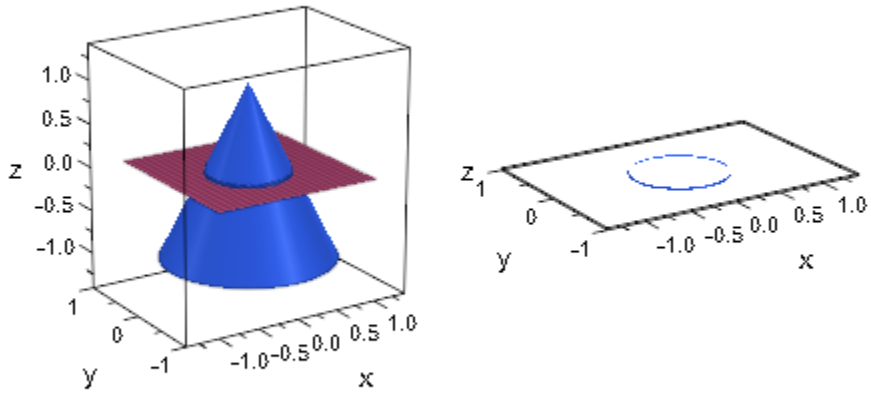



```
delete S1, S2, S3, S4:
```

Example 2

Conic sections are the curves that you get when intersecting a cone and a plane. The first scene displays a plane and a rotating cone, the second the corresponding conic section:

```
c := plot::Cone(1, [-sin(a), 0, -cos(a)], [sin(a), 0, cos(a)],
                a = 0..2*PI):
s := plot::Surface([x, y, 0], x = -1..1, y = -1..1):
S1 := plot::Scene3d(c, s):
S2 := plot::Scene3d(c, ViewingBoxZRange = -0.01 .. 0.01):
plot(S1, S2, Layout = Horizontal)
```




```
delete c, s, S1, S2:
```

Parameters

object2d$_1$, object2d$_2$, …

2D coordinate systems or graphical 2D objects

See Also

MuPAD Functions

`plot` | `plot::copy`

MuPAD Graphical Primitives

`plot::Canvas` | `plot::CoordinateSystem2d` | `plot::CoordinateSystem3d` | `plot::Scene3d`

plot::Scene3d

3D scenes

Syntax

```
plot::Scene3d(object3d1, object3d2, …, <a = amin .. amax>, options)
```

Description

`plot::Scene3d` is a container to display one or more coordinate systems containing graphical objects. Scene objects must be created when several graphical scenes are to be displayed simultaneously in a plot.

Usually, you do not need to create a scene object explicitly, because a `plot` command such as `plot(object1, object2, …)` creates a default scene object implicitly to display the graphical objects in.

The need for creating scene objects explicitly arises only when several scenes are to be displayed simultaneously in one plot.

The MuPAD graphics makes a clear division between 2D and 3D. Scene objects of type `plot::Scene2d` do not accept 3D objects and `plot::Scene3d` objects do not accept 2D objects. When several scenes are displayed simultaneously in a single plot, all scenes must be of the same dimension.

Strictly speaking, a 3D scene object is a container for coordinate systems of type `plot::CoordinateSystem3d`. However, you do not have to bother about this technicality because a suitable default coordinate system is created internally, when graphical primitives are passed to `plot::Scene3d`.

Scene objects are always visible in the interactive object browser of the MuPAD graphics tool (see section Viewer, Browser, and Inspector: Interactive Manipulation of this document). Each scene contains one or more coordinate systems as its children. When the scene object is selected, it provides access to a variety of attributes that are associated with scenes. The scene attributes allow to

- set annotations (`Header`, `Footer`) and control the `Legend`,

- set layout parameters such as height and width (if the canvas attribute `Layout` is set to `Absolute` or `Relative`),
- set a direction for the automatic camera in 3D (`CameraDirection`),
- set various style parameters such as `BackgroundColor` etc.

A complete listing of the attributes associated with a scene is given below. Follow the links to the help pages of the attributes to find more detailed information.

Apart from these attributes of the scene object, also attributes for the coordinate system and the graphical objects inside the scene can be specified when generating a scene object. These attribute values are inherited to the coordinate system and the graphical objects as new default values.

A plot may contain more than one graphical scene. In such a case, separate scene objects must be created explicitly by the user and passed to a `plot` command (or inserted into an object of type `plot::Canvas`). For details on the layout of scenes inside the drawing area (“canvas”), see the help page of the canvas attribute `Layout`.

Cf. “Example 1” on page 24-1174.

Attributes

Attribute	Purpose	Default Value
`BackgroundColor`	background color	`RGB::White`
`BackgroundColor2`	second background color for color blends	`RGB::Grey75`
`BackgroundStyle`	color blends in the background	`Flat`
`BackgroundTransparent`	plot a scene on a transparent background	`FALSE`
`BorderColor`	color of frame/border around canvas and scenes	`RGB::Grey50`
`BorderWidth`	width of frame/border around canvas and scenes	`0`
`Bottom`	distance of bottom of scene to bottom of canvas	`0`

Attribute	Purpose	Default Value
BottomMargin	bottom margin width	1
CameraDirection	the direction of the automatic camera	
CameraDirectionX	the direction of the automatic camera, x-component	
CameraDirectionY	the direction of the automatic camera, y-component	
CameraDirectionZ	the direction of the automatic camera, z-component	
Footer	footer text	
FooterFont	font of footers (scene and canvas)	[" sans-serif ", 12]
FooterAlignment	alignment of footer of canvas and scenes	Center
Header	header text	
HeaderFont	font of headers (scene and canvas)	[" sans-serif ", 12]
HeaderAlignment	alignment of header of canvas and scenes	Center
Height	heights of canvas/scenes	80
Left	distance of left of scene to left of canvas	0
LeftMargin	left margin width	1
LegendFont	font of legend entries	[" sans-serif ", 8]
LegendVisible	switch legend on/off	FALSE
LegendPlacement	legend above or below	Bottom
LegendAlignment	legend at left, center, or right	Center

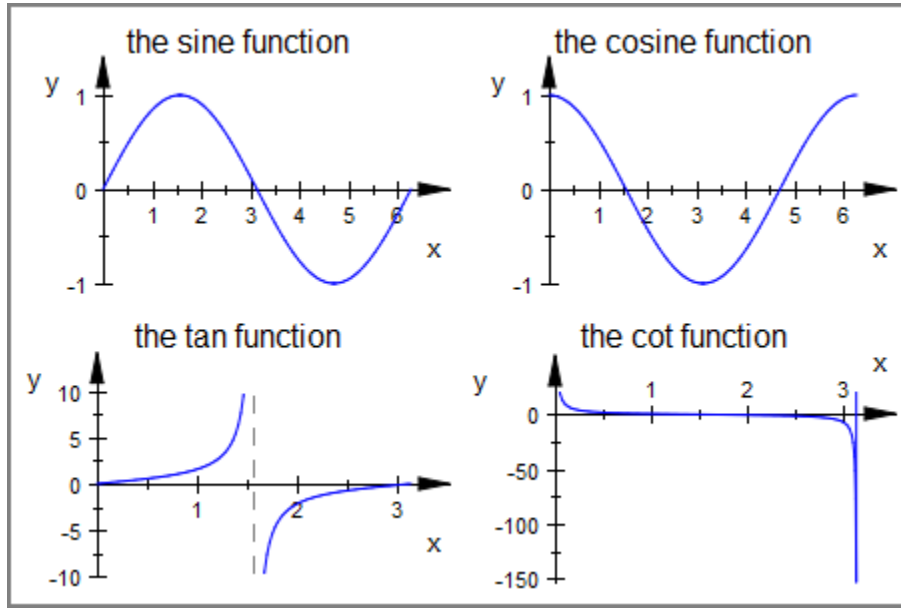
Attribute	Purpose	Default Value
`Lighting`	light schemes for 3D graphics	`Automatic`
`Margin`	margins around canvas and scenes	1
`Name`	the name of a plot object (for browser and legend)	
`RightMargin`	right margin width	1
`TopMargin`	top margin width	1
`Width`	widths of canvas/scenes	120
`YXRatio`	scaling ratio between y and x axes	1
`ZXRatio`	scaling ratio between z and x axes	2/3

Examples

Example 1

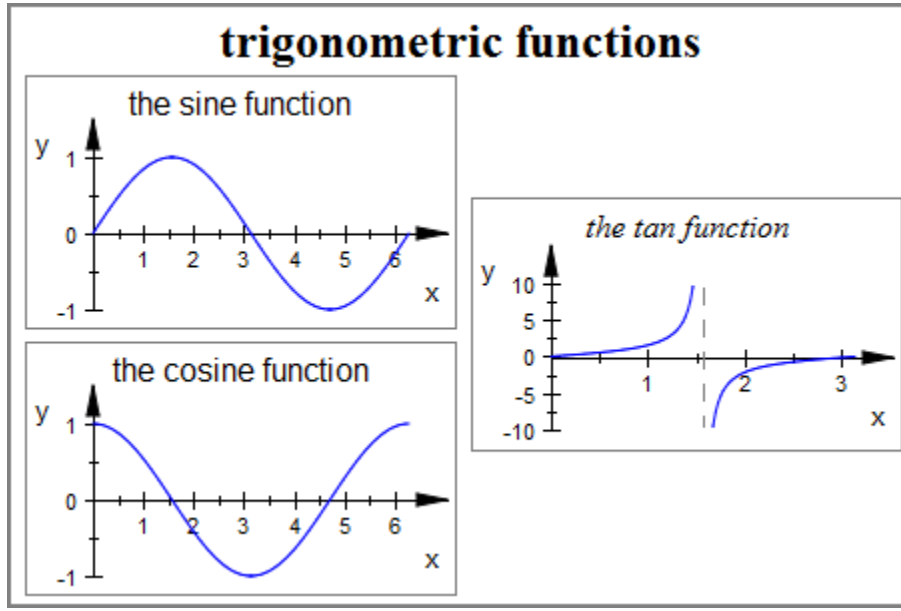
Scene objects have to be created explicitly only when several scenes are to be displayed simultaneously in one plot. The following call uses the automatic layout of several scenes in a canvas:

```
S1 := plot::Scene2d(plot::Function2d(sin(x), x = 0..2*PI),
                    Header = "the sine function"):
S2 := plot::Scene2d(plot::Function2d(cos(x), x = 0..2*PI),
                    Header = "the cosine function"):
S3 := plot::Scene2d(plot::Function2d(tan(x), x = 0..PI),
                    Header = "the tan function"):
S4 := plot::Scene2d(plot::Function2d(cot(x), x = 0..PI),
                    Header = "the cot function"):
plot(S1, S2, S3, S4, BorderWidth = 0.5*unit::mm)
```



We use the canvas attribute `Layout` = `Relative` to position 3 of these scenes in the canvas. The size of the scenes is set with the attributes `Width` and `Height`, specifying multiples of the canvas' width and height. The bottom left corner of each scene is positioned with the scene attributes `Bottom` and `Left`:

```
S1::Width := 0.475: S1::Height := 0.42:
S2::Width := 0.475: S2::Height := 0.42:
S3::Width := 0.475: S3::Height := 0.42:
S1::Bottom := 0.46: S1::Left := 0.02:
S2::Bottom := 0.02: S2::Left := 0.02:
S3::Bottom := 0.26: S3::Left := 0.51:
S3::HeaderFont := ["Times New Roman", Italic, 12]:
plot(S1, S2, S3, Layout = Relative,
     BorderWidth = 0.5*unit::mm,
     plot::Scene2d::BorderWidth = 0.2*unit::mm,
     Header = "trigonometric functions",
     HeaderFont = ["Times New Roman", Bold, 18]):
```

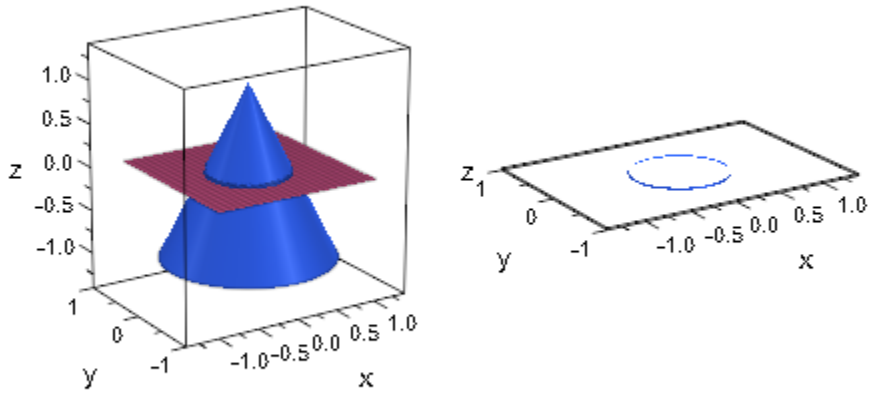



```
delete S1, S2, S3, S4:
```

Example 2

Conic sections are the curves that you get when intersecting a cone and a plane. The first scene displays a plane and a rotating cone, the second the corresponding conic section:

```
c := plot::Cone(1, [-sin(a), 0, -cos(a)], [sin(a), 0, cos(a)],
                a = 0..2*PI):
s := plot::Surface([x, y, 0], x = -1..1, y = -1..1):
S1 := plot::Scene3d(c, s):
S2 := plot::Scene3d(c, ViewingBoxZRange = -0.01 .. 0.01):
plot(S1, S2, Layout = Horizontal)
```




```
delete c, s, S1, S2:
```

Parameters

object3d$_1$, object3d$_2$, …

3D coordinate systems or graphical 3D objects

See Also

MuPAD Functions

`plot` | `plot::copy`

MuPAD Graphical Primitives

`plot::Canvas` | `plot::CoordinateSystem2d` | `plot::CoordinateSystem3d` | `plot::Scene2d`

plot::ClippingBox

Clipping of 3D objects

Syntax

```
plot::ClippingBox(xmin .. xmax, ymin .. ymax, zmin .. zmax, <a = amin .. amax>, options)
```

Description

A `plot::ClippingBox` defines a cubic box with edges parallel to the coordinate axes. When a clipping box is inserted in a 3D scene, only the parts of the graphical objects in the scene are visible that lie inside the coordinate range defined by the clipping box.

Inserting a `plot::ClippingBox` into a 3D scene has a similar effect as specifying a viewing box for the scene by the attribute `ViewingBox`.

However, the specified viewing box fills the entire drawing region of the plot, whereas a `plot::ClippingBox` preserves the space in the drawing region that the invisible parts would fill if no clipping box was used.

Moreover, in contrast to `plot::ClippingBox`, the visibility range defined by the `ViewingBox` cannot be animated.

In fact, the main purpose of `plot::ClippingBox` is to provide an animated version of the `ViewingBox`.

Size and location of the `ViewingBox` remain unaffected by the presence of a clipping box. Also coordinate axes are not clipped.

Only one single `plot::ClippingBox` should be used inside a 3D scene.

Attributes

Attribute	Purpose	Default Value
`Frames`	the number of frames in an animation	`50`

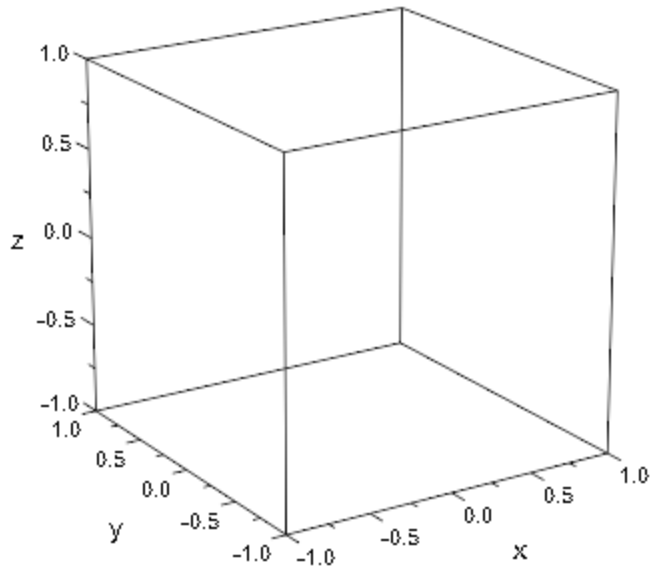
Attribute	Purpose	Default Value
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`
`Visible`	visibility	`TRUE`
`XMax`	final value of parameter “x”	
`XMin`	initial value of parameter “x”	
`XRange`	range of parameter “x”	
`YMax`	final value of parameter “y”	
`YMin`	initial value of parameter “y”	
`YRange`	range of parameter “y”	
`ZMax`	final value of parameter “z”	
`ZMin`	initial value of parameter “z”	
`ZRange`	range of parameter “z”	

Examples

Example 1

We plot a full sphere yet rendering only a part of it visible. This is done by passing a suitable animated `plot::ClippingBox` to the `plot` command. Note that the viewing box remains unaffected:

```
plot(plot::Sphere(1, [0, 0, 0]),
     plot::ClippingBox(-1 + a .. 1 - a,
                       -1 + a .. 1 - a,
                       -1 .. 1, a = 0..1))
```



Example 2

We plot a Klein bottle. By chopping off the upper parts, one can have a look inside:

```
KleinBottle := plot::Tube([6*cos(u)*(sin(u)-1), 0, 14*sin(u)],
                          4 - 2*cos(u), u = -PI..PI):
C := plot::ClippingBox(-15..15, -10..10, -20.. a,
```

```
                          a = 15 .. -20):
plot(KleinBottle, C, Axes = None)
```

```
delete KleinBottle, C:
```

Parameters

x_{min}, x_{max}

The borders of the visible range of the *x* coordinate: numerical real values or arithmetical expressions of the animation parameter `a`.

x_{min}, x_{max} are equivalent to the attributes `XMin`, `XMax`.

y_{min}, y_{max}

The borders of the visible range of the *y* coordinate: numerical real values or arithmetical expressions of the animation parameter `a`.

y_{min}, y_{max} are equivalent to the attributes `YMin`, `YMax`.

z_{min}, z_{max}

The borders of the visible range of the *z* coordinate: numerical real values or arithmetical expressions of the animation parameter `a`.

z_{min}, z_{max} are equivalent to the attributes `ZMin`, `ZMax`.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::CoordinateSystem3d`

plot::Reflect2d

Reflection about a 2D point or a line

Syntax

```
plot::Reflect2d([x1, y1], <[x2, y2]>, obj1, obj2, …, <a = amin .. amax>, options)
```

Description

`plot::Reflect2d([ x1, y1], object)` reflects a 2D object about the point (x_1, y_1).

`plot::Reflect2d([ x1, y1], [ x2, y2], object)` reflects a 2D object about the line through the points (x_1, y_1) and (x_2, y_2).

Reflections are transformation objects that mirror their contents about a straight line (in 2D) or a plane (in 3D). In the degenerate case where both points on the line coincide, or the normal vector is given as `[0, 0, 0]`, or if only one point is specified, they reflect about a point.

Like all transformation objects, reflections may contain any number of objects of the appropriate dimension. Plotting the reflection object renders the reflections of all graphical objects inside.

Reflections can be animated. If the contained objects are animated, too, the animations will run simultaneously.

Animated reflection objects are rather "cheap" concerning computing and storing costs. For more complex graphical objects, it is more efficient to use an animated reflection object than to redefine the object for each frame.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`

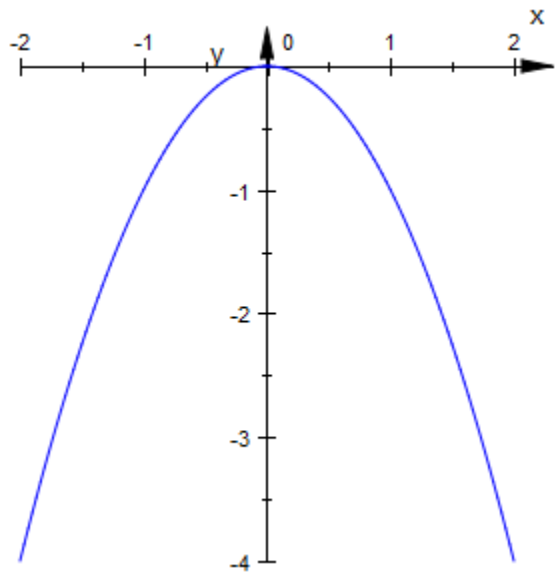
Attribute	Purpose	Default Value
`Frames`	the number of frames in an animation	`50`
`From`	starting point of arrows and lines	
`FromX`	starting point of arrows and lines, x-coordinate	
`FromY`	starting point of arrows and lines, y-coordinate	
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`
`To`	end point of arrows and lines	
`ToX`	end point of arrows and lines, x-coordinate	
`ToY`	end point of arrows and lines, y-coordinate	

Examples

Example 1

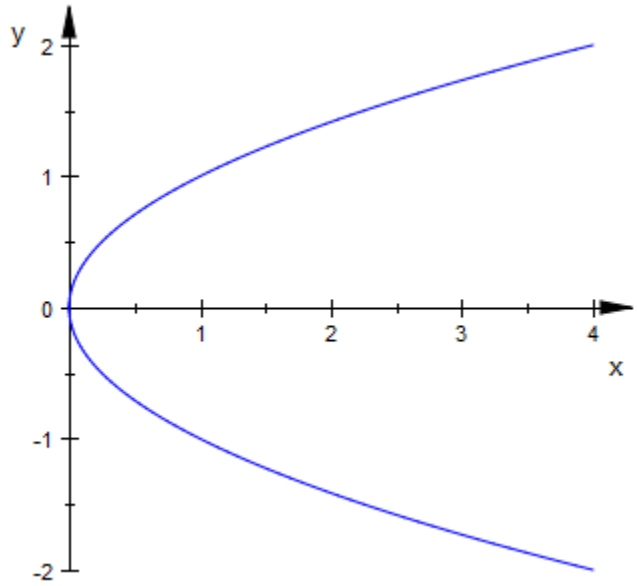
We plot the reflection of a function graph about the origin:

```
plot(plot::Reflect2d([0, 0],
                     plot::Function2d(x^2, x=-2..2)))
```



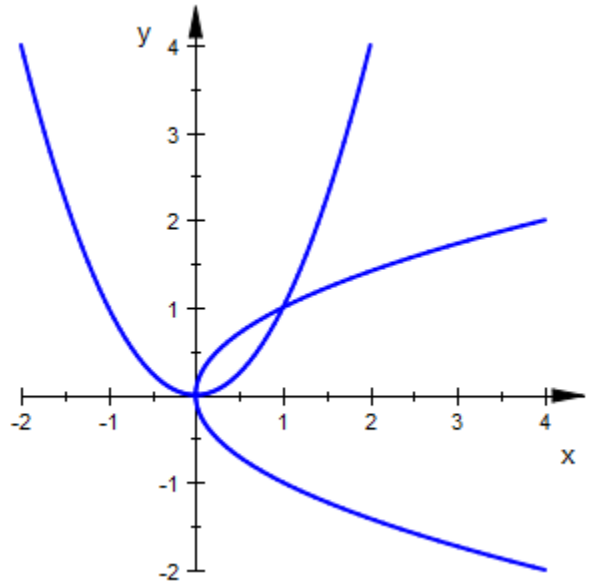
Reflecting a function about the main diagonal (the line through the origin and the point (1, 1) shows the (multivalued) inverse function:

```
plot(plot::Reflect2d([0, 0], [1, 1],
                     plot::Function2d(x^2, x=-2..2)))
```



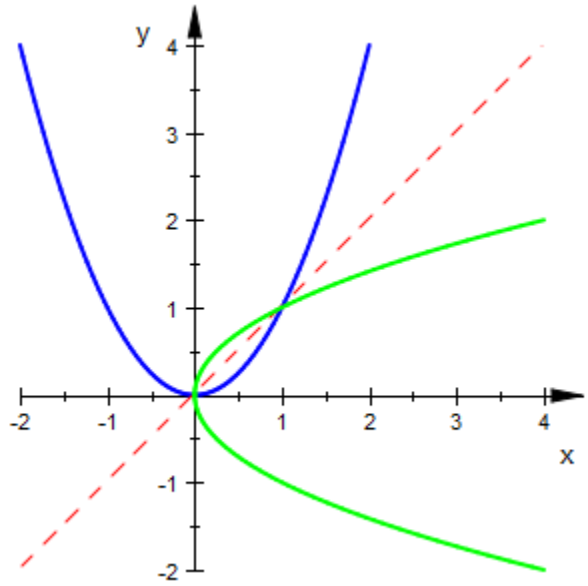
To display both an object and its mirror image, assign the object to some variable and plot both:

```
f := plot::Function2d(x^2, x=-2..2, LineWidth = 0.5):
plot(f, plot::Reflect2d([0, 0], [1, 1], f))
```



The following command shows two more useful variations: First, we use `plot::Line2d` to display the line of reflection. Second, we employ `plot::modify` to change the line color of the mirrored function graph:

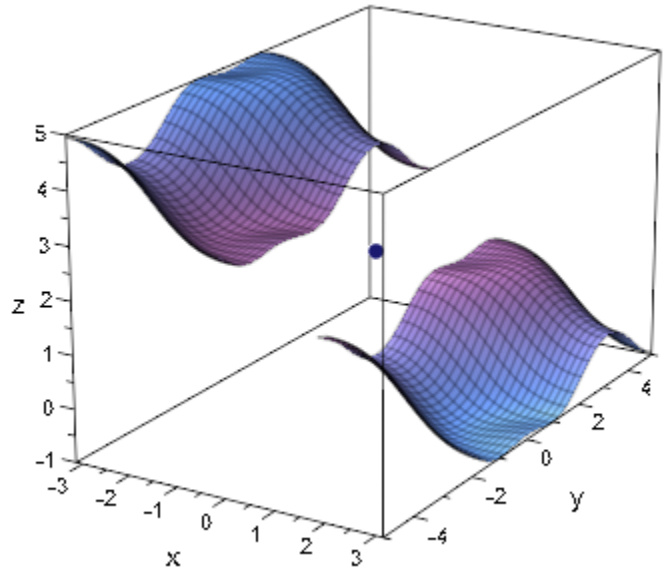
```
g := plot::Line2d([0, 0], [1, 1],
                  Color = RGB::Red,
                  LineStyle = Dashed,
                  Extension = Infinite):
f1 := plot::Reflect2d([0, 0], [1, 1],
                 plot::modify(f, LineColor = RGB::Green)):
plot(f, g, f1)
```



Example 2

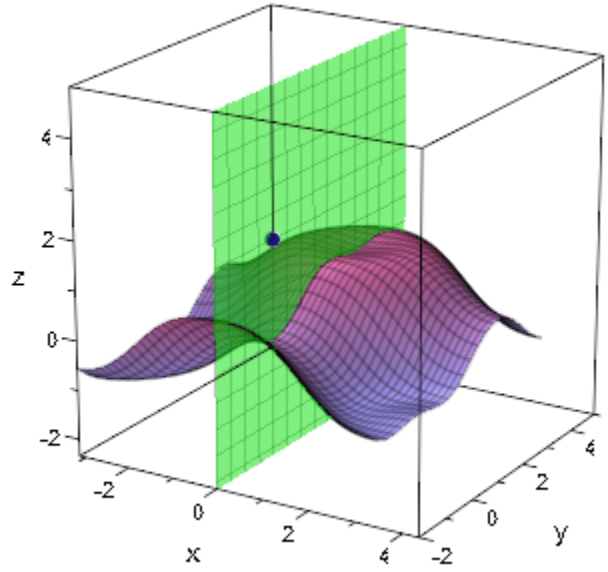
A 3D function graph and its reflection about the point (0, 0, 2):

```
f := plot::Function3d(sin(cos(x) - cos(y)), x = 0..PI, y = -2..5):
p := plot::Point3d([0 , 0, 2], PointSize=2):
plot(f, plot::Reflect3d([0, 0, 2], f), p,
     CameraDirection=[30, -50, 20])
```



The same function graph and its reflection at a plane through the point (0, 0, 2) with an animated normal vector:

```
pl := plot::Plane([0, 0, 2], [a, 0, 1-a], a=0..1, Color=RGB::Green.[0.5]):
plot(f, plot::Reflect3d([0, 0, 2], [a, 0, 1-a], a=0..1, f), p, pl,
     CameraDirection=[30, -50, 20])
```



Parameters

x_1, y_1, x_2, y_2

The coordinates of two points on a line: real numerical values or arithmetical expressions of the animation parameter `a`.

x_1, y_1, x_2, y_2 are equivalent to the attributes `From`, `To`, `FromX`, `FromY`, `ToX`, `ToY`.

`obj1, obj2, …`

Arbitrary plot objects of the appropriate dimension

`a`

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Line2d` | `plot::Plane` | `plot::Reflect3d` | `plot::Transform2d` | `plot::Transform3d`

plot::Reflect3d

Reflection about a 3D point or a plane

Syntax

```
plot::Reflect3d([x, y, z], <[n_x, n_y, n_z]>, obj1, obj2, …, <a = a_min .. a_max>, options)
```

Description

`plot::Reflect3d([x, y, z], object)` reflects a 3D object about the point (x, y, z).

`plot::Reflect3d([x, y, z], [n_x, n_y, n_z], object)` reflects a 3D object about the plane through the point (x, y, z) with normal (n_x, n_y, n_z).

Reflections are transformation objects that mirror their contents about a straight line (in 2D) or a plane (in 3D). In the degenerate case where both points on the line coincide, or the normal vector is given as `[0, 0, 0]`, or if only one point is specified, they reflect about a point.

Like all transformation objects, reflections may contain any number of objects of the appropriate dimension. Plotting the reflection object renders the reflections of all graphical objects inside.

Reflections can be animated. If the contained objects are animated, too, the animations will run simultaneously.

Animated reflection objects are rather "cheap" concerning computing and storing costs. For more complex graphical objects, it is more efficient to use an animated reflection object than to redefine the object for each frame.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`

Attribute	Purpose	Default Value
`Frames`	the number of frames in an animation	`50`
`Name`	the name of a plot object (for browser and legend)	
`Normal`	normal vector of circles and discs, etc. in 3D	[0, 0, 1]
`NormalX`	normal vector of circles and discs, etc. in 3D, x-component	`0`
`NormalY`	normal vector of circles and discs, etc. in 3D, y-component	`0`
`NormalZ`	normal vector of circles and discs, etc. in 3D, z-component	`1`
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`Position`	positions of cameras, lights, and text objects	[0, 0, 0]
`PositionX`	x-positions of cameras, lights, and text objects	`0`
`PositionY`	y-positions of cameras, lights, and text objects	`0`
`PositionZ`	z-positions of cameras, lights, and text objects	`0`
`TimeEnd`	end time of the animation	`10.0`

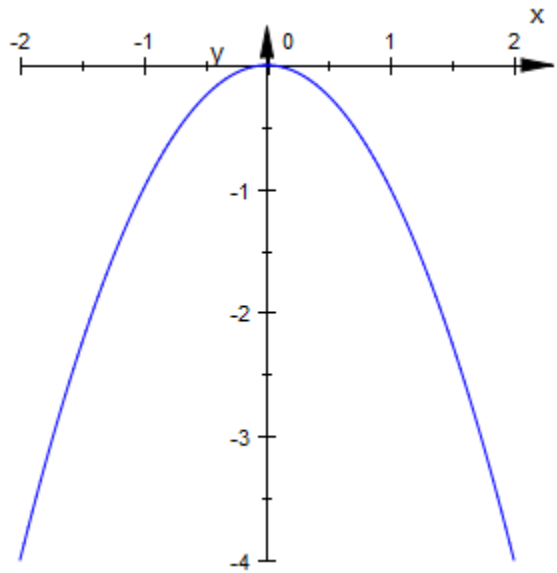
Attribute	Purpose	Default Value
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`

Examples

Example 1

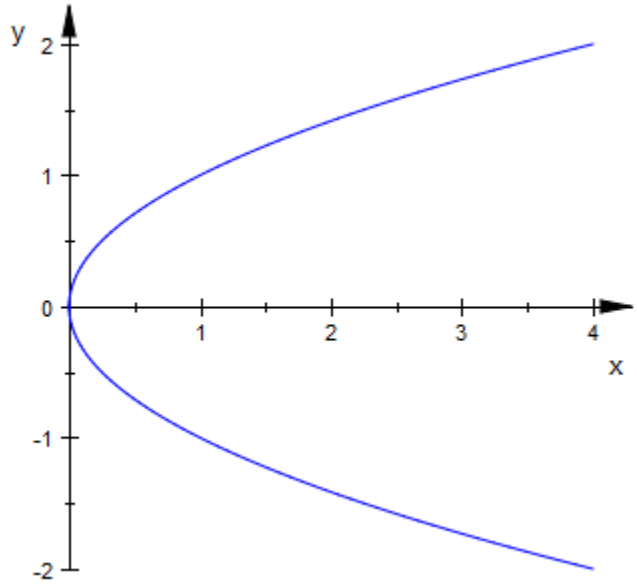
We plot the reflection of a function graph about the origin:

```
plot(plot::Reflect2d([0, 0],
                     plot::Function2d(x^2, x=-2..2)))
```



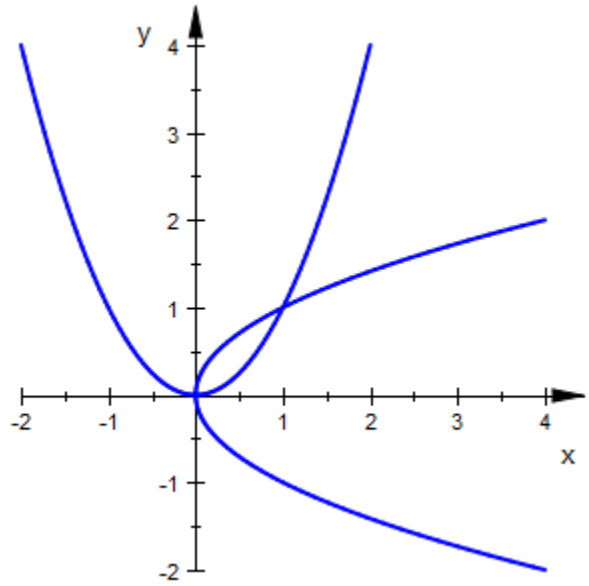
Reflecting a function about the main diagonal (the line through the origin and the point (1, 1) shows the (multivalued) inverse function:

```
plot(plot::Reflect2d([0, 0], [1, 1],
                     plot::Function2d(x^2, x=-2..2)))
```



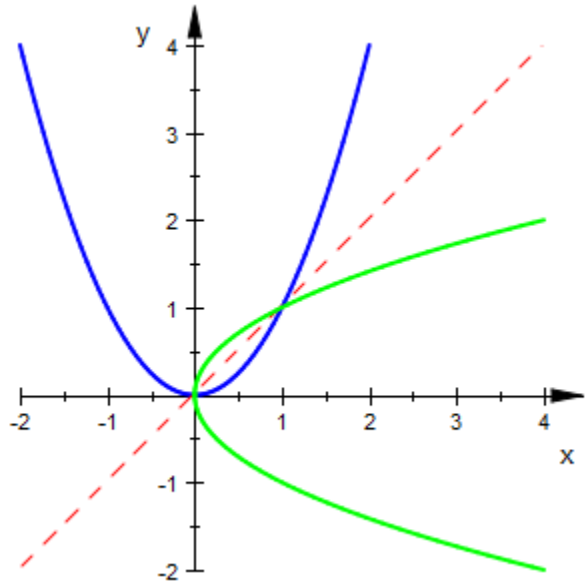
To display both an object and its mirror image, assign the object to some variable and plot both:

```
f := plot::Function2d(x^2, x=-2..2, LineWidth = 0.5):
plot(f, plot::Reflect2d([0, 0], [1, 1], f))
```



The following command shows two more useful variations: First, we use `plot::Line2d` to display the line of reflection. Second, we employ `plot::modify` to change the line color of the mirrored function graph:

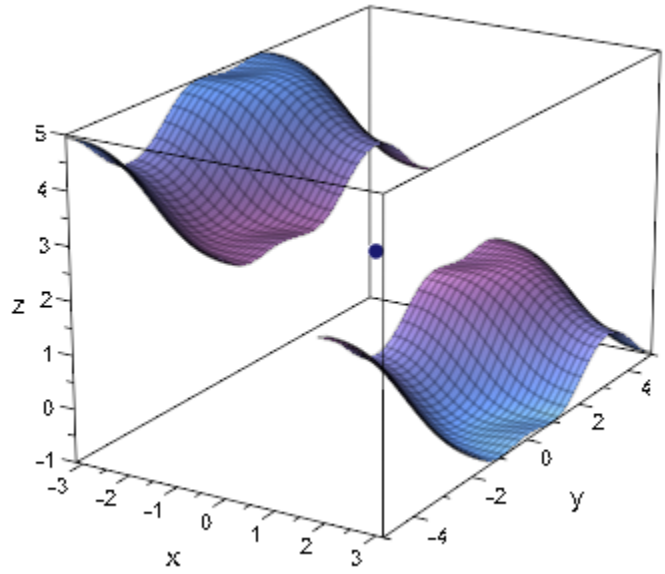
```
g := plot::Line2d([0, 0], [1, 1],
                  Color = RGB::Red,
                  LineStyle = Dashed,
                  Extension = Infinite):
f1 := plot::Reflect2d([0, 0], [1, 1],
                 plot::modify(f, LineColor = RGB::Green)):
plot(f, g, f1)
```



Example 2

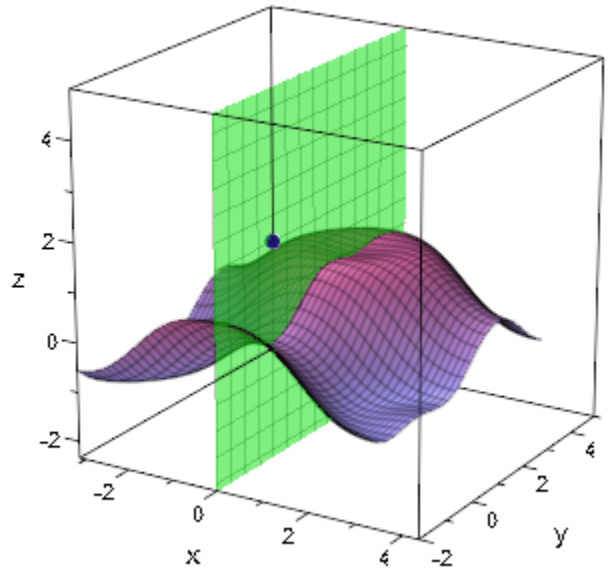
A 3D function graph and its reflection about the point (0, 0, 2):

```
f := plot::Function3d(sin(cos(x) - cos(y)), x = 0..PI, y = -2..5):
p := plot::Point3d([0 , 0, 2], PointSize=2):
plot(f, plot::Reflect3d([0, 0, 2], f), p,
     CameraDirection=[30, -50, 20])
```



The same function graph and its reflection at a plane through the point (0, 0, 2) with an animated normal vector:

```
pl := plot::Plane([0, 0, 2], [a, 0, 1-a], a=0..1, Color=RGB::Green.[0.5]):
plot(f, plot::Reflect3d([0, 0, 2], [a, 0, 1-a], a=0..1, f), p, pl,
     CameraDirection=[30, -50, 20])
```



Parameters

x, y, z

The coordinates of the mirror point or a point on the mirror plane, respectively: real numerical values or arithmetical expressions of the animation parameter `a`.

x, y, z are equivalent to the attributes `Position`, `PositionX`, `PositionY`, `PositionZ`.

n_x, n_y, n_z

The coordinates of the normal of the mirror plane: real numerical values or arithmetical expressions of the animation parameter `a`.

n_x, n_y, n_z are equivalent to the attributes `Normal`, `NormalX`, `NormalY`, `NormalZ`.

`obj1, obj2, …`

Arbitrary plot objects of the appropriate dimension

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Line2d` | `plot::Plane` | `plot::Reflect2d` | `plot::Transform2d` | `plot::Transform3d`

plot::Rotate2d

Rotations of 2D objects

Syntax

`plot::Rotate2d(angle, <[`c_x`, `c_y`]>, `obj_1`, <`obj_2`, …>, <a = `a_{min}` .. `a_{max}`>, options)`

Description

`plot::Rotate2d(angle, [`c_x`, `c_y`], object)` rotates a 2D object counter clockwise by the given angle around the rotation center `[`c_x`, `c_y`]`.

In 2D, the direction of the rotation is counter clock wise. Use negative angles to rotate clock wise.

Rotate objects can rotate several graphical objects simultaneously. Plotting the rotate object renders all graphical objects inside.

Rotated objects have a tendency to overestimate their `ViewingBox`. Cf. the help page of `ViewingBox`. In such a case, you should specify a suitable `ViewingBox` explicitly.

Transformation objects can be used inside rotate objects. If they are animated, the animations run simultaneously.

Animated rotate objects are rather “cheap” concerning computing and storing costs. For more complex graphical objects, it is more efficient to use an animated rotate object than to redefine the object for each frame.

The function `op` allows to extract the graphical objects inside a rotate object.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`

Attribute	Purpose	Default Value
`Angle`	rotation angle	`0`
`Center`	center of objects, rotation center	[0, 0]
`CenterX`	center of objects, rotation center, x-component	`0`
`CenterY`	center of objects, rotation center, y-component	`0`
`Frames`	the number of frames in an animation	`50`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`

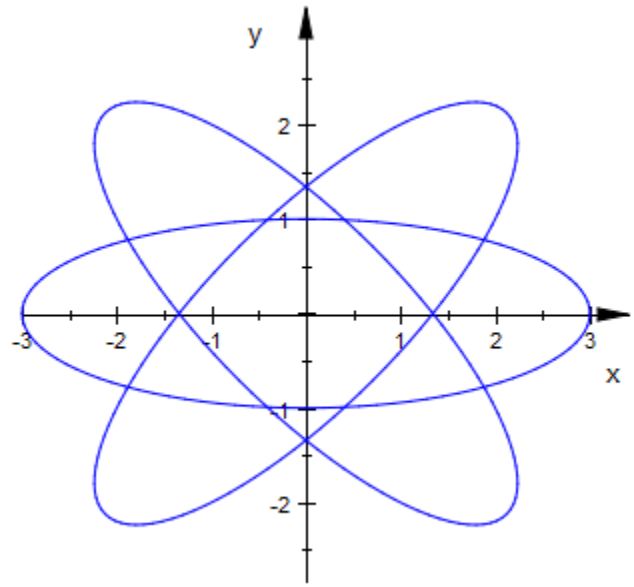
Examples

Example 1

Ellipses of type `plot::Ellipse2d` have symmetry axes parallel to the coordinate axes. You can use `plot::Rotate2d` to obtain ellipses with other orientations:

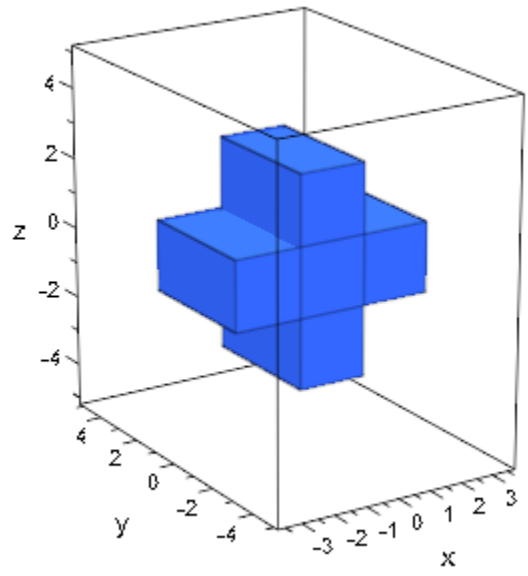
```
e0 := plot::Ellipse2d(3, 1, [0, 0]):
```

```
e1 :=  plot::Rotate2d(PI/4, [0, 0], e0):
e2 :=  plot::Rotate2d(-PI/4, [0, 0], e0):
plot(e0, e1, e2):
```



Similarly, 3D boxes with arbitrary orientation can be generated via `plot::Rotate3d`. We use several animated rotation objects:

```
b0 := plot::Box(-3..3, -2..2, -1..1):
b1 :=  plot::Rotate3d(a, [0, 0, 0], [0, 0, 1], b0,
                      a = 0..PI/2, TimeRange = 0..3):
b2 :=  plot::Rotate3d(a, [0, 0, 0], [0, 1, 0], b1,
                      a = 0..PI/2, TimeRange = 3..6):
b3 :=  plot::Rotate3d(a, [0, 0, 0], [1, 0, 0], b2,
                      a = 0..PI/2, TimeRange = 6..9):
plot(b0, b3):
```

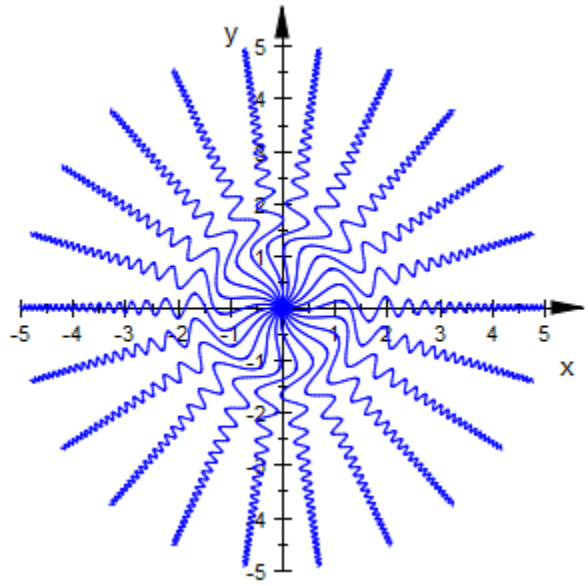



```
delete e0, e1, e2, b0, b1, b2, b3:
```

Example 2

We plot several copies of a function plot, rotated by different angles:

```
f := plot::Function2d(sin(x^3)/(x^2+1), x = -5..5, Mesh = 300):
plot(plot::Rotate2d(f, Angle = PI/11*a) $ a = 0..10):
```



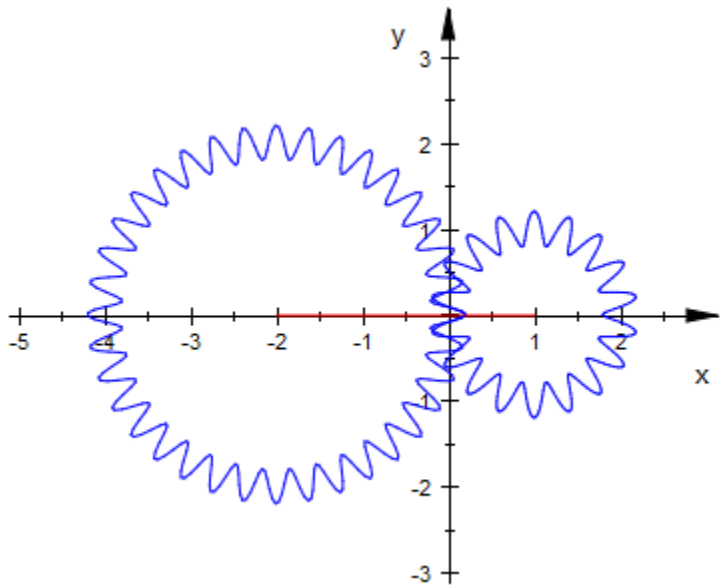

```
delete f:
```

Example 3

We plot turning cogs. Each animated rotate object rotates a curve and a line simultaneously:

```
r1 := 2: x1 := -r1: y1:= 0:
r2 := 1: x2 :=  r2: y2:= 0:
dr := 0.2:
cog1 := plot::Curve2d([x1 + (r1 + dr*cos(36*u))*cos(u),
                       y1 + (r1 + dr*cos(36*u))*sin(u)],
                      u = 0..2*PI, Mesh = 360):
cog2 := plot::Curve2d([x2 + (r2 - dr*cos(18*u))*cos(u),
                       y2 + (r2 - dr*cos(18*u))*sin(u)],
                       u = 0..2*PI, Mesh = 360):
line1 :=  plot::Line2d([x1, y1], [x1 + r1 + dr, y1],
                       Color = RGB::Red):
line2 :=  plot::Line2d([x2, y2], [x2 - r2 + dr, y2],
                       Color = RGB::Red):
```

```
Cog1 := plot::Rotate2d(-a, [x1,  y1], cog1, line1,
                       a = 0..2*PI, Frames = 180):
Cog2 := plot::Rotate2d(2*a, [x2, y2], cog2, line2,
                       a = 0..2*PI, Frames = 180):
plot(Cog1, Cog2, Scaling = Constrained):
```

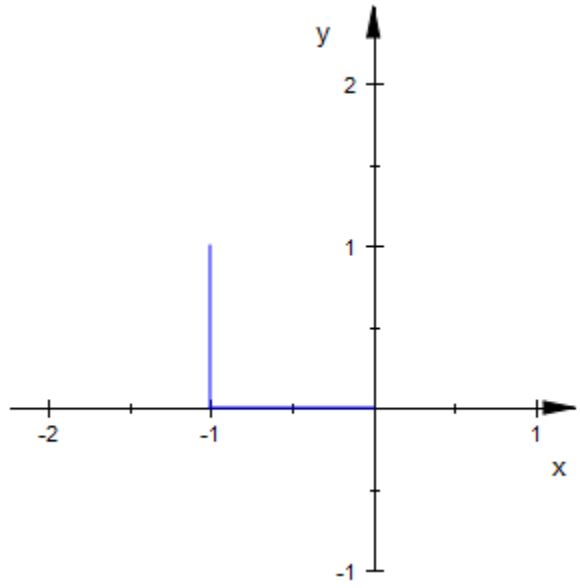



```
delete r1, x1, y1, r2, x2, y2, dr, cog1, cog2,
       line1, line2, Cog1, Cog2:
```

Example 4

We use an animated rotation inside another animated rotation:

```
L1 := plot::Line2d([0, 0], [0, 1]):
L2 := plot::Rotate2d(a, [0, 1], a = 0..2*PI,
                     plot::Line2d([0, 1], [1, 1])):
plot(plot::Rotate2d(a, [0, 0], L1, L2, a = 0..PI/2)):
```




```
delete L1, L2:
```

Parameters

`angle`

The rotation angle in radians: a numerical real value or an arithmetical expression of the animation parameter `a`.

angle is equivalent to the attribute `Angle`.

c_x, c_y

The components of the rotation center: numerical real values or arithmetical expressions of the animation parameter `a`. If no rotation center is specified, the center `[0, 0, 0]` is used.

c_x, c_y are equivalent to the attributes `Center`, `CenterX`, `CenterY`.

obj_1, obj_2, …

Arbitrary plot objects of the appropriate dimension

a

Animation parameter, specified as a = a_{min}..a_{max}, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Rotate3d` | `plot::Scale2d` | `plot::Scale3d` | `plot::Transform2d` | `plot::Transform3d` | `plot::Translate2d` | `plot::Translate3d`

plot::Rotate3d

Rotations of 3D objects

Syntax

```
plot::Rotate3d(angle, <[c_x, c_y, c_z], [d_x, d_y, d_z]>, obj_1, <obj_2, …>, <a = a_min .. a_max>, op
```

Description

`plot::Rotate3d(angle, [c_x, c_y, c_z], [d_x, d_y, d_z], object)` rotates a 3D object by the given angle around the rotation axis defined by the point `[c_x, c_y, c_z]` and the direction `[d_x, d_y, d_z]`.

In 3D, the rotation is implemented following the “right hand rule”: Stretch the thumb of your right hand and bend the fingers. When the thumb points into the direction of the rotation axis, your finger tips indicate the direction of the rotation.

Use negative angles to rotate in the opposite direction.

Rotate objects can rotate several graphical objects simultaneously. Plotting the rotate object renders all graphical objects inside.

Rotated objects have a tendency to overestimate their `ViewingBox`. Cf. the help page of `ViewingBox`. In such a case, you should specify a suitable `ViewingBox` explicitly.

Transformation objects can be used inside rotate objects. If they are animated, the animations run simultaneously.

Animated rotate objects are rather “cheap” concerning computing and storing costs. For more complex graphical objects, it is more efficient to use an animated rotate object than to redefine the object for each frame.

The function `op` allows to extract the graphical objects inside a rotate object.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Angle`	rotation angle	`0`
`Axis`	rotation axis	`[0, 0, 1]`
`AxisX`	x-component of rotation axis	`0`
`AxisY`	y-component of rotation axis	`0`
`AxisZ`	z-component of rotation axis	`1`
`Center`	center of objects, rotation center	`[0, 0, 0]`
`CenterX`	center of objects, rotation center, x-component	`0`
`CenterY`	center of objects, rotation center, y-component	`0`
`CenterZ`	center of objects, rotation center, z-component	`0`
`Frames`	the number of frames in an animation	`50`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`

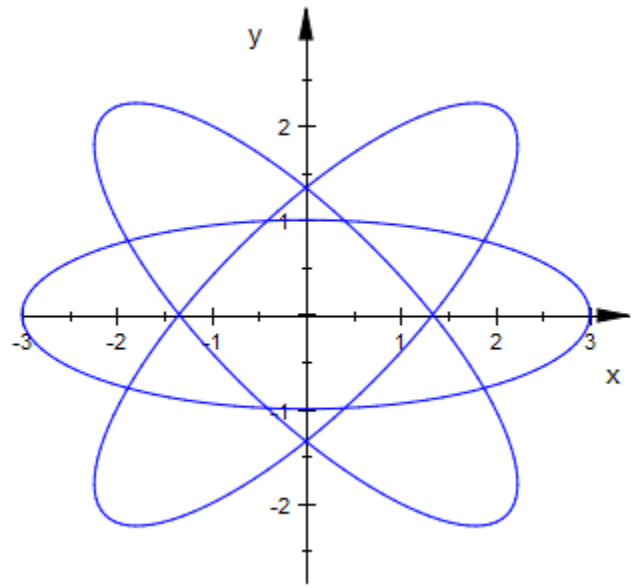
Attribute	Purpose	Default Value
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`

Examples

Example 1

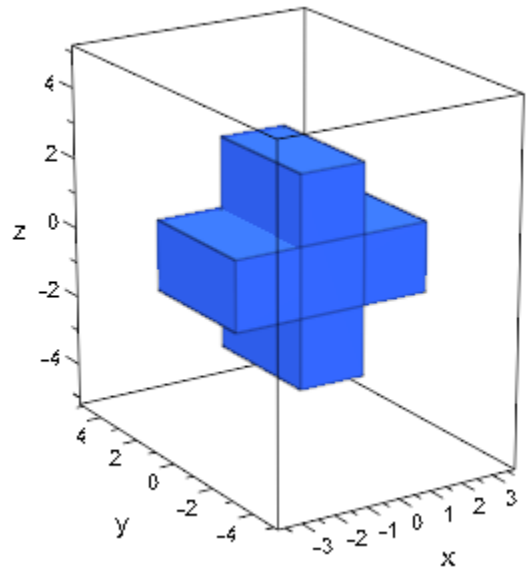
Ellipses of type `plot::Ellipse2d` have symmetry axes parallel to the coordinate axes. You can use `plot::Rotate2d` to obtain ellipses with other orientations:

```
e0 := plot::Ellipse2d(3, 1, [0, 0]):
e1 :=  plot::Rotate2d(PI/4, [0, 0], e0):
e2 :=  plot::Rotate2d(-PI/4, [0, 0], e0):
plot(e0, e1, e2):
```



Similarly, 3D boxes with arbitrary orientation can be generated via `plot::Rotate3d`. We use several animated rotation objects:

```
b0 := plot::Box(-3..3, -2..2, -1..1):
b1 :=  plot::Rotate3d(a, [0, 0, 0], [0, 0, 1], b0,
                      a = 0..PI/2, TimeRange = 0..3):
b2 :=  plot::Rotate3d(a, [0, 0, 0], [0, 1, 0], b1,
                      a = 0..PI/2, TimeRange = 3..6):
b3 :=  plot::Rotate3d(a, [0, 0, 0], [1, 0, 0], b2,
                      a = 0..PI/2, TimeRange = 6..9):
plot(b0, b3):
```

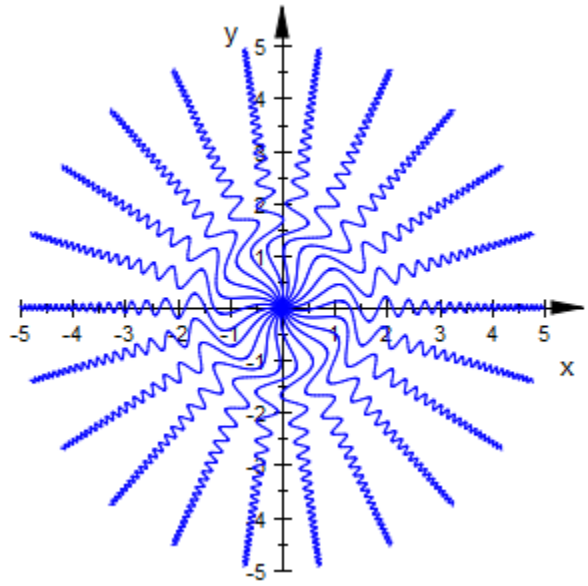



```
delete e0, e1, e2, b0, b1, b2, b3:
```

Example 2

We plot several copies of a function plot, rotated by different angles:

```
f := plot::Function2d(sin(x^3)/(x^2+1), x = -5..5, Mesh = 300):
plot(plot::Rotate2d(f, Angle = PI/11*a) $ a = 0..10):
```



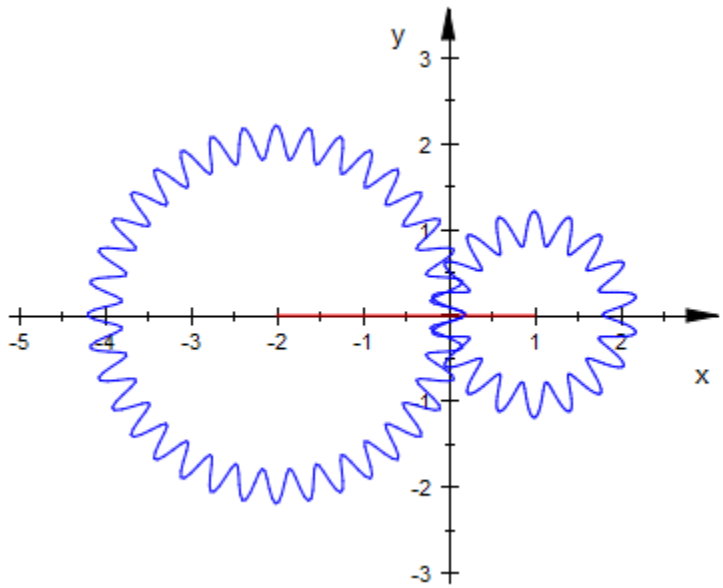

```
delete f:
```

Example 3

We plot turning cogs. Each animated rotate object rotates a curve and a line simultaneously:

```
r1 := 2: x1 := -r1: y1:= 0:
r2 := 1: x2 :=  r2: y2:= 0:
dr := 0.2:
cog1 := plot::Curve2d([x1 + (r1 + dr*cos(36*u))*cos(u),
                       y1 + (r1 + dr*cos(36*u))*sin(u)],
                      u = 0..2*PI, Mesh = 360):
cog2 := plot::Curve2d([x2 + (r2 - dr*cos(18*u))*cos(u),
                       y2 + (r2 - dr*cos(18*u))*sin(u)],
                       u = 0..2*PI, Mesh = 360):
line1 :=  plot::Line2d([x1, y1], [x1 + r1 + dr, y1],
                       Color = RGB::Red):
line2 :=  plot::Line2d([x2, y2], [x2 - r2 + dr, y2],
                       Color = RGB::Red):
```

```
Cog1 := plot::Rotate2d(-a, [x1,  y1], cog1, line1,
                       a = 0..2*PI, Frames = 180):
Cog2 := plot::Rotate2d(2*a, [x2, y2], cog2, line2,
                       a = 0..2*PI, Frames = 180):
plot(Cog1, Cog2, Scaling = Constrained):
```

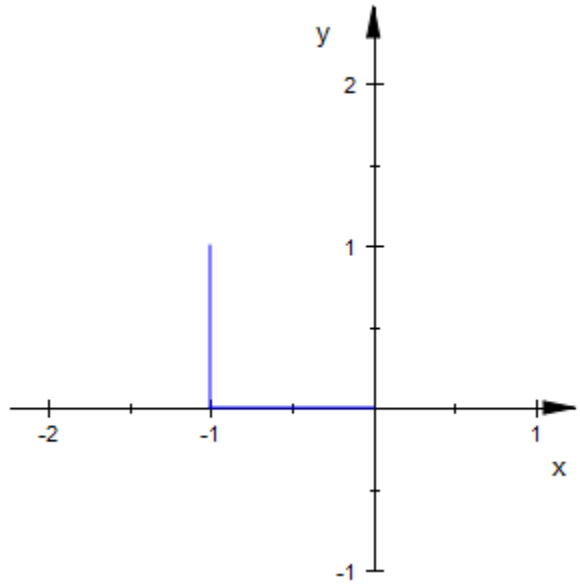



```
delete r1, x1, y1, r2, x2, y2, dr, cog1, cog2,
       line1, line2, Cog1, Cog2:
```

Example 4

We use an animated rotation inside another animated rotation:

```
L1 := plot::Line2d([0, 0], [0, 1]):
L2 := plot::Rotate2d(a, [0, 1], a = 0..2*PI,
                     plot::Line2d([0, 1], [1, 1])):
plot(plot::Rotate2d(a, [0, 0], L1, L2, a = 0..PI/2)):
```




```
delete L1, L2:
```

Parameters

`angle`

The rotation angle in radians: a numerical real value or an arithmetical expression of the animation parameter `a`.

angle is equivalent to the attribute `Angle`.

c_x, c_y, c_z

The components of the rotation center: numerical real values or arithmetical expressions of the animation parameter `a`. If no rotation center is specified, the center `[0, 0, 0]` is used.

c_x, c_y, c_z are equivalent to the attributes `Center`, `CenterX`, `CenterY`, `CenterZ`.

d_x, d_y, d_z

The components of the direction of the rotations axis: numerical real values or arithmetical expressions of the animation parameter `a`. If no direction is specified, the direction `[0, 0, 1]` is used.

d_x, d_y, d_z are equivalent to the attributes `Axis`, `AxisX`, `AxisY`, `AxisZ`.

obj_1, obj_2, …

Arbitrary plot objects of the appropriate dimension

a

Animation parameter, specified as $a = a_{min}..a_{max}$, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Rotate2d` | `plot::Scale2d` | `plot::Scale3d` | `plot::Transform2d` | `plot::Transform3d` | `plot::Translate2d` | `plot::Translate3d`

plot::Scale2d

Scaling of 2D objects

Syntax

```
plot::Scale2d([sx, sy], obj1, <obj2, …>, <a = amin .. amax>, options)
```

Description

`plot::Scale2d([sx, sy], objects)` applies the scaling transformation $x \to A\,x$ with the diagonal matrix $A = diag(s_x, s_y)$ to 2D objects.

Scale objects can scale several graphical objects simultaneously. Plotting the scale object renders all graphical objects inside.

Transformation objects can be used inside scale objects. If they are animated, the animations run simultaneously.

Animated scale objects are rather “cheap” concerning computing and storing costs. For more complex graphical objects, it is more efficient to use an animated scale object than to redefine the object for each frame.

The function `op` allows to extract the graphical objects inside a scale object.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Frames`	the number of frames in an animation	`50`
`Name`	the name of a plot object (for browser and legend)	

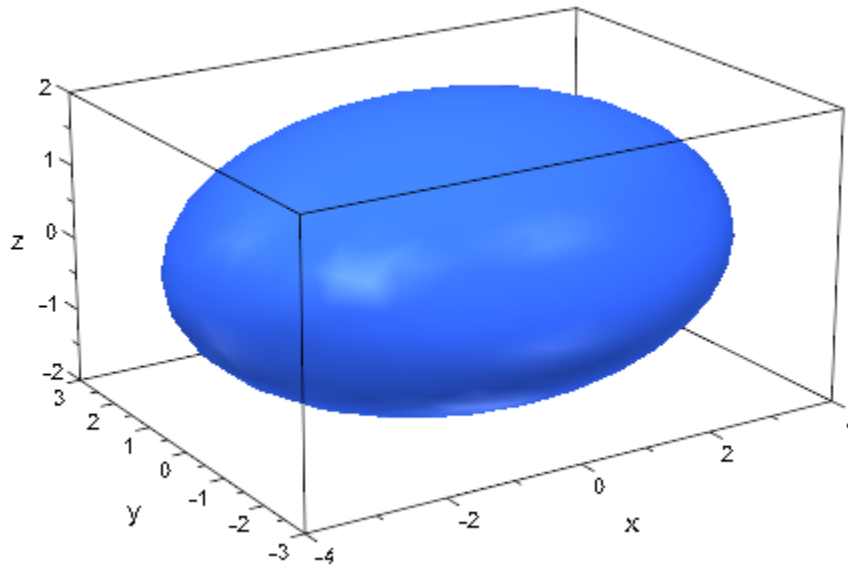
Attribute	Purpose	Default Value
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
Scale	scaling factors	[1, 1]
ScaleX	scaling factor in x-direction	1
ScaleY	scaling factor in y-direction	1
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0

Examples

Example 1

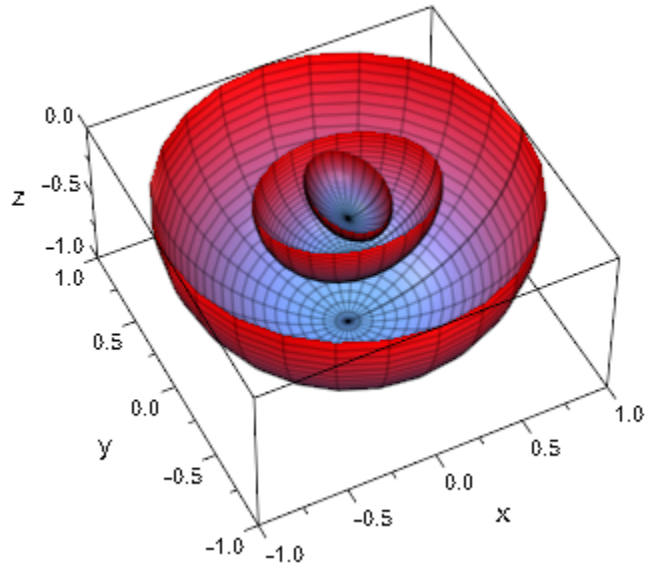
A scaling transformation turns a sphere into an ellipsoid:

```
plot(plot::Scale3d([1 + 3*a, 1 + 2*a, 1 + a],
                   plot::Sphere(1, [0, 0, 0]),
                   a = 0..1))
```



We plot a (southern) hemisphere and two scaled copies:

```
A0 := plot::Spherical([1, u, v], u = 0..2*PI, v = PI/2 .. PI):
A1 := plot::Scale3d([0.5, 0.4, 0.5], A0):
A2 := plot::Scale3d([0.2, 0.3, 0.2], A0):
plot(A0, A1, A2, CameraDirection = [-1, -2, 2.5]):
```




```
delete A0, A1, A2:
```

Parameters

s_x, s_y

The scaling factors: numerical real values or arithmetical expressions of the animation parameter `a`.

s_x, s_y are equivalent to the attributes `Scale`, `ScaleX`, `ScaleY`.

obj_1, obj_2, …

Arbitrary plot objects of the appropriate dimension

a

Animation parameter, specified as a = a_{min}..a_{max}, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Rotate2d` | `plot::Rotate3d` | `plot::Scale3d` | `plot::Transform2d` | `plot::Transform3d` | `plot::Translate2d` | `plot::Translate3d`

plot::Scale3d

Scaling of 3D objects

Syntax

```
plot::Scale3d([sx, sy, sz], obj1, <obj2, …>, <a = amin .. amax>, options)
```

Description

`plot::Scale3d([sx, sy, sz], objects)` applies the scaling transformation $x \to A\,x$ with the diagonal matrix $A = diag(s_x, s_y, s_z)$ to 3D objects.

Scale objects can scale several graphical objects simultaneously. Plotting the scale object renders all graphical objects inside.

Transformation objects can be used inside scale objects. If they are animated, the animations run simultaneously.

Animated scale objects are rather “cheap” concerning computing and storing costs. For more complex graphical objects, it is more efficient to use an animated scale object than to redefine the object for each frame.

The function `op` allows to extract the graphical objects inside a scale object.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Frames`	the number of frames in an animation	`50`
`Name`	the name of a plot object (for browser and legend)	

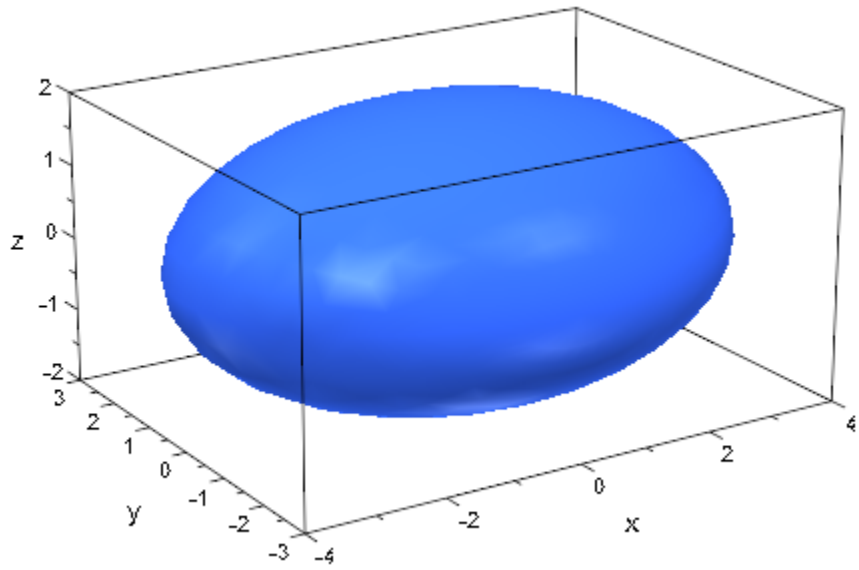
Attribute	Purpose	Default Value
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`Scale`	scaling factors	[1, 1, 1]
`ScaleX`	scaling factor in x-direction	1
`ScaleY`	scaling factor in y-direction	1
`ScaleZ`	scaling factor in z-direction	1
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`

Examples

Example 1

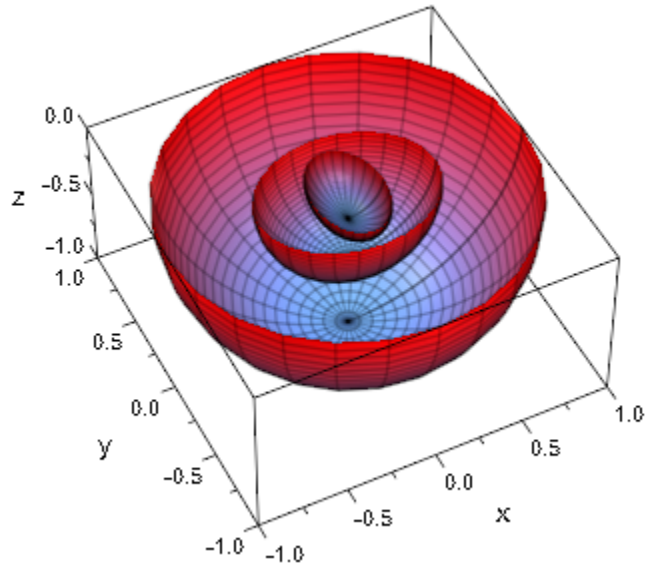
A scaling transformation turns a sphere into an ellipsoid:

```
plot(plot::Scale3d([1 + 3*a, 1 + 2*a, 1 + a],
                   plot::Sphere(1, [0, 0, 0]),
                   a = 0..1))
```



We plot a (southern) hemisphere and two scaled copies:

```
A0 := plot::Spherical([1, u, v], u = 0..2*PI, v = PI/2 .. PI):
A1 := plot::Scale3d([0.5, 0.4, 0.5], A0):
A2 := plot::Scale3d([0.2, 0.3, 0.2], A0):
plot(A0, A1, A2, CameraDirection = [-1, -2, 2.5]):
```




```
delete A0, A1, A2:
```

Parameters

s_x, s_y, s_z

The scaling factors: numerical real values or arithmetical expressions of the animation parameter `a`.

s_x, s_y, s_z are equivalent to the attributes `Scale`, `ScaleX`, `ScaleY`, `ScaleZ`.

obj_1, obj_2, …

Arbitrary plot objects of the appropriate dimension

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Rotate2d` | `plot::Rotate3d` | `plot::Scale2d` | `plot::Transform2d` | `plot::Transform3d` | `plot::Translate2d` | `plot::Translate3d`

plot::Transform2d

Affine linear transformation of 2D objects

Syntax

```
plot::Transform2d(<b2d>, A2d, obj1, <obj2, …>, <a = amin .. amax>, options)
```

Description

`plot::Transform2d(b, A, objects)` with a vector b and a matrix A applies the affine linear transformation $x \to A\,x + b$ to 2D objects.

The transformation matrix `A` can be specified by a list of lists

```
[[A1, 1, A1, 2, …], [A2, 1, A2, 2, …], …]
```

with the sublists representing the rows.

A plain list

```
[A1, 1, A1, 2, A2, 1, A2, 2]
```

represents the matrix row by row in 2D.

Transform objects can transform several graphical objects simultaneously. Plotting the transform object renders all graphical objects inside.

Transformed objects have a tendency to overestimate their `ViewingBox`. Cf. the help page of `ViewingBox`. In such a case, you should specify a suitable `ViewingBox` explicitly.

Transformation objects can be used inside transformation objects. If they are animated, the animations run simultaneously.

Animated transform objects are rather “cheap” concerning computing and storing costs. For more complex graphical objects, it is more efficient to use an animated transform object than to redefine the object for each frame.

The function `op` allows to extract the graphical objects inside a transformation object.

Attributes

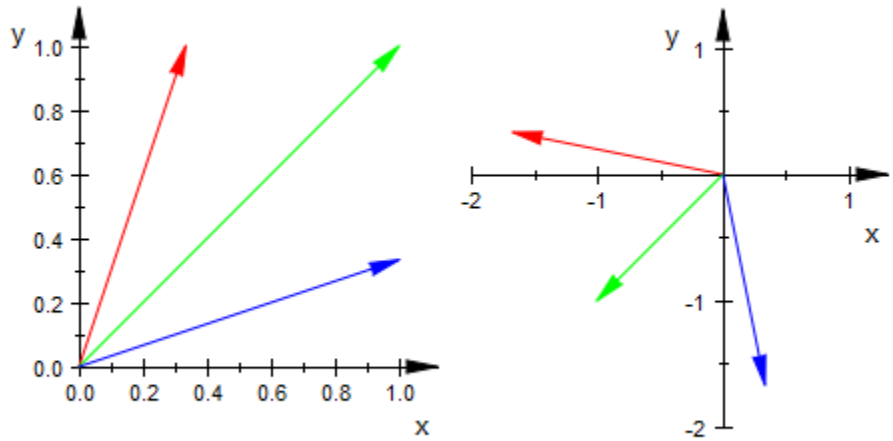
Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Frames`	the number of frames in an animation	`50`
`Matrix2d`	transformation matrices	[1, 0, 0, 1]
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`Shift`	shift vector	[0, 0]
`ShiftX`	shift vector	`0`
`ShiftY`	shift vector	`0`
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`

Examples

Example 1

We visualize a linear transformation $x \rightarrow A\,x$ without shift:

```
x1 := plot::Arrow2d([0, 0], [1/3, 1], Color = RGB::Red):
x2 := plot::Arrow2d([0, 0], [1, 1], Color = RGB::Green):
x3 := plot::Arrow2d([0, 0], [1, 1/3], Color = RGB::Blue):
A :=  matrix([[1, -2], [-2, 1]]):
plot(plot::Scene2d(x1, x2, x3),
     plot::Scene2d(plot::Transform2d(A, x1, x2, x3)),
     Scaling = Constrained, Layout = Horizontal):
```




```
delete x1, x2, x3, A:
```

Example 2

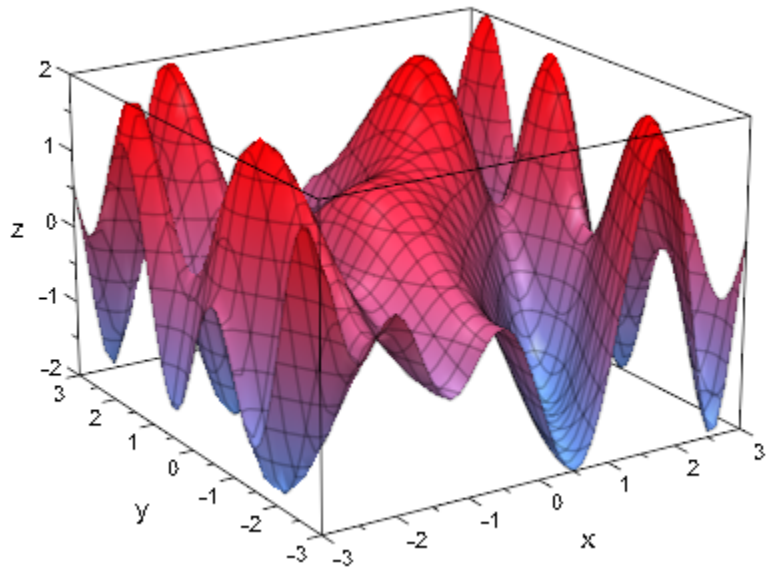
For some applications, it is very popular to plot a function in 3D together with a projection of its contour lines onto the lower or upper bounding plane. MuPAD has no direct option for this, but with `plot::Transform3d`, it is possible to achieve the same effect. Assume you have the function under consideration in a `plot::Function3d` object:

```
f := plot::Function3d(sin(x*y)+cos(x^2-y),
```

```
                    x=-3..3, y=-3..3, Submesh=[1,1]):
```

To plot contour lines at all, we use the attribute `ZContours`. Since we don't want to change our `f`, we create a modified *copy* using `plot::modify`:

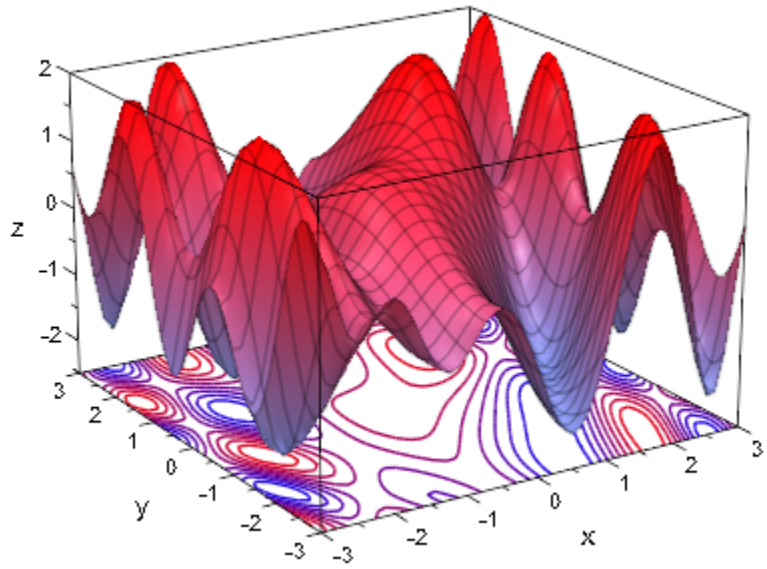
```
plot(plot::modify(f, ZContours = [Automatic, 10]))
```



To only get contour lines, we have to change a few more parameters: We need to switch off the surface and the parameter lines. Then, we add height coloring to our lines and use `plot::Transform3d` to *project* them onto the plane $z = -2.5$. Finally, we plot these lines together with the original function:

```
plot(f,
     plot::Transform3d([0, 0, -2.5], // shift vector
                       [1, 0, 0,  // transformation matrix
                        0, 1, 0,
                        0, 0, 0],
               plot::modify(f,
                      Filled = FALSE,
                      XLinesVisible = FALSE, YLinesVisible = FALSE,
                      ZContours = [Automatic, 10],
                      LineColorFunction =  // height coloring
```

```
((x, y, z) -> [(z+2)/4, 0, (2-z)/4])))
```



Parameters

b_{2d}

The 2D shift vector: a list with 2 entries. Also vectors generated by `matrix` and arrays are accepted. The entries must be numerical values or arithmetical expressions of the animation paramater `a`.

b_{2d} is equivalent to the attribute `Shift`.

A_{2d}

The 2D transformation matrix: a 2×2 matrix, a 2×2 array, a list of 2 lists, or a plain list with 4 entries. The entries must be numerical values or arithmetical expressions of the animation paramater `a`.

A_{2d} is equivalent to the attribute `Matrix2d`.

obj1, obj2, …

Plot objects of the appropriate dimension

a

Animation parameter, specified as a = $a_{min}..a_{max}$, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Rotate2d` | `plot::Rotate3d` | `plot::Scale2d` | `plot::Scale3d` | `plot::Transform3d` | `plot::Translate2d` | `plot::Translate3d`

plot::Transform3d

Affine linear transformation of 3D objects

Syntax

`plot::Transform3d(<`b_{3d}`>, `A_{3d}`, `obj_1`, <`obj_2`, …>, <a = `a_{min}` .. `a_{max}`>, options)`

Description

`plot::Transform3d(b, A, objects)` transforms 3D objects accordingly.

The transformation matrix `A` may be specified by a list of lists

`[[`$A_{1,1}$`, `$A_{1,2}$`, …], [`$A_{2,1}$`, `$A_{2,2}$`, …], …]`

with the sublists representing the rows.

A plain list

`[`$A_{1,1}$`, `$A_{1,2}$`, …, `$A_{3,2}$`, `$A_{3,3}$`]`

represents the matrix row by row in 3D.

Transform objects can transform several graphical objects simultaneously. Plotting the transform object renders all graphical objects inside.

Transformed objects have a tendency to overestimate their `ViewingBox`. Cf. the help page of `ViewingBox`. In such a case, you should specify a suitable `ViewingBox` explicitly.

Transformation objects can be used inside transformation objects. If they are animated, the animations run simultaneously.

Animated transform objects are rather “cheap” concerning computing and storing costs. For more complex graphical objects, it is more efficient to use an animated transform object than to redefine the object for each frame.

The function `op` allows to extract the graphical objects inside a transformation object.

Attributes

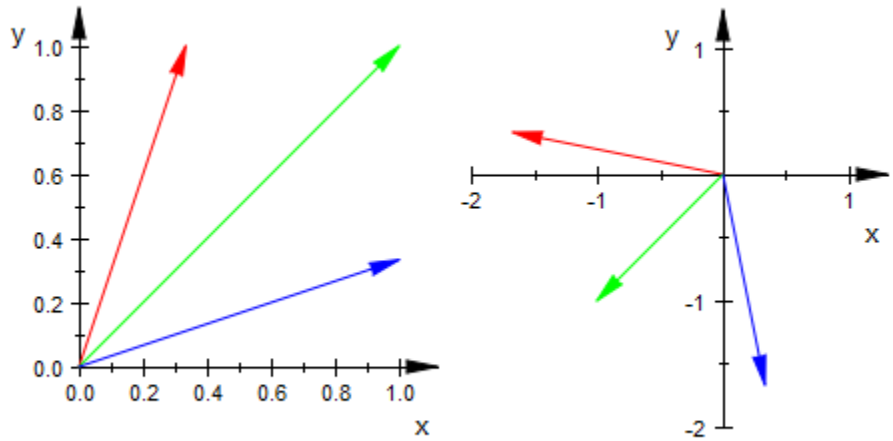
Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Frames`	the number of frames in an animation	`50`
`Matrix3d`	transformation matrices	[1, 0, 0, 0, 1, 0, 0, 0, 1]
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`Shift`	shift vector	[0, 0, 0]
`ShiftX`	shift vector	`0`
`ShiftY`	shift vector	`0`
`ShiftZ`	shift vector	`0`
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`

Examples

Example 1

We visualize a linear transformation $x \rightarrow A\,x$ without shift:

```
x1 := plot::Arrow2d([0, 0], [1/3, 1], Color = RGB::Red):
x2 := plot::Arrow2d([0, 0], [1, 1], Color = RGB::Green):
x3 := plot::Arrow2d([0, 0], [1, 1/3], Color = RGB::Blue):
A :=  matrix([[1, -2], [-2, 1]]):
plot(plot::Scene2d(x1, x2, x3),
     plot::Scene2d(plot::Transform2d(A, x1, x2, x3)),
     Scaling = Constrained, Layout = Horizontal):
```




```
delete x1, x2, x3, A:
```

Example 2

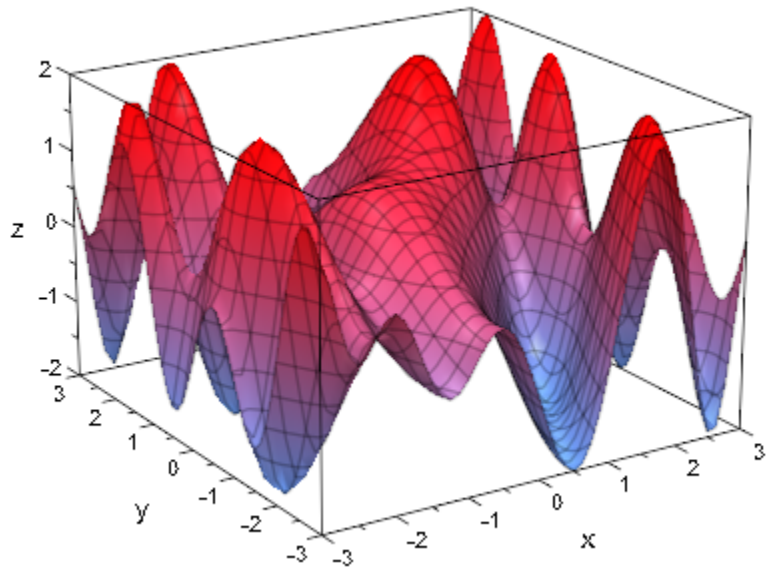
For some applications, it is very popular to plot a function in 3D together with a projection of its contour lines onto the lower or upper bounding plane. MuPAD has no direct option for this, but with `plot::Transform3d`, it is possible to achieve the same effect. Assume you have the function under consideration in a `plot::Function3d` object:

```
f := plot::Function3d(sin(x*y)+cos(x^2-y),
```

```
                    x=-3..3, y=-3..3, Submesh=[1,1]):
```

To plot contour lines at all, we use the attribute `ZContours`. Since we don't want to change our `f`, we create a modified *copy* using `plot::modify`:

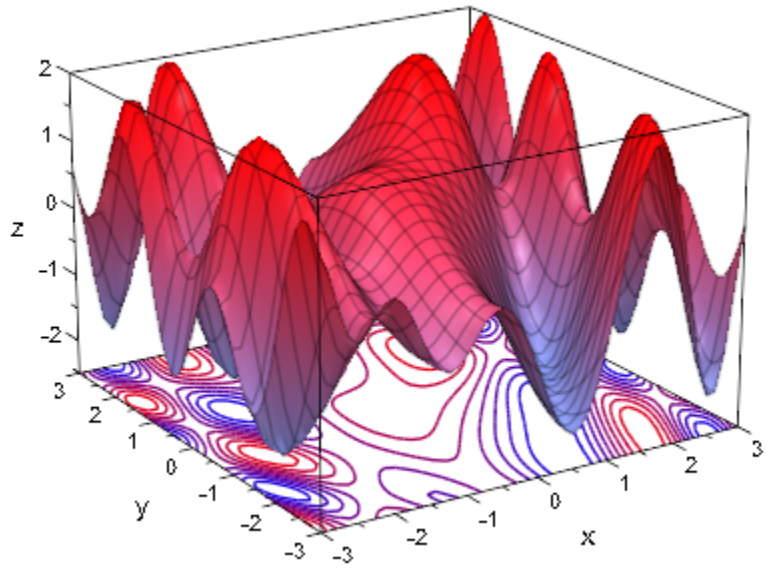
```
plot(plot::modify(f, ZContours = [Automatic, 10]))
```



To only get contour lines, we have to change a few more parameters: We need to switch off the surface and the parameter lines. Then, we add height coloring to our lines and use `plot::Transform3d` to *project* them onto the plane $z = -2.5$. Finally, we plot these lines together with the original function:

```
plot(f,
     plot::Transform3d([0, 0, -2.5], // shift vector
                       [1, 0, 0,  // transformation matrix
                        0, 1, 0,
                        0, 0, 0],
               plot::modify(f,
                      Filled = FALSE,
                      XLinesVisible = FALSE, YLinesVisible = FALSE,
                      ZContours = [Automatic, 10],
                      LineColorFunction =  // height coloring
```

```
((x, y, z) -> [(z+2)/4, 0, (2-z)/4])))
```



Parameters

b_{3d}

The 3D shift vector: a list with 3 entries. Also vectors generated by `matrix` or arrays are accepted. The entries must be numerical values or arithmetical expressions of the animation paramater `a`.

b_{3d} is equivalent to the attribute `Shift`.

A_{3d}

The 3D transformation matrix: a 3×3 matrix, a 3×3 array, a list of 3 lists, or a plain list with 9 entries. The entries must be numerical values or arithmetical expressions of the animation paramater `a`.

A_{3d} is equivalent to the attribute `Matrix3d`.

obj1, obj2, …

Plot objects of the appropriate dimension

a

Animation parameter, specified as a = $a_{min}..a_{max}$, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Rotate2d` | `plot::Rotate3d` | `plot::Scale2d` | `plot::Scale3d` | `plot::Transform2d` | `plot::Translate2d` | `plot::Translate3d`

plot::Translate2d

Translation of 2D objects

Syntax

```
plot::Translate2d([dx, dy], obj1, <obj2, …>, <a = amin .. amax>, options)
```

Description

`plot::Translate2d([dx, dy], object)` shifts a 2D object by d_x units along the x-axis and d_y units along the y-axis.

Translate objects can translate several graphical objects simultaneously. Plotting the translate object renders all graphical objects inside.

Transformation objects can be used inside translation objects. If they are animated, the animations run simultaneously.

Animated translate objects are rather "cheap" concerning computing and storing costs. For more complex graphical objects, it is more efficient to use an animated translate object than to redefine the object for each frame.

The function `op` allows to extract the graphical objects inside a translate object.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Frames`	the number of frames in an animation	`50`
`Name`	the name of a plot object (for browser and legend)	

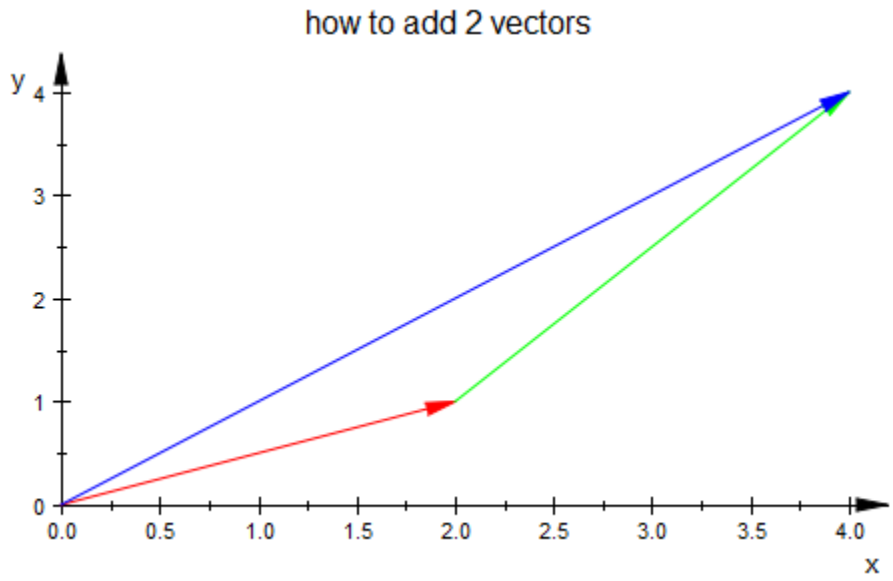
Attribute	Purpose	Default Value
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
Shift	shift vector	[0, 0]
ShiftX	shift vector	0
ShiftY	shift vector	0
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0

Examples

Example 1

We use an animated translation object to shift a vector to the tip of another vector:

```
A1 := plot::Arrow2d([0, 0], [2, 1], Color = RGB::Red):
A2 := plot::Arrow2d([0, 0], [2, 3], Color = RGB::Green):
plot(A1, plot::Translate2d([2*a, a], A2, a = 0..1,
                           TimeRange = 0..4),
     plot::Arrow2d([0, 0], [4, 4], Color = RGB::Blue,
                   VisibleFromTo = 4..6),
     Header = "how to add 2 vectors"):
```




```
delete A1, A2:
```

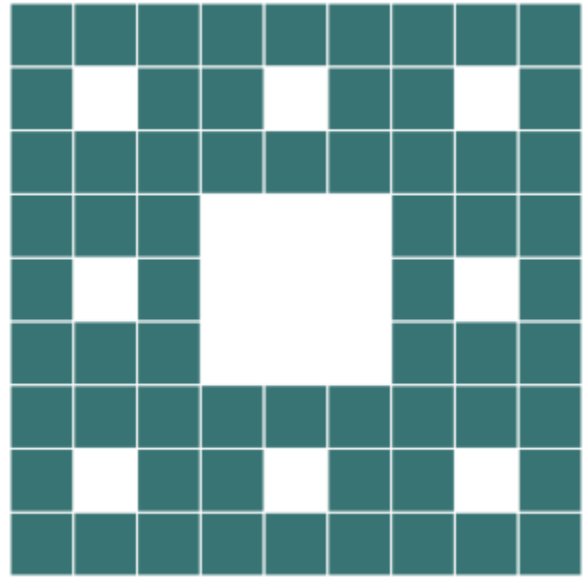
Example 2

Note that `plot::Translate2d` and `plot::Translate3d` do not actually change the objects translated, so it is possible to use the same object in different places in the same plot. As an example, we show an intuitive way of constructing the Sierpinski carpet, a flat version of the Menger sponge.

The Sierpinski carpet is a fractal with the general shape of a square and the property that the following operation maps it onto itself: Take eight copies, scale them by $\frac{1}{3}$, and arrange them in a square with the middle left unfilled. Now, this can be directly written in MuPAD code:

```
Carpet := proc(iter)
  local square;
begin
  if iter <= 1 then
    return(plot::Polygon2d([[0,0], [0,1], [1,1], [1,0]],
```

```
                          Closed, Filled, FillPattern = Solid,
                          FillColor = [0.2234, 0.4563, 0.4568],
                          LinesVisible, LineColor = RGB::White,
                          Scaling = Constrained, Axes = None));
  else
    square := plot::Scale2d([1/3, 1/3], Carpet(iter-1));
    return(plot::Group2d(
                plot::Translate2d([  0,   0], square),
                plot::Translate2d([  0, 1/3], square),
                plot::Translate2d([  0, 2/3], square),
                plot::Translate2d([1/3,   0], square),
              //   plot::Translate2d([1/3, 1/3], square),
                plot::Translate2d([1/3, 2/3], square),
                plot::Translate2d([2/3,   0], square),
                plot::Translate2d([2/3, 1/3], square),
                plot::Translate2d([2/3, 2/3], square)));
  end_if;
end_proc:

plot(Carpet(3))
```



Parameters

d_x, d_y

The components of the shift vector: numerical real values or arithmetical expressions of the animation parameter `a`.

d_x, d_y are equivalent to the attribute `Shift`.

obj_1, obj_2, …

Arbitrary plot objects of the appropriate dimension

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Rotate2d` | `plot::Rotate3d` | `plot::Scale2d` | `plot::Scale3d` | `plot::Transform2d` | `plot::Transform3d` | `plot::Translate3d`

plot::Translate3d

Translation of 3D objects

Syntax

`plot::Translate3d([`d_x`, `d_y`, `d_z`], `obj_1`, <`obj_2`, …>, <a = `a_{min}` .. `a_{max}`>, options)`

Description

`plot::Translate3d([`d_x`, `d_y`, `d_z`], object)` shifts a 3D object.

Translate objects can translate several graphical objects simultaneously. Plotting the translate object renders all graphical objects inside.

Transformation objects can be used inside translation objects. If they are animated, the animations run simultaneously.

Animated translate objects are rather “cheap” concerning computing and storing costs. For more complex graphical objects, it is more efficient to use an animated translate object than to redefine the object for each frame.

The function `op` allows to extract the graphical objects inside a translate object.

Attributes

Attribute	Purpose	Default Value
`AffectViewingBox`	influence of objects on the `ViewingBox` of a scene	`TRUE`
`Frames`	the number of frames in an animation	`50`
`Name`	the name of a plot object (for browser and legend)	

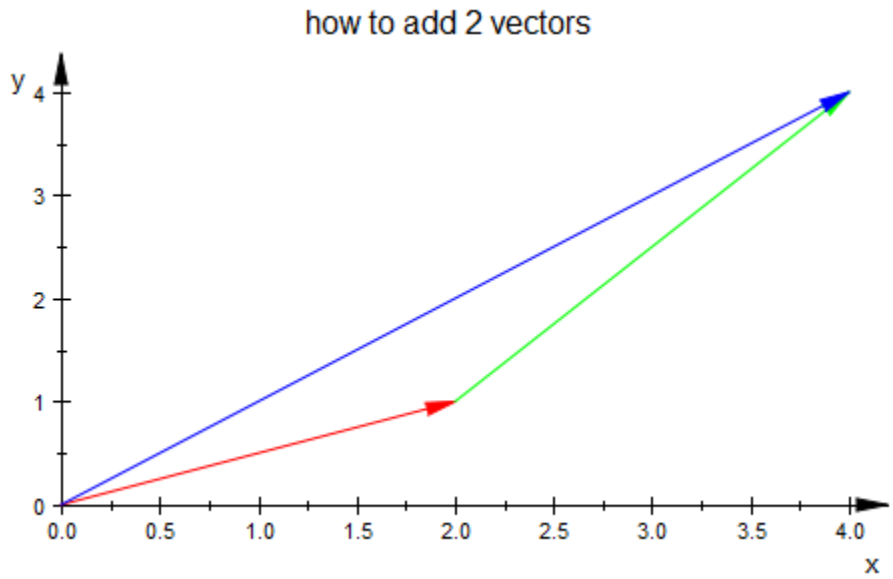
Attribute	Purpose	Default Value
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
Shift	shift vector	[0, 0, 0]
ShiftX	shift vector	0
ShiftY	shift vector	0
ShiftZ	shift vector	0
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0

Examples

Example 1

We use an animated translation object to shift a vector to the tip of another vector:

```
A1 := plot::Arrow2d([0, 0], [2, 1], Color = RGB::Red):
A2 := plot::Arrow2d([0, 0], [2, 3], Color = RGB::Green):
plot(A1, plot::Translate2d([2*a, a], A2, a = 0..1,
                           TimeRange = 0..4),
     plot::Arrow2d([0, 0], [4, 4], Color = RGB::Blue,
                   VisibleFromTo = 4..6),
     Header = "how to add 2 vectors"):
```




```
delete A1, A2:
```

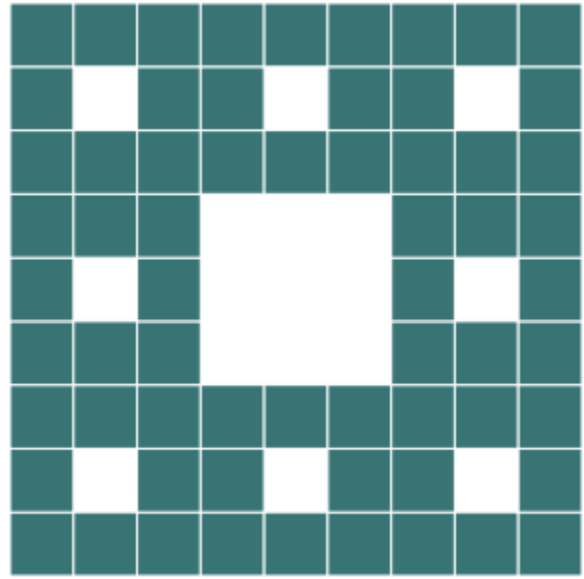
Example 2

Note that `plot::Translate2d` and `plot::Translate3d` do not actually change the objects translated, so it is possible to use the same object in different places in the same plot. As an example, we show an intuitive way of constructing the Sierpinski carpet, a flat version of the Menger sponge.

The Sierpinski carpet is a fractal with the general shape of a square and the property that the following operation maps it onto itself: Take eight copies, scale them by $\frac{1}{3}$, and arrange them in a square with the middle left unfilled. Now, this can be directly written in MuPAD code:

```
Carpet := proc(iter)
  local square;
begin
  if iter <= 1 then
    return(plot::Polygon2d([[0,0], [0,1], [1,1], [1,0]],
```

```
                          Closed, Filled, FillPattern = Solid,
                          FillColor = [0.2234, 0.4563, 0.4568],
                          LinesVisible, LineColor = RGB::White,
                          Scaling = Constrained, Axes = None));
  else
    square := plot::Scale2d([1/3, 1/3], Carpet(iter-1));
    return(plot::Group2d(
                plot::Translate2d([  0,   0], square),
                plot::Translate2d([  0, 1/3], square),
                plot::Translate2d([  0, 2/3], square),
                plot::Translate2d([1/3,   0], square),
              //   plot::Translate2d([1/3, 1/3], square),
                plot::Translate2d([1/3, 2/3], square),
                plot::Translate2d([2/3,   0], square),
                plot::Translate2d([2/3, 1/3], square),
                plot::Translate2d([2/3, 2/3], square)));
  end_if;
end_proc:

plot(Carpet(3))
```



Parameters

d_x, d_y, d_z

The components of the shift vector: numerical real values or arithmetical expressions of the animation parameter `a`.

d_x, d_y, d_z are equivalent to the attribute `Shift`.

obj_1, obj_2, …

Arbitrary plot objects of the appropriate dimension

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::Rotate2d` | `plot::Rotate3d` | `plot::Scale2d` | `plot::Scale3d` | `plot::Transform2d` | `plot::Transform3d` | `plot::Translate2d`

plot::AmbientLight

Ambient light

Syntax

```
plot::AmbientLight(<intensity>, <a = a_min .. a_max>, options)
```

Description

`plot::AmbientLight(intensity)` generates undirected ambient light of the given intensity.

Each 3D scene is illuminated by several light sources that are set automatically and cannot be controlled by the user. Cf. the help page of `Lighting`.

If special light effects are desired, the user can create alternative light sources of various types such as `plot::AmbientLight`, `plot::DistantLight`, `plot::PointLight`, and `plot::SpotLight`.

If at least one user defined light source is inserted into the scene (e.g., by simply passing the light objects as input parameters to the `plot` command), the automatic lights are switched off and the user defined lights are used to illuminate the scene.

While directed lights such as `plot::DistantLight` etc. create shading effects that add depth to the picture, a certain amount of undirected ambient light is usually needed.

`plot::AmbientLight(intensity)` creates ambient light whose intensity is given by the parameter `intensity`. When the intensity is 1, the ambient light dominates all other light sources.

By default, white light is created. Other colours can be chosen by the attribute `LightColor`.

It does not make sense to have more than one ambient light object in a scene.

Attributes

Attribute	Purpose	Default Value
`Frames`	the number of frames in an animation	`50`
`LightColor`	the color of light	`RGB::White`
`LightIntensity`	intensity of light	`1.0`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`
`Visible`	visibility	`TRUE`

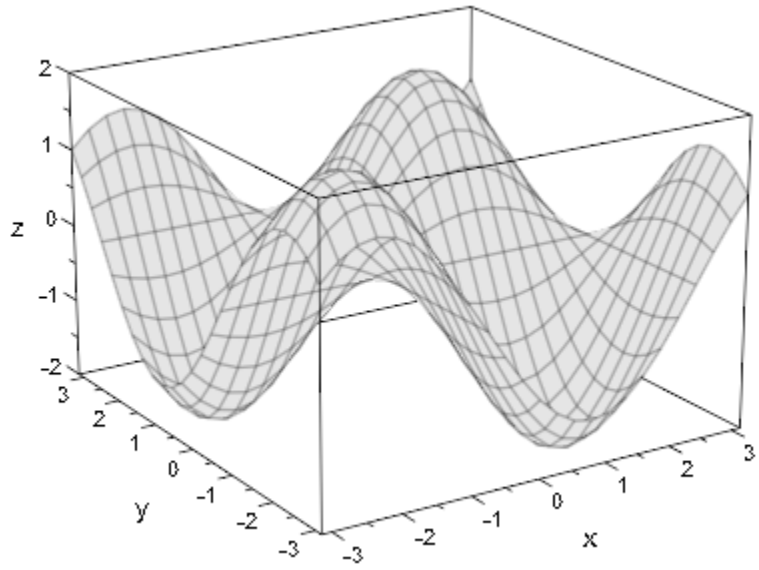
Examples

Example 1

We create a 3D function graph in flat white and use ambient white light to illuminate it:

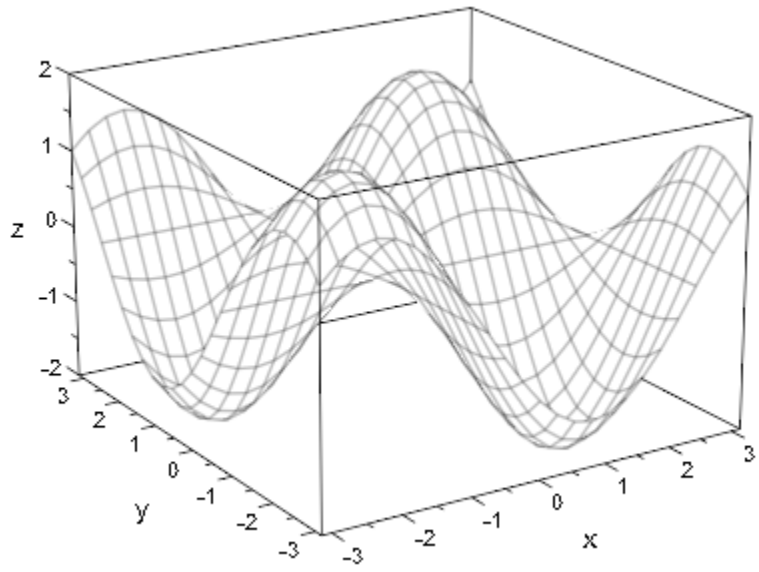
```
f := plot::Function3d(sin(x + y) + cos(x - y),
                      x = -PI..PI, y = -PI..PI,
                      FillColorType = Flat,
```

```
                        Color = RGB::White):
ambientlight := plot::AmbientLight(0.7):
plot(f, ambientlight):
```



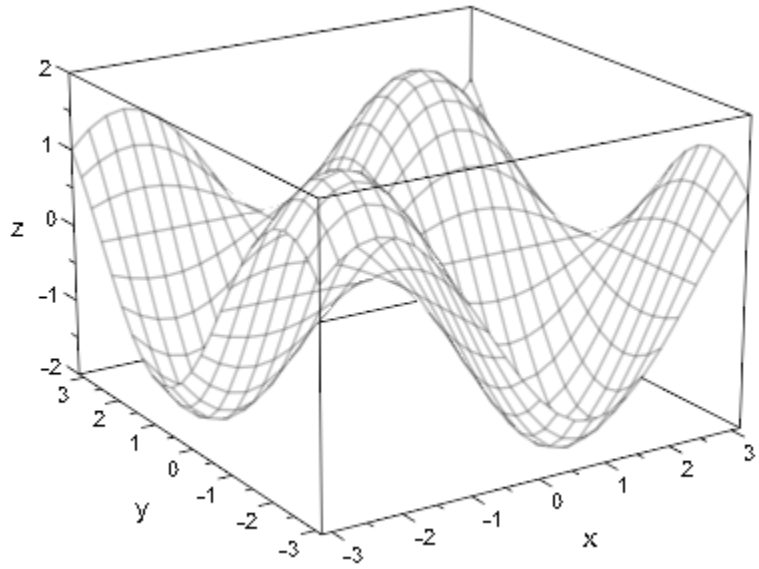
We create another ambient light with animated intensity:

```
ambientlight := plot::AmbientLight(a, a = 0..1):
plot(f, ambientlight)
```



We add directed yellow light to the scene and study the mixture between the “sunlight” and an increasing amount of ambient light. When the ambient light is at full intensity, it dominates the directed light completely:

```
sunlight :=  plot::DistantLight([0, 0, 0], [5, 1, -3], 1,
                                LightColor = RGB::Yellow):
plot(f, ambientlight, sunlight)
```




```
delete f, ambientlight, sunlight:
```

Parameters

intensity

The intensity of the light: a numerical value between 0 and 1 or an arithmetical expression of the animation parameter `a`.

intensity is equivalent to the attribute `LightIntensity`.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions

`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::DistantLight` | `plot::PointLight` | `plot::SpotLight`

plot::Camera

Camera

Syntax

```
plot::Camera([p_x, p_y, p_z], [f_x, f_y, f_z], angle, <a = a_min .. a_max>, options)
```

Description

`plot::Camera([p_x, p_y, p_z], [f_x, f_y, f_z], angle)` creates a camera at the position `[p_x, p_y, p_z]` pointing towards the focal point `[f_x, f_y, f_z]`. The opening angle of its lense is given by `angle`.

When creating a 3D scene, an "automatic camera" is used. Its location can be controlled by the attribute `CameraDirection`, but there are now further means of manipulating its parameters.

If the automatic camera does not suffice for your purposes, you may define your own camera by `plot::Camera`. Inserting such a camera object in your scene (for example, just by passing the camera as an argument to the `plot` command), the automatic camera is switched off and the new camera determines the view.

A camera of type `plot::Camera` allows to set all parameters determining the view and its perspective. Further, all parameters can be animated such that an animated "flight" through a 3D scene can be realized.

The first argument `[p_x, p_y, p_z]` in the call generating a camera is the `Position` of the camera in 3-space. The second argument `[f_x, f_y, f_z]` is the point the camera is aimed at (`FocalPoint`).

The optical axis is given by the vector `FocalPoint - Position`.

Together with the opening angle of the zoom lense (`ViewingAngle`), these parameters determine the view of the scene.

The `FocalPoint` vector can be replaced by any other point on the optical axes without changing the view. (`FocalPoint` and `Position` should not coincide, though.)

By default, the *z*-direction in 3-space corresponds to the vertical direction of the final picture. If this is not desired, the camera can be rotated around its optical axes using the attribute `UpVector`.

Depending on the distance of the camera to the graphical scene and the opening angle of the lense, the scene may fill only a small portion of the viewing area if the camera is too far away. If the camera is too close, only some parts of the scene may be visible

Just as for a real camera, you will have to move closer to or farther away from the scene to make it fill the drawing area as desired. Alternatively, you may keep the camera position fixed and use the zoom lense by choosing an appropriate `ViewingAngle`.

As in real life, you have to find appropriate parameters experimentally by looking at the picture and changing the parameters interactively.

Alternatively, you may define the camera with the attribute `OrthogonalProjection = TRUE`. This has the same effect as positioning the camera at a large distance from the scene using a powerful tele lense.

In this case, the camera ignores the `ViewingAngle` and the `Position` in 3-space. It is moved along the optical axis `FocalPoint - Position` to infinity and chooses an infinitesimal small viewing angle such that the scene fills the drawing area optimally.

Several cameras can be present simultaneously in a graphical scene. The first camera specified in the `plot` command determines the views.

One may switch between the cameras by clicking on the corresponding camera in the interactive “object browser” of the MuPAD graphics tool (see section Viewer, Browser, and Inspector: Interactive Manipulation of this document).

You may place your own light sources in the scene. When specifying the attribute `CameraCoordinates = TRUE` in the definition of the lights, they are attached to the camera and move automatically, when the camera is moved.

Attributes

Attribute	Purpose	Default Value
`FocalPoint`	the focal point of a camera	
`FocalPointX`	the focal point of a camera, x-coordinate	

Attribute	Purpose	Default Value
FocalPointY	the focal point of a camera, y-coordinate	
FocalPointZ	the focal point of a camera, z-coordinate	
Frames	the number of frames in an animation	50
KeepUpVector	keep the UpVector constant when moving the camera?	TRUE
Name	the name of a plot object (for browser and legend)	
OrthogonalProjection	parallel projection without perspective distortion	FALSE
ParameterEnd	end value of the animation parameter	
ParameterName	name of the animation parameter	
ParameterBegin	initial value of the animation parameter	
ParameterRange	range of the animation parameter	
Position	positions of cameras, lights, and text objects	
PositionX	x-positions of cameras, lights, and text objects	
PositionY	y-positions of cameras, lights, and text objects	
PositionZ	z-positions of cameras, lights, and text objects	
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0

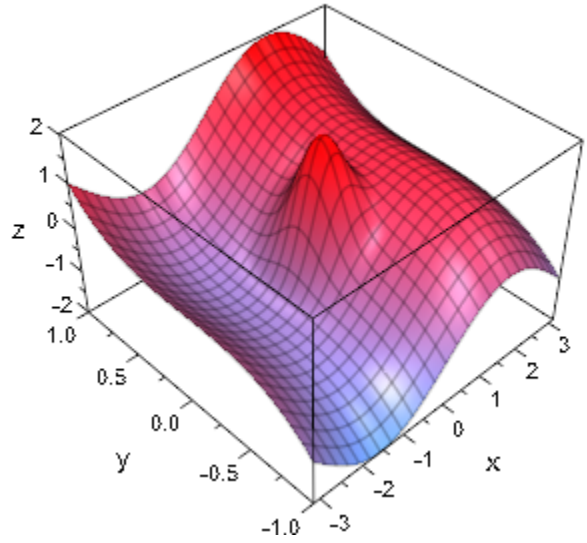
Attribute	Purpose	Default Value
`UpVector`	"up direction" of a camera	[0.0, 0.0, 1.0]
`UpVectorX`	x-component of the "up vector" of the camera	0.0
`UpVectorY`	y-component of the "up vector" of the camera	0.0
`UpVectorZ`	z-component of the "up vector" of the camera	1.0
`ViewingAngle`	opening angle of the camera lense	
`Visible`	visibility	TRUE

Examples

Example 1

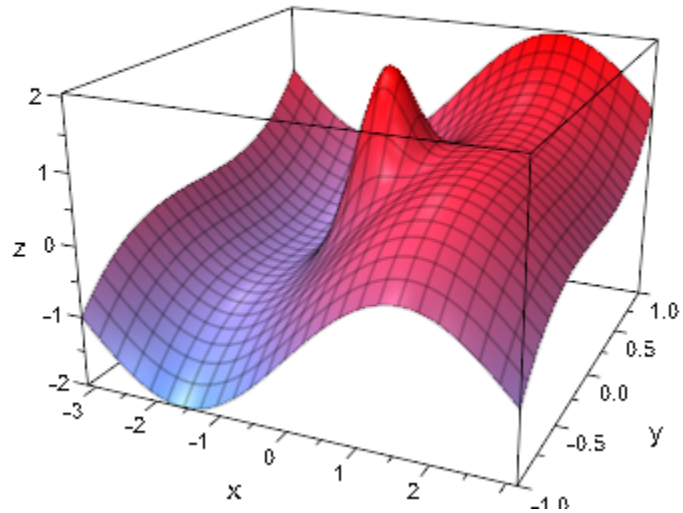
We use our own camera to view the 3D graph of a function:

```
f := plot::Function3d(sin(x) + y^3 + 2*exp(-3*x^2 - 20*y^2),
                      x = -PI..PI, y = -1 .. 1,
                      Submesh = [2, 2]):
camera := plot::Camera([-12, -4, 14], [0, 0, 0], PI/7):
plot(f, camera):
```



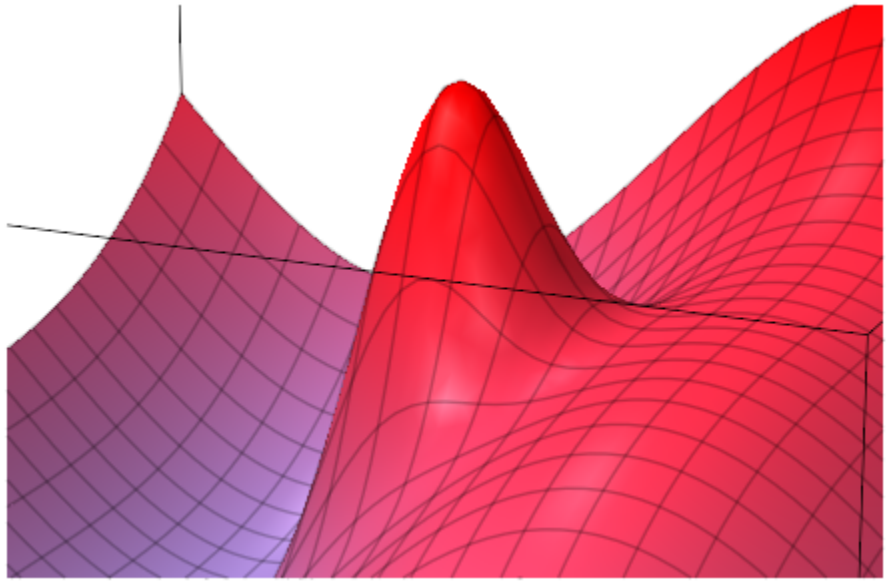
We move the camera to another position:

```
camera::Position := [7, -5, 6]:
plot(f, camera):
```



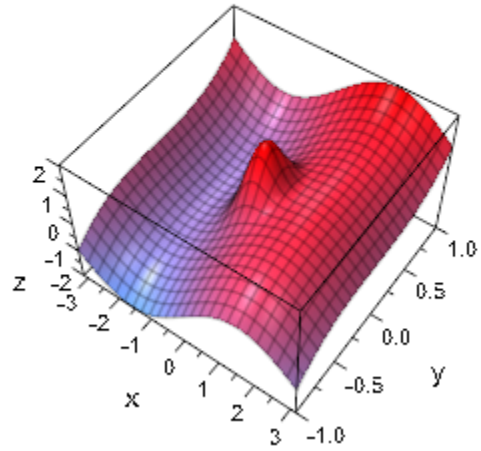
We turn the camera towards the central peak and zoom in by decreasing the openening angle of the zoom lense:

```
camera::FocalPoint := [0, 0, 1]:
camera::ViewingAngle := PI/20:
plot(f, camera):
```



We create an animated camera and fly through the scene:

```
camera := plot::Camera([-15 + 3*a, 4 - a, 3 +  (a - 4)^2],
                       [0, 0, 1.5], PI/6, a = 0..8,
                       Frames = 100, TimeRange = 0..20):
plot(f, camera):
```




```
delete f, camera:
```

Parameters

p_x, p_y, p_z

Coordinates of the camera position: numerical real values or arithmetical expressions of the animation parameter `a`.

p_x, p_y, p_z are equivalent to the attributes `PositionX`, `PositionY`, `PositionZ`.

f_x, f_y, f_z

Coordinates of the the focal point: numerical real values or arithmetical expressions of the animation parameter `a`.

f_x, f_y, f_z are equivalent to the attributes `FocalPointX`, `FocalPointY`, `FocalPointZ`.

angle

The opening angle of the lense in radians: a numerical real value or an arithmetical expression of the animation parameter `a` representing a value between 0 and π.

angle is equivalent to the attribute `ViewingAngle`.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions

`CameraCoordinates` | `CameraDirection` | `FocalPoint` | `KeepUpVector` | `OrthogonalProjection` | `plot` | `plot::copy` | `Position` | `UpVector` | `ViewingAngle`

More About

- "Cameras in 3D"

plot::DistantLight

Directed distant light (“sunlight”)

Syntax

```
plot::DistantLight([p_x, p_y, p_z], [t_x, t_y, t_z], <intensity>, <a = a_min .. a_max>, options)
```

Description

`plot::DistantLight([p_x, p_y, p_z], [t_x, t_y, t_z], intensity)` creates a distant light source emitting parallel light shining into the direction `[t_x - p_x, t_y - p_y, t_z - p_z]`

Each 3D scene is illuminated by several light sources that are set automatically and cannot be controlled by the user. Cf. the help page of `Lighting`.

If special light effects are desired, the user can create alternative light sources of various types such as `plot::AmbientLight`, `plot::DistantLight`, `plot::PointLight`, and `plot::SpotLight`.

If at least one user defined light source is inserted into the scene (e.g., by simply passing the light objects as input parameters to the `plot` command), the automatic lights are switched off and the user defined lights are used to illuminate the scene.

Note: The vector `[p_x, p_y, p_z]` does *not* represent the position of a distant light in space. The light source is infinitely far away.

When using `[t_x, t_y, t_z] = [0, 0, 0]`, you may think of `[p_x, p_y, p_z]` as the *direction* where the light source is located.

When using `[p_x, p_y, p_z] = [0, 0, 0]`, you may think of `[t_x, t_y, t_z]` as the *direction* into which the light is shining.

By default, white light is created. Other colors can be chosen by the attribute `LightColor`.

When using the attribute `CameraCoordinates` = `TRUE`, the light source is fixed to the camera. It moves automatically, when the camera is moved.

Directed light such as `plot::DistantLight` create shading effects that add depth to the picture. Usually, a certain amount of undirected ambient light of type `plot::AmbientLight` enhances the picture.

Attributes

Attribute	Purpose	Default Value
`CameraCoordinates`	position of light sources relative to the camera?	`FALSE`
`Frames`	the number of frames in an animation	`50`
`LightColor`	the color of light	`RGB::White`
`LightIntensity`	intensity of light	`1.0`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`Position`	positions of cameras, lights, and text objects	
`PositionX`	x-positions of cameras, lights, and text objects	
`PositionY`	y-positions of cameras, lights, and text objects	
`PositionZ`	z-positions of cameras, lights, and text objects	

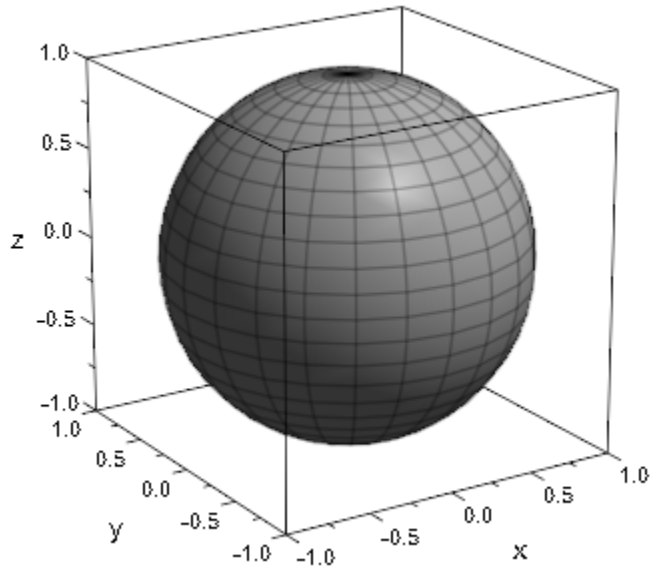
Attribute	Purpose	Default Value
`Target`	the target point of a light	
`TargetX`	the target point of a light, x component	
`TargetY`	the target point of a light, y component	
`TargetZ`	the target point of a light, z component	
`TimeEnd`	end time of the animation	`10.0`
`TimeBegin`	start time of the animation	`0.0`
`TimeRange`	the real time span of an animation	`0.0 .. 10.0`
`Visible`	visibility	`TRUE`

Examples

Example 1

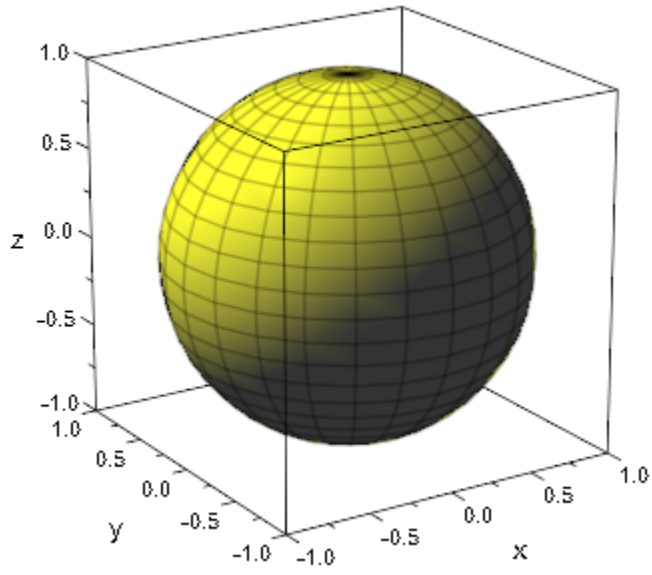
We create a white sphere and use a single directed white light to illuminate it:

```
f := plot::Surface(
        [cos(u)*sin(v), sin(u)*sin(v), cos(v)],
        u = 0..2*PI, v = 0..PI,
        FillColorType = Flat,
        FillColorFunction = RGB::White,
        Scaling = Constrained):
sunlight1 := plot::DistantLight([1, -2, 3], [0, 0, 0], 1/2):
plot(f, sunlight1):
```



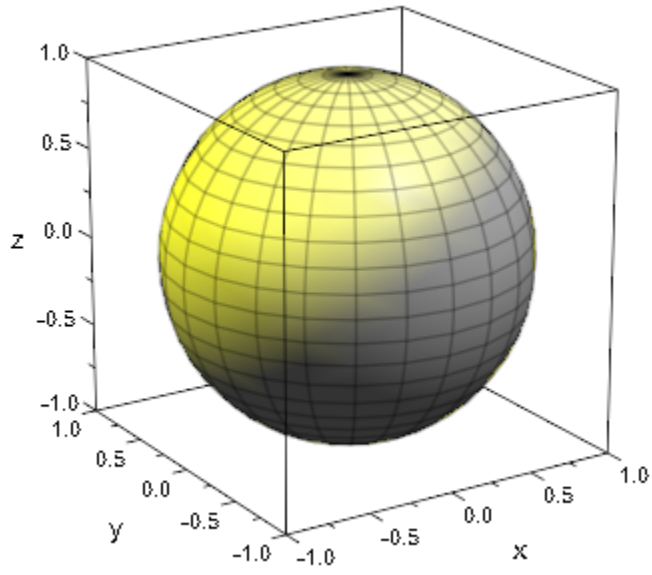
We create another distant light source shining from another direction, of yellow color and with animated intensity:

```
sunlight2 := plot::DistantLight([-2, 1, 3], [0, 0, 0], a,
                                LightColor = RGB::Yellow,
                                a = 0..1):
plot(f, sunlight2)
```



We use both lights simultaneously:

```
plot(f, sunlight1, sunlight2)
```




```
delete f, sunlight1, sunlight2:
```

Parameters

p_x, p_y, p_z

The coordinates of the sun's "position": numerical values or arithmetical expressions of the animation parameter `a`.

p_x, p_y, p_z are equivalent to the attributes `PositionX`, `PositionY`, `PositionZ`.

t_x, t_y, t_z

The coordinates of the point the light is shining to: numerical values or arithmetical expressions of the animation parameter `a`.

t_x, t_y, t_z are equivalent to the attributes `TargetX`, `TargetY`, `TargetZ`.

intensity

The intensity of the light: a numerical value between 0 and 1 or an arithmetical expression of the animation parameter `a`.

intensity is equivalent to the attribute `LightIntensity`.

a

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::AmbientLight` | `plot::PointLight` | `plot::SpotLight`

plot::PointLight

Point light

Syntax

```
plot::PointLight([x, y, z], <intensity>, <a = a_min .. a_max>, options)
```

Description

`plot::PointLight([x, y, z], intensity)` generates a point light at the position (*x*, *y*, *z*).

Each 3D scene is illuminated by several light sources that are set automatically and cannot be controlled by the user. Cf. the help page of `Lighting`.

If special light effects are desired, the user can create alternative light sources of various types such as `plot::AmbientLight`, `plot::DistantLight`, `plot::PointLight`, and `plot::SpotLight`.

If at least one user defined light source is inserted into the scene (e.g., by simply passing the light objects as input parameters to the `plot` command), the automatic lights are switched off and the user defined lights are used to illuminate the scene.

`plot::PointLight([x, y, z], intensity)` creates a point light at the position (*x*, *y*, *z*). It emits light into all directions.

Unlike in real life, the light flux of a point light does not decrease with the distance to the light source.

By default, white light is created. Other colors can be chosen by the attribute `LightColor`.

When using the attribute `CameraCoordinates` = `TRUE`, the light source is fixed to the camera. It moves automatically, when the camera is moved.

Light sources such as `plot::PointLight` create shading effects that add depth to the picture.

Usually, you will use point lights to highlight special details of the scene. For the illumination of the entire scene you will usually need additional undirected ambient light of type `plot::AmbientLight`, too.

Note that all light sources create a homogeneous lighting effect for a 3D triangle. Thus, realistic shading effects can only be achieved for surfaces with a sufficiently fine triangulation. For function graphs (`plot::Function3d`) and parametrized surfaces (`plot::Surface`), a fine triangulation is created by sufficiently high values of `XMesh`, `YMesh` or `UMesh`, `VMesh`, respectively.

Attributes

Attribute	Purpose	Default Value
`CameraCoordinates`	position of light sources relative to the camera?	`FALSE`
`Frames`	the number of frames in an animation	`50`
`LightColor`	the color of light	`RGB::White`
`LightIntensity`	intensity of light	`1.0`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`Position`	positions of cameras, lights, and text objects	
`PositionX`	x-positions of cameras, lights, and text objects	

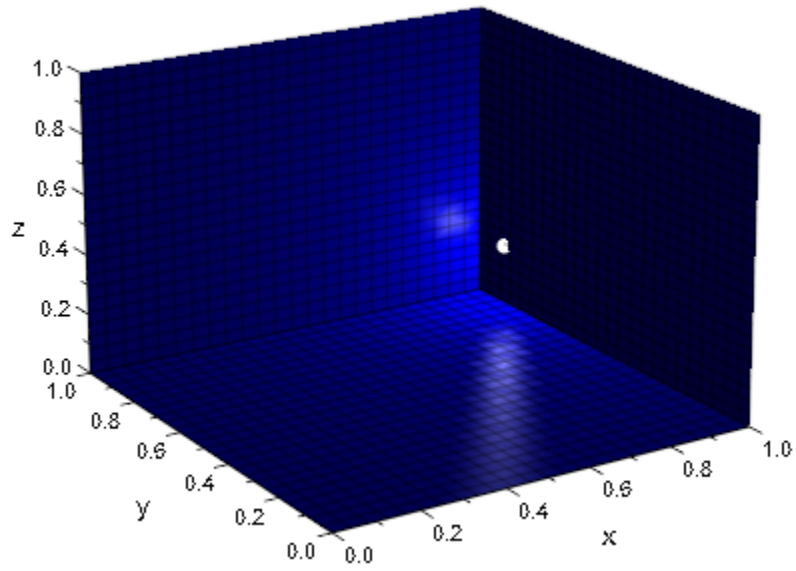
Attribute	Purpose	Default Value
PositionY	y-positions of cameras, lights, and text objects	
PositionZ	z-positions of cameras, lights, and text objects	
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Visible	visibility	TRUE

Examples

Example 1

We create three blue surfaces and illuminate them by an animated point light. The position of the point light is indicated by a white point:

```
s1 := plot::Surface([1, y, z], y = 0..1, z = 0..1):
s2 := plot::Surface([x, 1, z], x = 0..1, z = 0..1):
s3 := plot::Surface([x, y, 0], x = 0..1, y = 0..1):
p := plot::Point3d([a, 0.9, 0.2], a = 0..1,
                   PointSize = 2.0*unit::mm,
                   PointColor = RGB::White):
light := plot::PointLight([a, 0.9, 0.2], 1, a = 0..1):
plot(s1, s2, s3, p, light, Axes = Frame,
     FillColor = RGB::Blue, FillColorType = Flat):
```




```
delete s1, s2, s3, p, light:
```

Parameters

`x, y, z`

The coordinates of the point light: numerical values or arithmetical expressions of the animation parameter `a`.

x, y, z are equivalent to the attributes `Position`, `PositionX`, `PositionY`, `PositionZ`.

`intensity`

The intensity of the light: a numerical value between 0 and 1 or an arithmetical expression of the animation parameter `a`.

intensity is equivalent to the attribute `LightIntensity`.

a

Animation parameter, specified as a `= a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions
`plot` | `plot::copy`

MuPAD Graphical Primitives
`plot::AmbientLight` | `plot::DistantLight` | `plot::SpotLight`

plot::SpotLight

Spot light

Syntax

```
plot::SpotLight([p_x, p_y, p_z], [t_x, t_y, t_z], angle, <intensity>, <a = a_min .. a_max>, options
```

Description

`plot::SpotLight([`p_x`, `p_y`, `p_z`], [`t_x`, `t_y`, `t_z`], angle, intensity)` generates a spot light at the position p_x, p_y, p_z, aimed at the point t_x, t_y, t_z. The opening angle of the light cone is given by `angle`.

Each 3D scene is illuminated by several light sources that are set automatically and cannot be controlled by the user. Cf. the help page of `Lighting`.

If special light effects are desired, the user can create alternative light sources of various types such as `plot::AmbientLight`, `plot::DistantLight`, `plot::PointLight`, and `plot::SpotLight`.

If at least one user defined light source is inserted into the scene (e.g., by simply passing the light objects as input parameters to the `plot` command), the automatic lights are switched off and the user defined lights are used to illuminate the scene.

`plot::SpotLight([`p_x`, `p_y`, `p_z`], [`t_x`, `t_y`, `t_z`], angle)` creates a spot light source at the point [p_x, p_y, p_z] emitting a light cone towards the point [t_x, t_y, t_z]. In contrast to real life, the light flux of a spot light does not decrease with the distance to the light source.

By default, white light is created. Other colours can be chosen by the attribute `LightColor`.

When using the attribute `CameraCoordinates` = `TRUE`, the light source is fixed to the camera. It moves automatically, when the camera is moved.

Directed light such as `plot::SpotLight` create shading effects that add depth to the picture.

Usually, you will use spot lights to highlight special details of the scene. For the illumination of the entire scene you will usually need additional undirected ambient light of type `plot::AmbientLight`, too.

Note that all light sources create a homogeneous lighing effect for a 3D triangle. Thus, realistic shading effects can only be achieved for surfaces with a sufficiently fine triangulation. For function graphs (`plot::Function3d`) and parametrized surfaces (`plot::Surface`), a fine triangulation is created by sufficiently high values of `XMesh`, `YMesh` or `UMesh`, `VMesh`, respectively.

Attributes

Attribute	Purpose	Default Value
`CameraCoordinates`	position of light sources relative to the camera?	`FALSE`
`Frames`	the number of frames in an animation	`50`
`LightColor`	the color of light	`RGB::White`
`LightIntensity`	intensity of light	`1.0`
`Name`	the name of a plot object (for browser and legend)	
`ParameterEnd`	end value of the animation parameter	
`ParameterName`	name of the animation parameter	
`ParameterBegin`	initial value of the animation parameter	
`ParameterRange`	range of the animation parameter	
`Position`	positions of cameras, lights, and text objects	
`PositionX`	x-positions of cameras, lights, and text objects	

Attribute	Purpose	Default Value
PositionY	y-positions of cameras, lights, and text objects	
PositionZ	z-positions of cameras, lights, and text objects	
SpotAngle	opening angle of the light cone of a spot light	
Target	the target point of a light	
TargetX	the target point of a light, x component	
TargetY	the target point of a light, y component	
TargetZ	the target point of a light, z component	
TimeEnd	end time of the animation	10.0
TimeBegin	start time of the animation	0.0
TimeRange	the real time span of an animation	0.0 .. 10.0
Visible	visibility	TRUE

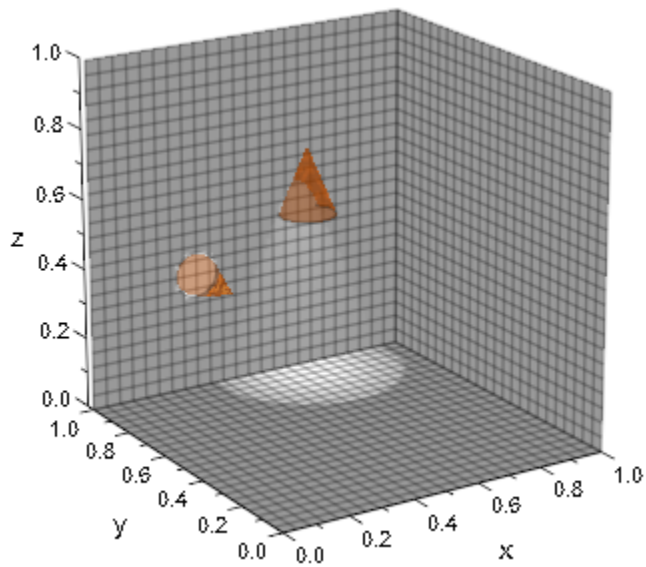
Examples

Example 1

We create three white surfaces and illuminate them by two animated white spot lights and some ambient light. The spot lights are indicated by little cones:

```
s1 := plot::Surface([1, y, z], y = 0..1, z = 0..1):
s2 := plot::Surface([x, 1, z], x = 0..1, z = 0..1,
                    Submesh = [2, 2]):
s3 := plot::Surface([x, y, 0], x = 0..1, y = 0..1,
                    Submesh = [2, 2]):
ambientlight:= plot::AmbientLight(0.4):
spotlight1 := plot::SpotLight([1/3, a, 1/3], [1/3, 1, 1/3],
```

```
                                PI/5, a = 0..0.8):
c1 := plot::Cone(0, [1/3, a, 1/3],
                 0.2*tan(PI/10), [1/3, a + 0.2, 1/3],
                 a = 0..0.8, Color = RGB::Orange.[0.5]):
spotlight2 := plot::SpotLight([2/3, a, 2/3], [2/3, a, 0],
                                PI/4, a = 0.15..0.95):
c2 := plot::Cone(0, [2/3, a, 2/3],
                 0.2*tan(PI/8), [2/3, a, 2/3 - 0.2],
                 a = 0.15..0.95, Color = RGB::Orange.[0.5]):
plot(s1, s2, s3, FillColor = RGB::White,
     FillColorType = Flat,
     ambientlight, c1, spotlight1,
     c2, spotlight2, Axes = Frame):
```




```
delete s1, s2, s3, c1, c2, ambientlight,
       spotlight1, spotlight2:
```

Parameters

p_x, p_y, p_z

The coordinates of the position: numerical values or arithmetical expressions of the animation parameter `a`.

p_x, p_y, p_z are equivalent to the attributes `Position`, `PositionX`, `PositionY`, `PositionZ`.

t_x, t_y, t_z

The coordinates of the point the light is shining to: numerical values or arithmetical expressions of the animation parameter `a`.

t_x, t_y, t_z are equivalent to the attributes `Target`, `TargetX`, `TargetY`, `TargetZ`.

`angle`

The opening angle of the light cone in radians: a numerical value between 0 and π or an arithmetical expression of the animation parameter `a`.

angle is equivalent to the attribute `SpotAngle`.

`intensity`

The intensity of the light: a numerical value between 0 and 1 or an arithmetical expression of the animation parameter `a`.

intensity is equivalent to the attribute `LightIntensity`.

`a`

Animation parameter, specified as `a = a_min..a_max`, where a_{min} is the initial parameter value, and a_{max} is the final parameter value.

See Also

MuPAD Functions

`plot` | `plot::copy`

MuPAD Graphical Primitives

`plot::AmbientLight` | `plot::DistantLight` | `plot::PointLight`

OutputFile, OutputOptions

File name to plot into

Value Summary

`OutputFile`, `OutputOptions`	Optional	See below

Description

The attribute `OutputFile` allows to specify a file name to direct the plot output into an external file instead of rendering the plot on the screen. The extension of the file name indicates the export format.

The available formats include `.xvz` and `.xvc` (the MuPAD proprietary XML format) as well as various standard bitmap formats such as `.bmp, .gif, .jpg` etc. and vector formats such as`.eps`, `.svg` etc. Animated MuPAD graphics can be exported to animated GIF files. On Windows systems, it also can be exported to `.avi` format.

Some of the export formats allow to specify certain parameters such as “resolution,” “quality” etc. Such parameters may be specified by the attribute `OutputOptions`.

MuPAD plots can be saved in “batch mode” by specifying the attribute `OutputFile = filename` in a `plot` call or in `plotfunc2d`, `plotfunc3d`. The file name must be a MuPAD string. For example:

```
plot(..graphical objects.., OutputFile = "mypicture.xvz"):
```

(Here, the extension `.xvz` of the file name `"mypicture.xvz"` indicates that the MuPAD XML data are to be written).

If the MuPAD environment variable `WRITEPATH` does not have a value, the previous call creates the file in the directory where MuPAD is running. (On Windows and Macintosh systems, this is, by default, the directory where it is installed).

After setting `WRITEPATH` to the absolute path of a folder, the file is created in that folder. For example, after

```
WRITEPATH := "C:\\Documents":
```

the command

```
plot(..graphical objects.., OutputFile = "mypicture.xvz"):
```

stores the plot data in the file "`C:\Documents\mypicture.xvz`." Alternatively, the file name can be specified as an absolute pathname:

```
plot(..objects.., OutputFile = "C:\\Documents\\mypicture.xvz"):
```

If a MuPAD notebook was saved to a file, its location is available inside the notebook as the environment variable `NOTEBOOKPATH`. If you wish to save your plot in the same folder as the notebook, you may call

```
plot(..objects.., OutputFile = NOTEBOOKPATH."mypicture.xvz"):
```

The plot data can be stored in various formats indicated by the extension of the file name. In particular, there are the MuPAD proprietary XML formats. The file extension `.xvz` indicates that XML data are to be written and, finally, the file is to be compressed. Alternatively, the extension `.xvc` may be used to write the XML data without final compression (the resulting text file can be read with any text editor). Files in both formats can be inserted into a MuPAD notebook and freely manipulated.

Apart from saving files as XML data, MuPAD pictures can also be saved in a variety of standard bitmap formats such as `.bmp`, `.gif`, `.jpg` etc. Also `.svg` and `.eps` export is available. Just use an appropriate extension of the file name indicating the format.

Note: Only XML files `.xvz` and `.xvc` retain the information necessary for interactive manipulation in a MuPAD notebook. All other formats are intended for exporting graphics to other programs.

If no file extension is specified in the file name, the default extension `.xvz` is used, i.e., XML data are written.

On Windows systems, animated MuPAD plots can be exported to `.avi` format. Cf. "Example 2" on page 24-1285.

In addition to `OutputFile`, there is the attribute `OutputOptions` to specify parameters for some of the export formats. The admissible value for this attribute is a list of equations:

```
OutputOptions = [<ReduceTo256Colors = b >, <DotsPerInch = n1>,
<Quality = n2>, <JPEGMode = n3>, <EPSMode = n4>, <AVIMode = n5>,
<WMFMode = n6>, <FramesPerSecond = n7>, <PlotAt = l1>]
```

Each entry of the list is optional. The parameters are

`b`	`TRUE` or `FALSE`. Has an effect for export to some raster formats only. With `TRUE`, only 256 different colors are stored in the raster file. The default value is `FALSE`.
n_1	Positive integer setting the resolution in DPI (dots per inch). Has an effect for export to raster formats only. The default value depends on the hardware.
n_2	One of the integers 1, 2, ..., 100. This integer represents a percentage value determining the quality of the export. Has an effect for JPG, 3D EPS, 3D WMF and AVI export only. The default value is `75`.
n_3	`0`, `1`, or `2`. Has an effect for JPG export only. The flag `0` represents the JPG mode `Baseline Sequential', `1` represents `Progressive', `2` represents `Sequential Optimized'. The default value is `0`.
n_4	`0` or `1`. Has an effect for EPS export only. The flag `0` represents the EPS mode `Painter's Algorithm', `1` represents `BSP Tree Algorithm'. The default value is `0`.
n_5	`0`, `1` or `2`. Has an effect for AVI export only. With `0`, the `Microsoft Video 1 Codec' is used. With `1`, the `Uncompressed Single Frame Codec' is used. With `2`, the `Radius Cinepak Codec' is used. The default value is `0`.

n_6	0, 1 or 2. Has an effect for WMF export only. With 0, the `Painter's Algorithm' is used. With 1, the `BSP Tree Algorithm' is used. With 2, a `embedded bitmap' is created. The default value is 0.
n_7	Positive integer setting the frames per second for the AVI to be generated. Has an effect for AVI and animated GIF export only. The default value is 15.
l_1	List of real values between `TimeBegin` and `TimeEnd` which determines the times at which pictures should be saved from an animation.

Examples

Example 1

The following commands save the plot in four different files in JPG, EPS, SVG, and BMP format, respectively:

```
f1 := plot::Function2d(sin(x), x = 0..PI, Color = RGB::Red):
f2 := plot::Function2d(cos(x), x = 0..PI, Color = RGB::Blue):
plot(f1, f2, OutputFile = "mypicture.jpg"):
plot(f1, f2, OutputFile = "mypicture.eps"):
plot(f1, f2, OutputFile = "mypicture.svg"):
plot(f1, f2, OutputFile = "mypicture.bmp"):
```

If no file extension is specified in the file name, the default extension `.xvz` is used, i.e., XML data are written. The following command creates the file `mypicture.xvz`:

```
plot(f1, f2, OutputFile = "mypicture"):
```

Example 2

An animated MuPAD plot can be exported to `.avi` format:

```
plotfunc2d(sin(x - a), x = 0 .. 2*PI, a = 0 .. 5,
```

```
          OutputFile = "myanimation.avi"):
```

Example 3

An animated MuPAD plot can be exported to serveral single images at given times:

```
plotfunc2d(sin(x - a), x = 0 .. 2*PI, a = 0 .. 5,
          OutputFile = "someName.png",
          OutputOptions=[PlotAt = [i $ i = 0..10 step  0.5]]):
```

More About

- “Save in Batch Mode”

AffectViewingBox

Influence of objects on the ViewingBox of a scene

Value Summary

Inherited	FALSE, or TRUE

Graphics Primitives

Objects	AffectViewingBox Default Values
plot::Arc2d, plot::Arc3d, plot::Arrow2d, plot::Arrow3d, plot::Bars2d, plot::Bars3d, plot::Box, plot::Boxplot, plot::Circle2d, plot::Circle3d, plot::Cone, plot::Conformal, plot::Curve2d, plot::Curve3d, plot::Cylinder, plot::Cylindrical, plot::Density, plot::Dodecahedron, plot::Ellipse2d, plot::Ellipse3d, plot::Ellipsoid, plot::Function2d, plot::Function3d, plot::Hatch, plot::Hexahedron, plot::Histogram2d, plot::Icosahedron, plot::Implicit2d, plot::Implicit3d, plot::Inequality, plot::Integral, plot::Iteration, plot::Line2d, plot::Line3d, plot::Listplot, plot::Lsys, plot::Matrixplot, plot::MuPADCube, plot::Octahedron, plot::Ode2d, plot::Ode3d, plot::Parallelogram2d,	TRUE

Objects	AffectViewingBox Default Values
plot::Parallelogram3d, plot::Piechart2d, plot::Piechart3d, plot::Plane, plot::Point2d, plot::Point3d, plot::PointList2d, plot::PointList3d, plot::Polar, plot::Polygon2d, plot::Polygon3d, plot::Prism, plot::Pyramid, plot::QQplot, plot::Raster, plot::Rectangle, plot::Reflect2d, plot::Reflect3d, plot::Rootlocus, plot::Rotate2d, plot::Rotate3d, plot::Scale2d, plot::Scale3d, plot::Scatterplot, plot::Sequence, plot::SparseMatrixplot, plot::Sphere, plot::Spherical, plot::Streamlines2d, plot::Sum, plot::Surface, plot::SurfaceSet, plot::SurfaceSTL, plot::Sweep, plot::Tetrahedron, plot::Text2d, plot::Text3d, plot::Transform2d, plot::Transform3d, plot::Translate2d, plot::Translate3d, plot::Tube, plot::Turtle, plot::VectorField2d, plot::VectorField3d, plot::Waterman, plot::XRotate, plot::ZRotate	

Description

`AffectViewingBox` determines whether the `ViewingBox` of an object should be taken into account for the total `ViewingBox` of the graphical scene.

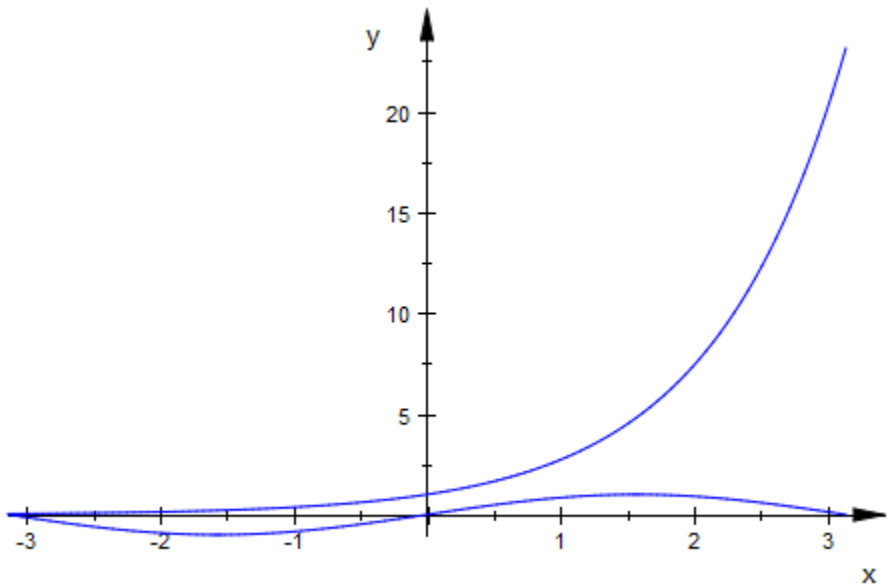
Usually, the visible area/volume of a graphical scene is automatically chosen as the smallest box containing all objects of the scene. Objects with `AffectViewingBox = FALSE` are ignored in the computation of this box.

Examples

Example 1

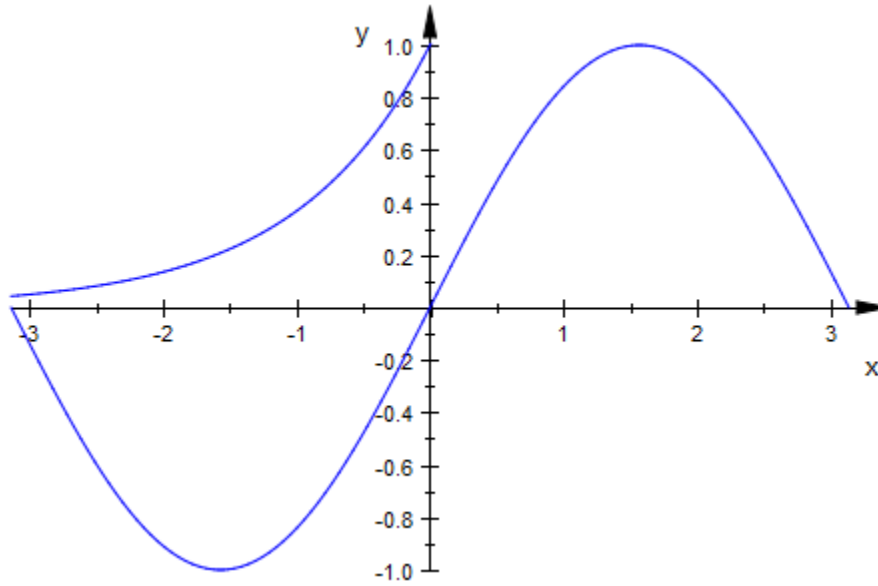
We plot the sine and the exponential function in one scene:

```
plot(plot::Function2d(sin(x), x = -PI..PI),
     plot::Function2d(exp(x), x = -PI..PI))
```



The exponential function dominates the sine. We set `AffectViewingBox = FALSE` for `exp`. Now, only the sine function determines the visible area and `exp` is only visible where it is in the `ViewingBox` of the sine function:

```
plot(plot::Function2d(sin(x), x = -PI..PI),
     plot::Function2d(exp(x), x = -PI..PI,
                      AffectViewingBox = FALSE))
```



See Also

MuPAD Functions
`ViewingBox`

Angle

Rotation angle

Value Summary

Optional	MuPAD expression

Graphics Primitives

Objects	Angle Default Values
`plot::Arc2d`, `plot::Arc3d`, `plot::Prism`, `plot::Pyramid`, `plot::Rotate2d`, `plot::Rotate3d`	`0`

Description

`Angle` determines the rotation angle in transformation objects of type `plot::Rotate2d` and `plot::Rotate3d`, respectively, and other graphical objects. The angle has to be specified in radians.

In 2D, the direction of the rotation is counter clock wise. Use negative angles to rotate clock wise.

In 3D, the rotation is implemented following the “right hand rule”: Stretch the thumb of your right hand and bend the fingers. When the thumb points into the direction of the rotation axis, your finger tips indicate the direction of the rotation. Use negative angles to rotate in the opposite direction.

Examples

Example 1

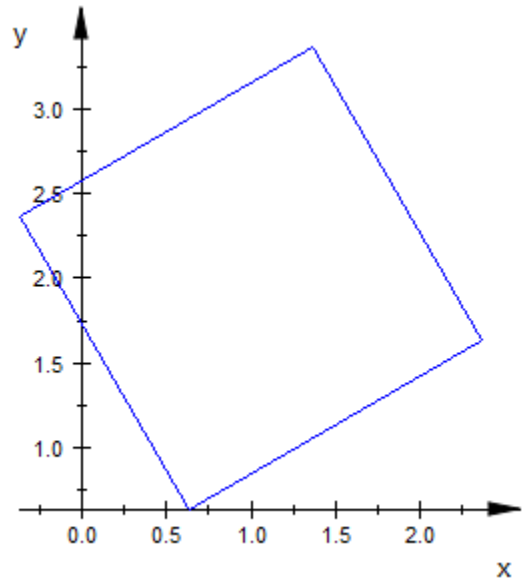
Rectangles of type `plot::Rectangle` are always parallel to the coordinate axes. To have one with a different orientation, we must rotate it:

```
r0 := plot::Rectangle(0..2, 1..3):
r1 := plot::Rotate2d(r0, Center = [1, 2], Angle = PI/6)
```

$$\text{plot::Rotate2d}\left(\frac{\pi}{6}, \text{Center} = [1, 2], \text{plot::Rectangle}(0..2, 1..3)\right)$$

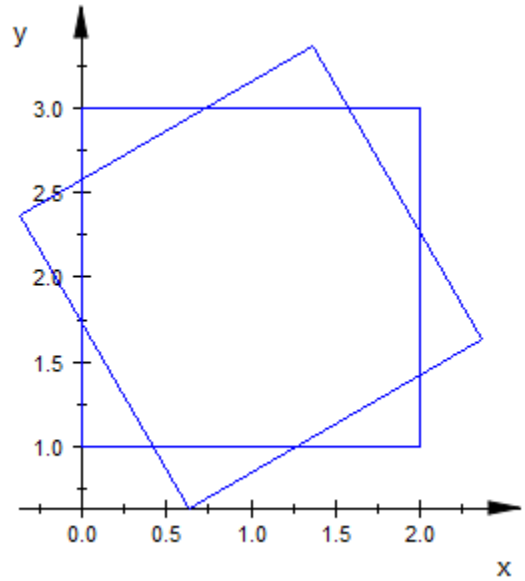
As you can see in the output above, the rotate object *contains* the rotated object and acts as a grouping construct. This means that we only need to plot `r1` to see the rotated object:

```
plot(r1)
```



Plotting both `r0` and `r1` yields a plot showing the rotated rectangle together with the unrotated one:

```
plot(r0, r1)
```

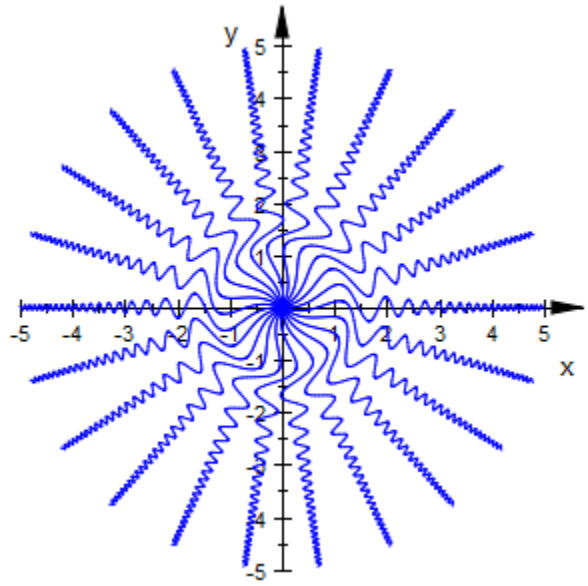



```
delete r0, r1:
```

Example 2

Using `plot::Rotate2d`, we plot several copies of a function plot, rotated at different angles:

```
f := plot::Function2d(sin(x^3)/(x^2+1), x = -5..5, Mesh = 300):
plot(plot::Rotate2d(f, Angle = PI/11*a) $ a = 0..10):
```

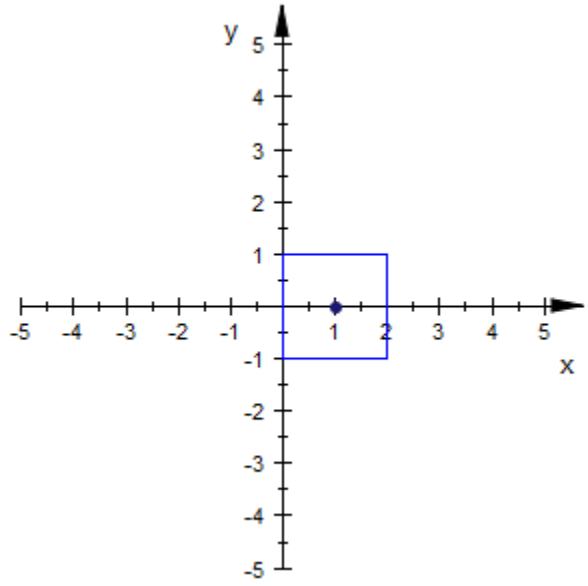



```
delete f:
```

Example 3

The rotation angle can be animated. We use an animated `plot::Rotate2d` object to rotate a square around a center that moves along a circle around the origin:

```
p := plot::Point2d([cos(a), sin(a)], a = 0..2*PI,
                   Frames = 100):
r := plot::Rotate2d(plot::Rectangle(0..2, -1..1), Angle = a,
                    Center = [0, 0], a = 0..2*PI):
q := plot::Rotate2d(r, Angle = 4*a, Center = [cos(a), sin(a)],
                    a = 0..2*PI, Frames = 200):
plot(p, q)
```




```
delete p, r, q:
```

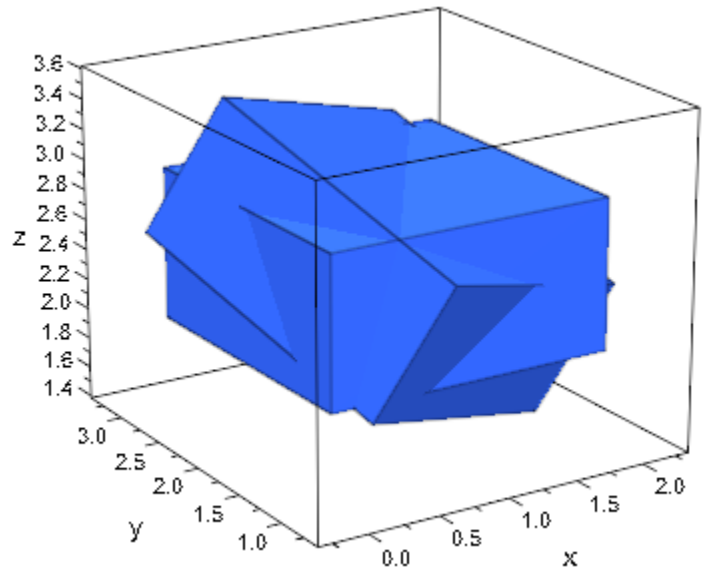
Example 4

This is the 3D analogue of “Example 1” on page 24-1291. Boxes of type `plot::Box` are always parallel to the coordinate axes. To have one with a different orientation, we must rotate it:

```
b0 := plot::Box(0..2, 1..3, 2..3):
b1 := plot::Rotate3d(b0, Center = [1, 2, 2.5],
                     Axis = [1, 1, 1], Angle = PI/5)
```

$$\text{plot::Rotate3d}\left(\frac{\pi}{5}, \text{Center} = [1, 2, 2.5], \text{Axis} = [1, 1, 1], \text{plot::Box}(0..2, 1..3, 2..3)\right)$$

```
plot(b0, b1)
```

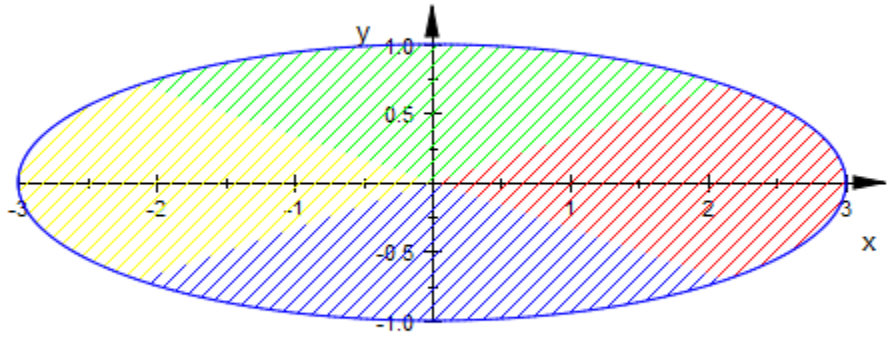



```
delete b0, b1:
```

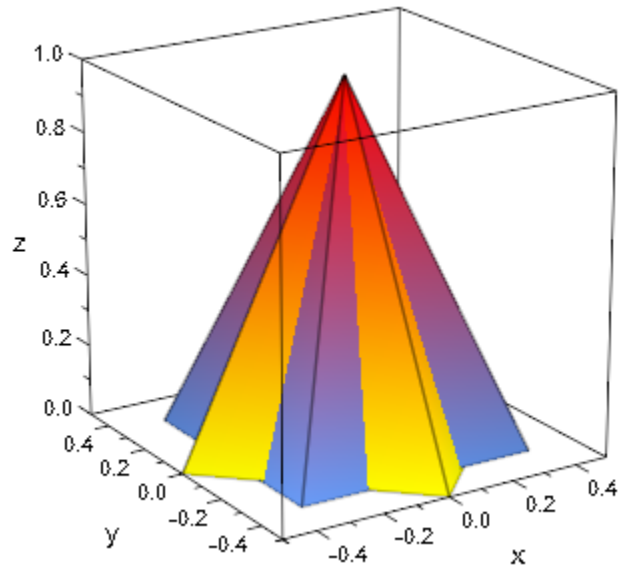
Example 5

Some graphic objects, e.g. `plot::Arc2d` and `plot::Pyramid`, also accept a rotation angle:

```
arc:= [3, 1], [0, 0], -PI/4..PI/4, Filled:
plot(
  plot::Arc2d(arc, Angle=0,      FillColor=RGB::Red),
  plot::Arc2d(arc, Angle=1/2*PI, FillColor=RGB::Green),
  plot::Arc2d(arc, Angle=PI,     FillColor=RGB::Yellow),
  plot::Arc2d(arc, Angle=3/2*PI, FillColor=RGB::Blue)
)
```




```
plot(plot::Pyramid(1/2, Angle=0),
     plot::Pyramid(1/2, Angle=PI/4, FillColor2=RGB::Yellow))
```




```
delete arc:
```

See Also

MuPAD Functions
`Axis` | `Center`

AngleRange, AngleBegin, AngleEnd

Angle range

Value Summary

`AngleRange`	[`AngleBegin` .. `AngleEnd`]	Range of arithmetical expressions
`AngleBegin`, `AngleEnd`	Optional	MuPAD expression

Graphics Primitives

Objects	Default Values

`plot::Tube`, `plot::XRotate`, `plot::ZRotate`	`AngleRange`: `0 .. 2*PI`
	`AngleBegin`: `0`
	`AngleEnd`: `2*PI`

`plot::Arc2d`, `plot::Arc3d`	`AngleRange`: `0 .. PI/2`
	`AngleBegin`: `0`
	`AngleEnd`: `PI/2`

Description

`AngleRange`, `AngleBegin`, `AngleEnd` define a range for the angle in circular arcs and surfaces of revolution.

For circular arcs of type `plot::Arc2d`, the attributes `AngleBegin` and `AngleEnd` define the starting point and the end point of the arc. The values are the usual polar angles measuring the angle to the positive *x*-axis in radians.

For surfaces of revolution of type `plot::XRotate` or `plot::ZRotate`, respectively, the attributes `AngleBegin` and `AngleEnd` define the starting point and the end point of the revolution.

For `plot::XRotate`, the values are the polar angles to the positive *y*-axis, specified in radians.

For `plot::ZRotate`, the values are the usual angles to the positive *x*-axis in radians, known from cylindrical coordinates.

Values for `AngleBegin` and `AngleEnd` may depend on the animation parameter and must evaluate to real numbers for any given time stamp.

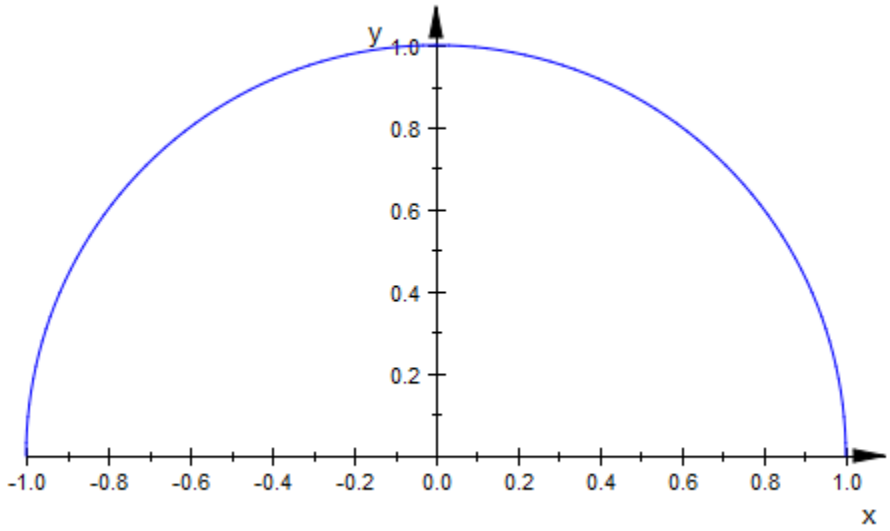
`AngleRange` provides a shortcut for setting `AngleBegin` and `AngleEnd`. The attribute `AngleRange = a_1..a_2` is equivalent to `AngleBegin = a_1`, `AngleEnd = a_2`.

Examples

Example 1

We define a semi-circle as a circular arc with a range of the polar angle from 0 to 180 degrees (i.e., π in radians):

```
arc := plot::Arc2d(1, 0 .. PI):
plot(arc)
```

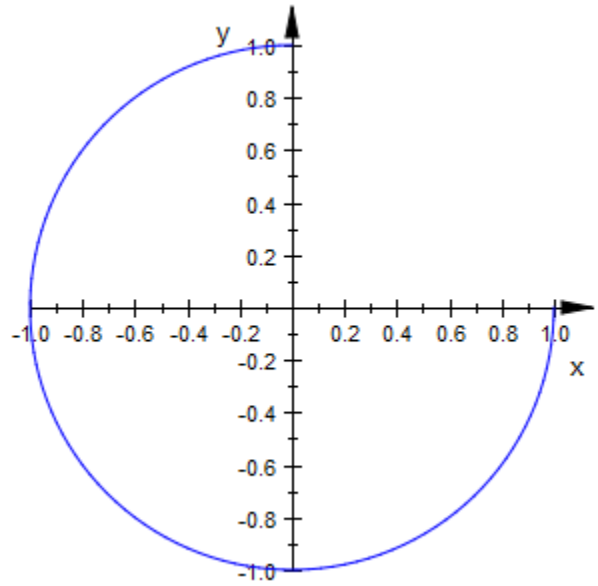


The range is stored as the attribute `AngleRange` in the object and can be accessed and changed:

```
arc::AngleBegin, arc::AngleEnd, arc::AngleRange
```

$$0,\ \pi,\ 0..\pi$$

```
arc::AngleRange :=  PI/2 .. 2*PI:
plot(arc)
```

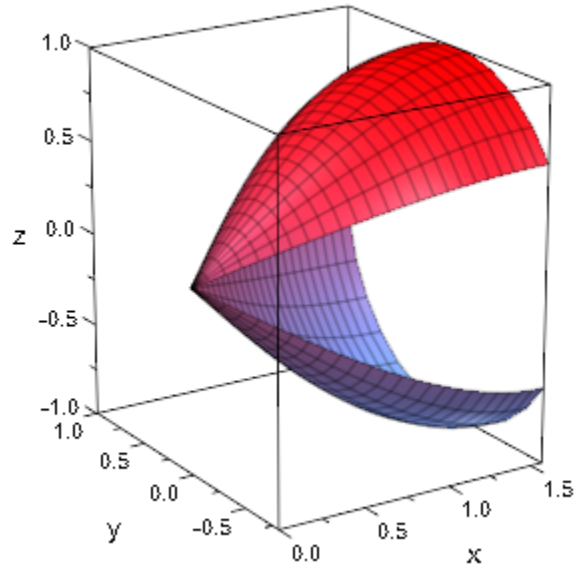



```
delete arc:
```

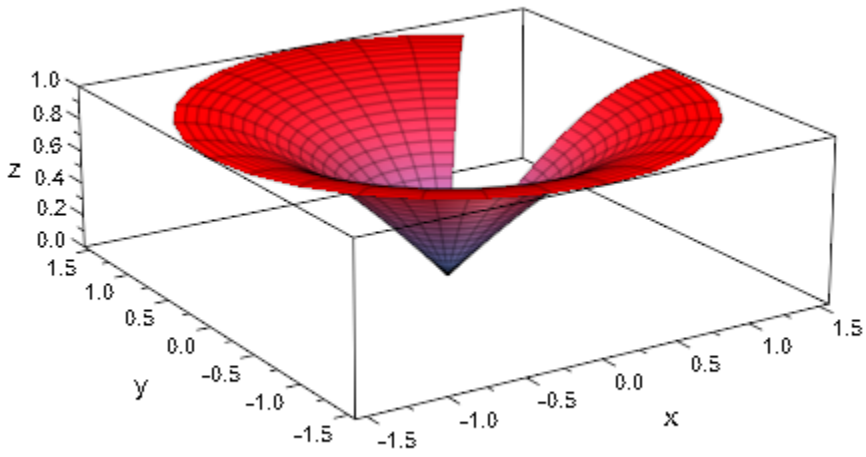
Example 2

We leave gaps in the following surfaces of revolution by restricting the revolution angle:

```
plot(plot::XRotate(sin(x), x = 0 .. PI/2,
                   AngleRange = -0.8*PI .. 0.8*PI)):
```




```
plot(plot::ZRotate(sin(x), x = 0 .. PI/2,
                   AngleRange =  0.3*PI .. 2*PI)):
```


1.0
0.8
0.6
z
0.4
0.2
0.0
1.5
1.0
0.5
0.0
-0.5
-1.0
-1.5
y
-1.5
-1.0
-0.5
0.0
0.5
1.0
1.5
x

Area

Area of a histogram plot

Value Summary

Optional	Non-negative real number

Graphics Primitives

Objects	Area Default Values
`plot::Histogram2d`	`0`

Description

`Area` determines whether bars of a histogram plot are scaled with respect to their heights or with respect to their areas, and by how much.

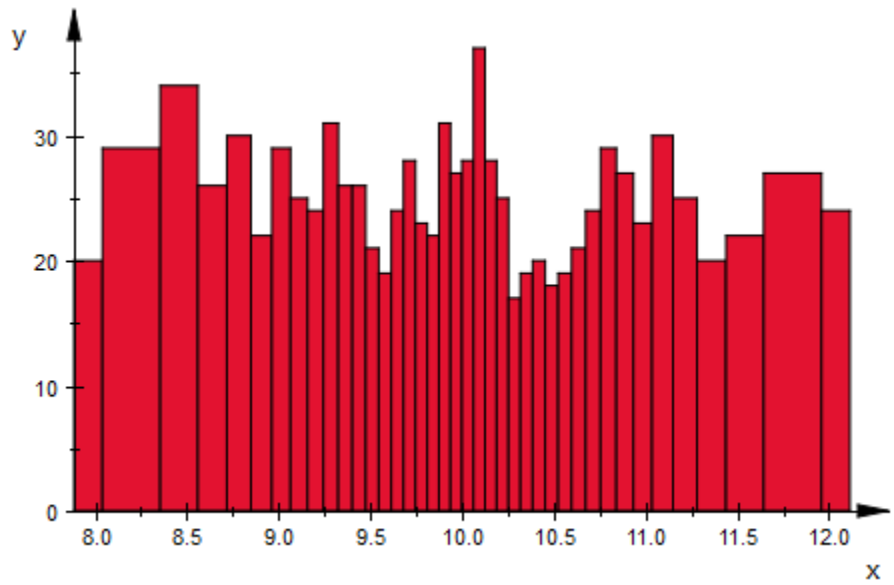
By default, the bars of a histogram plot use a height that is equal to the absolute number of data points in he corresponding cell. Using `Area`, the user can change this behavior to make the *areas* of the bars proportional to this number.

Examples

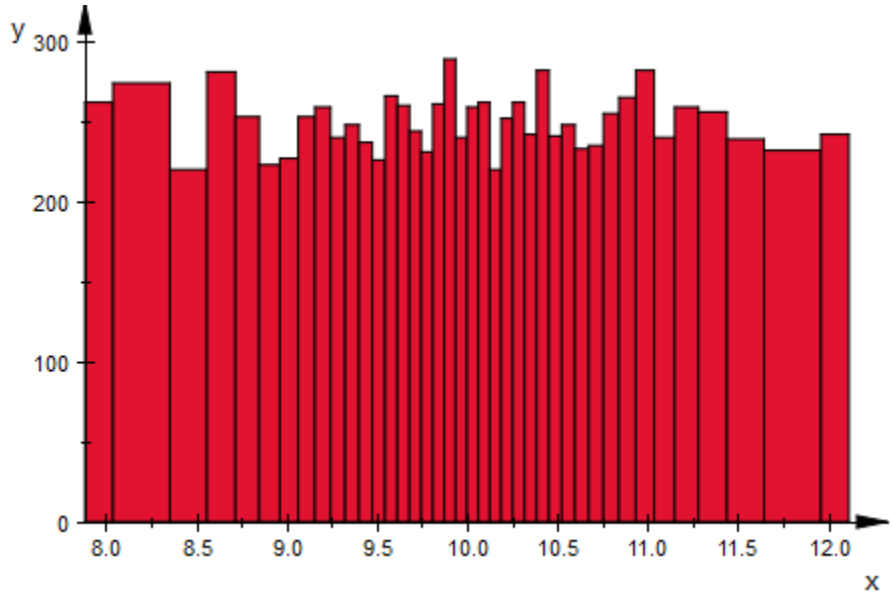
Example 1

For any distribution with a continuous quantile, we can define, using `stats::equiprobableCells`, a list of n cells where each cell is "hit" with the same probability, $\frac{1}{n}$. By the law of large numbers, we expect the number of elements in each cell to be approximating $\frac{N}{n}$ for large values of N, the number of samples:

```
X := stats::normalRandom(10, 1):
cells := stats::equiprobableCells(40,
            stats::normalQuantile(10, 1)):

N := 1000:
data := [X() $ i = 1..N]:
plot(plot::Histogram2d(data, Cells = cells))
```

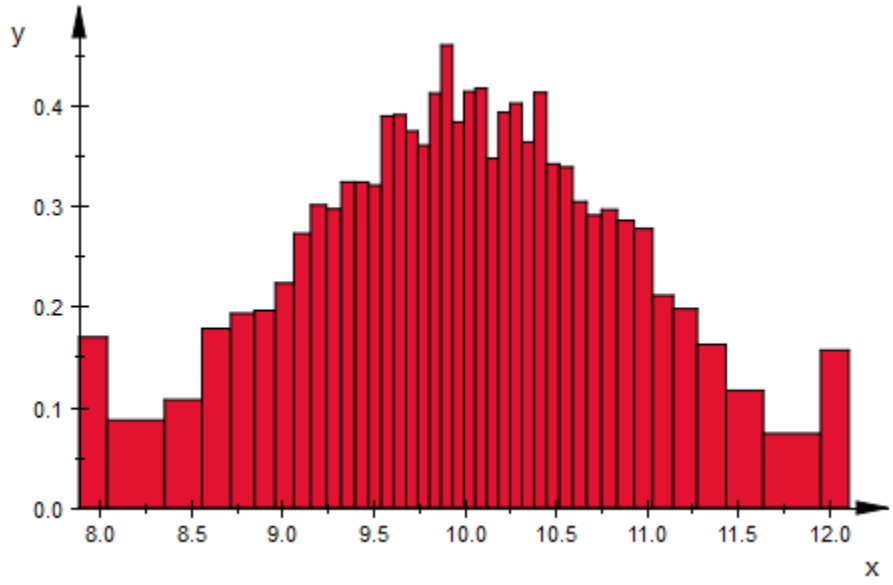



```
N := 10000:
data := [X() $ i = 1..N]:
plot(plot::Histogram2d(data, Cells = cells))
```

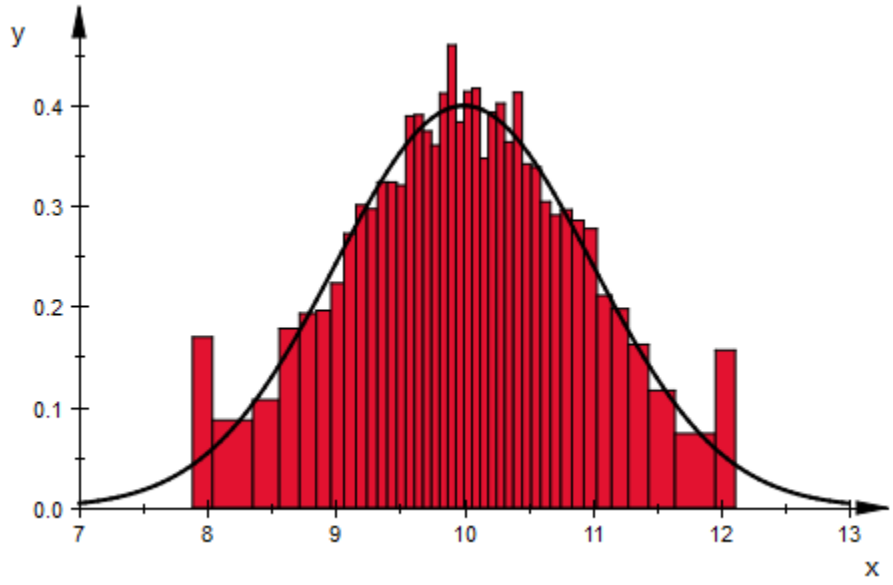


On the other hand, if we want to display a histogram as an approximation to the probability distribution, we want not the height, but rather the area of the rectangles to correspond to our measurements. Moreover, the sum of all areas should be 1, so we set `Area` to this value:

```
plot(plot::Histogram2d(data, Cells = cells, Area = 1))
```




```
plot(plot::Histogram2d(data, Cells = cells, Area = 1),
     plot::Function2d(stats::normalPDF(10,1)(x), x = 7..13,
                      Color = RGB::Black, LineWidth = 0.5))
```



See Also

MuPAD Functions
`Cells`

Averaged

Mode for computing quantile lines in box plots

Value Summary

Optional	TRUE or FALSE

Graphics Primitives

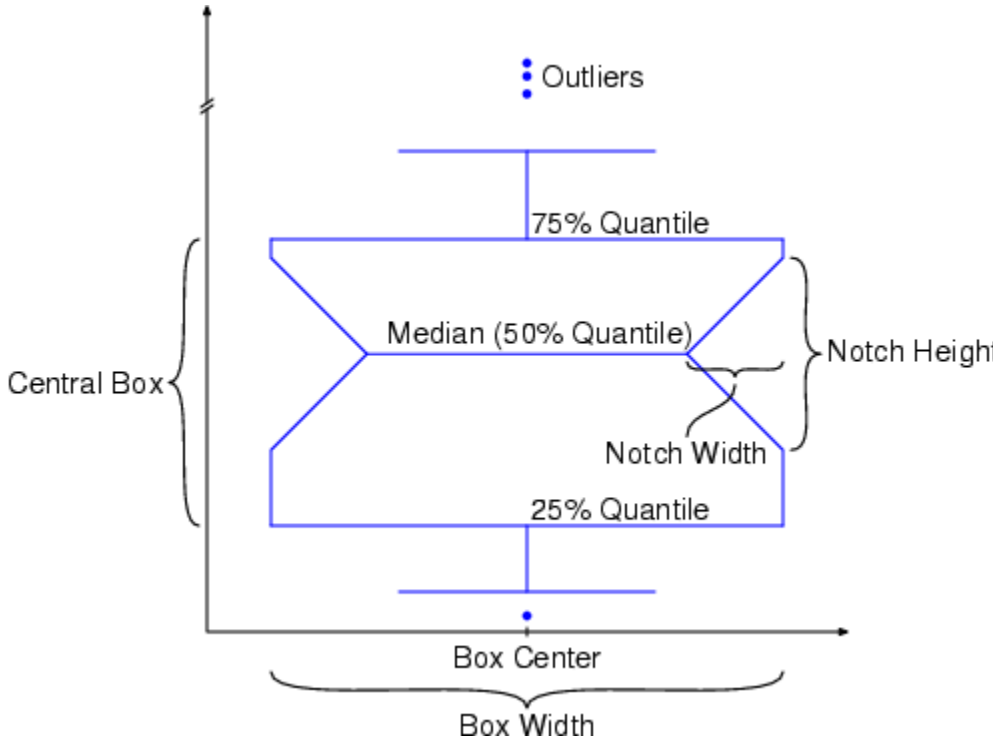
Objects	Averaged Default Values
plot::Boxplot	TRUE

Description

Statistical box plots indicate the 25%/50%/75% quantiles of data samples by horizontal lines. With the default `Averaged = TRUE`, the quantile lines are computed using `stats::empiricalQuantile` using the option `Averaged`.

A plot of type `plot::Boxplot` serves for visualizing and comparing statistical data samples. The plot reduces the data to few simple descriptive parameters.

A typical notched box looks like this:



The location of the 25%/50%/75% quantile lines are computed internally via `stats::empiricalQuantile`. When using `Averaged` = `TRUE` in the box plot, the quantile function is called with the option `Averaged` (see the help page of `stats::empiricalQuantile` for details).

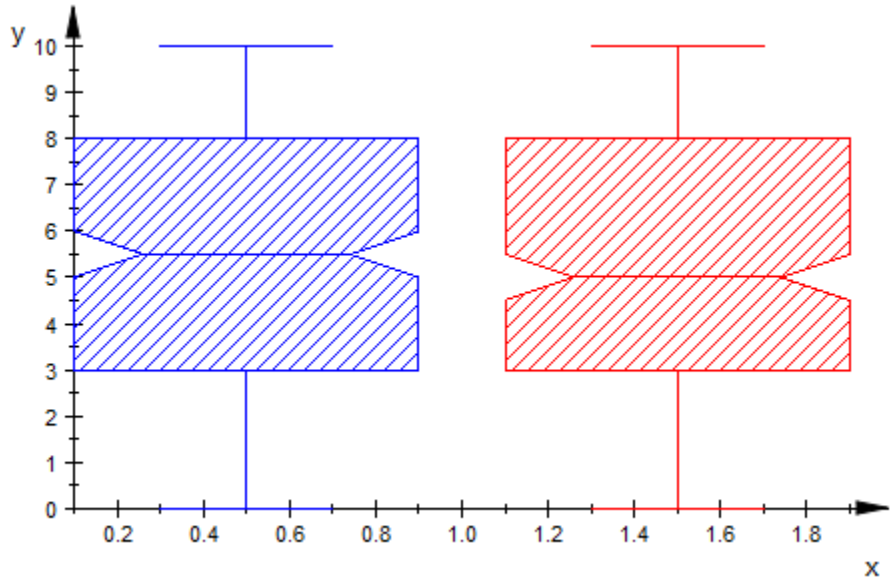
Examples

Example 1

By default, the quantile lines of the boxes are computed with the option `Averaged`. When using `Averaged` = `FALSE`, the quantiles are computed without this option:

```
r := random(0..10):
SEED := 123:
data := [r() $ k = 1..250]:
```

```
plot(
     plot::Boxplot(data, Averaged = TRUE, BoxCenters = 0.5,
                         Color = RGB::Blue, Notched),
     plot::Boxplot(data, Averaged = FALSE, BoxCenters = 1.5,
                         Color = RGB::Red, Notched)
    ):
```




```
delete r, SEED, data:
```

See Also

MuPAD Functions

BoxCenters | BoxWidths | DrawMode | Notched

Axis, AxisX, AxisY, AxisZ

Rotation axis

Value Summary

`Axis`	Library wrapper for "[`AxisX`, `AxisY`, `AxisZ`]" (3D)	List of 2 or 3 expressions, depending on the dimension
`AxisX`, `AxisY`, `AxisZ`	Optional	MuPAD expression

Graphics Primitives

Objects	Default Values
`plot::Rotate3d`	`Axis`: [0, 0, 1] `AxisX`, `AxisY`: 0 `AxisZ`: 1

Description

`Axis` is a vector determining the direction of the rotation axis in rotation objects of type `plot::Rotate3d`. It is given by a list of 3 components.

`AxisX` etc. refer to the x, y, z components of this vector.

A rotation in 3D is determined by a line around which is rotated. The line is given by a point on the line (the `Center`) and a direction vector (the `Axis`). The rotation angle is given by the attribute `Angle`.

The length of the `Axis` vector is of no relevance. However, it should not be zero.

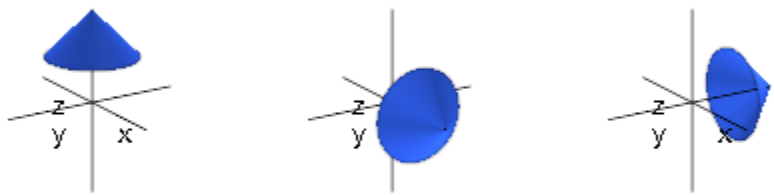
The rotation is implemented following the "right hand rule": Stretch the thumb of your right hand and bend the fingers. When the thumb points into the direction of the rotation axis, your finger tips indicate the direction of the rotation. Use negative angles to rotate in the opposite direction or replace the `Axis` vector by its negative.

Examples

Example 1

A cone is first rotated around the *x*-axis. The rotated cone is then rotated around the *z*-axis:

```
c0 := plot::Cone(1, [0, 0, 1], [0, 0, 2]):
c1 := plot::Rotate3d(c0, Center = [0, 0, 0],
                     Axis = [1, 0, 0], Angle = PI/2):
c2 := plot::Rotate3d(c1, Center = [0, 0, 0],
                     Axis = [0, 0, 1], Angle = PI/2):
plot(plot::Scene3d(c0, Axes = Origin,
                   ViewingBox = [-2..2, -2..2, -2..2]),
     plot::Scene3d(c1, Axes = Origin,
                   ViewingBox = [-2..2, -2..2, -2..2]),
     plot::Scene3d(c2, Axes = Origin,
                   ViewingBox = [-2..2, -2..2, -2..2]),
     TicksNumber = None,
     Width = 120*unit::mm, Height = 40*unit::mm,
     Layout = Horizontal):
```



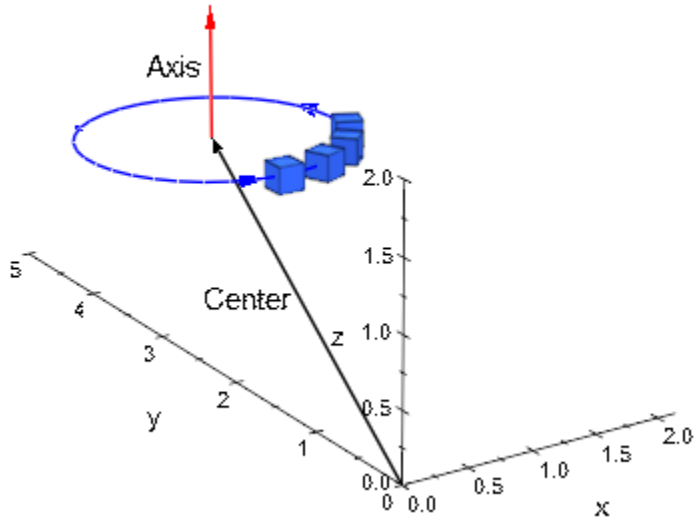

```
delete c0, c1, c2:
```

Example 2

We illustrate the “right hand rule”. A small box `b0` is rotated. The rotated copies `b1`, `b2`, `b3` are plotted together with the original box:

```
center := [1, 4, 1]:
axis := [0, 0, 1]:
```

```
b0 := plot::Box(0.9..1.1, 2.9 .. 3.1, 0.9 .. 1.1):
b1 := plot::Rotate3d(b0, Center = center, Axis = axis,
                     Angle = PI/8):
b2 := plot::Rotate3d(b1, Center = center, Axis = axis,
                     Angle = PI/8):
b3 := plot::Rotate3d(b2, Center = center, Axis = axis,
                     Angle = PI/8):
centerplusaxis := [center[i] + axis[i] $ i = 1..3]:
plot(b0, b1, b2, b3,
     plot::Arrow3d([0, 0, 0], center, Color = RGB::Black,
                   Title = "Center",
                   TitlePosition = [0.1, 2, 0.5]),
     plot::Arrow3d(center, centerplusaxis,
                   Title = "Axis", Color = RGB::Red,
                   TitlePosition = [0.7, 4, 1.5]),
     plot::Circle3d(1, center, axis),
     plot::Rotate3d(plot::Arrow3d([0, 4, 1], [0, 3.9, 1],
                                  Color = RGB::Blue),
                    Axis = axis, Center = center,
                    Angle = 0.43*PI + a*2*PI/3) $ a = 1..3,
     Axes = Origin
):
```



```
delete center, axis, b0, b1, b2, b3, centerplusaxis:
```

See Also

MuPAD Functions
`Angle` | `Center`

Base, Top, BaseX, TopX, BaseY, TopY, BaseZ, TopZ

Base center of cones, cylinders, pyramids and prisms

Value Summary

`Base`	Library wrapper for "[`BaseX`, `BaseY`]" (2D), "[`BaseX`, `BaseY`, `BaseZ`]" (3D)	List of 2 or 3 expressions, depending on the dimension
`Top`	Library wrapper for "[`TopX`, `TopY`]" (2D), "[`TopX`, `TopY`, `TopZ`]" (3D)	List of 2 or 3 expressions, depending on the dimension
`BaseX`, `BaseY`, `BaseZ`, `TopX`, `TopY`, `TopZ`	Mandatory	MuPAD expression

Graphics Primitives

Objects	Default Values
`plot::Cone`, `plot::Cylinder`, `plot::Prism`, `plot::Pyramid`	`Base`: [0, 0, 0] `Top`: [0, 0, 1] `BaseX`, `TopX`, `BaseY`, `TopY`, `BaseZ`: 0 `TopZ`: 1

Description

`Base` is a vector determining the position of the base center of cones/conical frustums, cylinders, pyramids/frustums of pyramids and prisms. It is given by a list or vector of 3 components.

`BaseX` etc. refer to the *x*, *y*, *z* components of this vector.

`Top` is a vector determining the position of the top center of cones/conical frustums, cylinders, pyramids/frustums of pyramids and prisms. For a cone, this is the tip. The vector is given by a list or vector of 3 components.

`TopX` etc. refer to the *x*, *y*, *z* components of this vector.

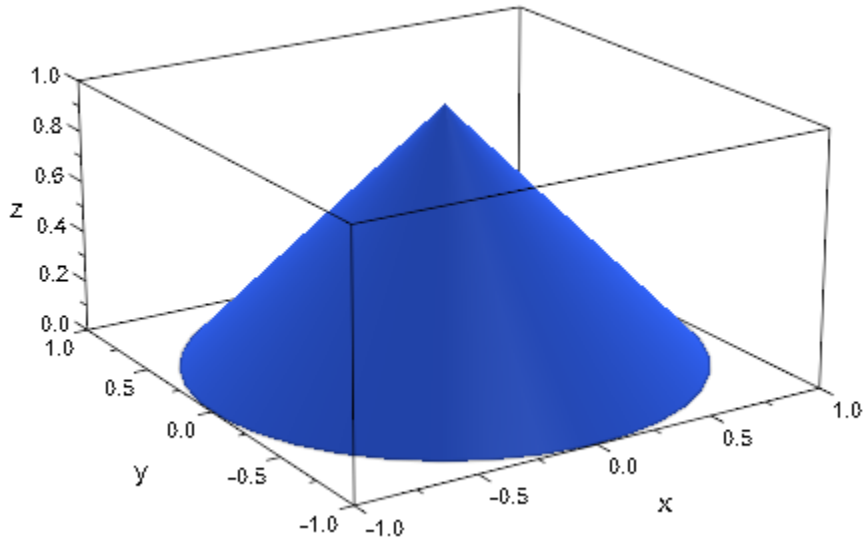
The values of these attributes can be animated.

Examples

Example 1

We define a cone:

```
c := plot::Cone(1, [0, 0, 0], [0, 0, 1]):
plot(c)
```

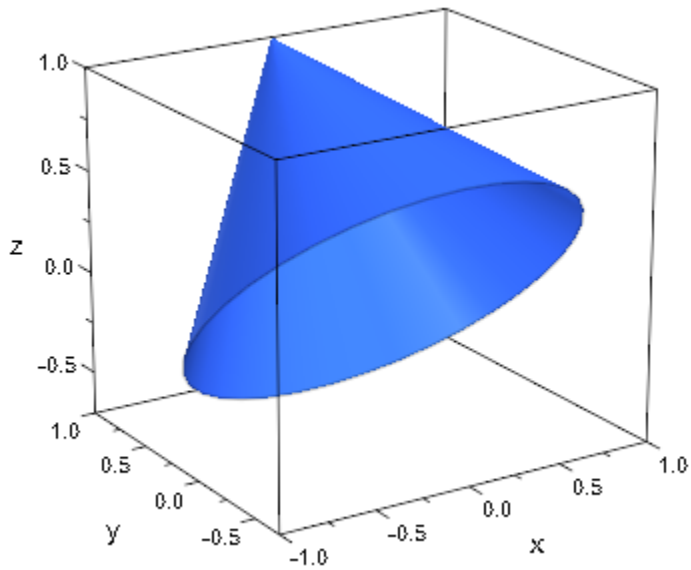


The second and third argument are the base center and the top center, respectively. Internally, they are stored as the attributes `Bottom` and `Top`. We can access the object's attributes and change them:

```
c::Base, c::Top
```

$$[0, 0, 0], [0, 0, 1]$$

```
c::Top := [0, 1, 1]:
plot(c):
```

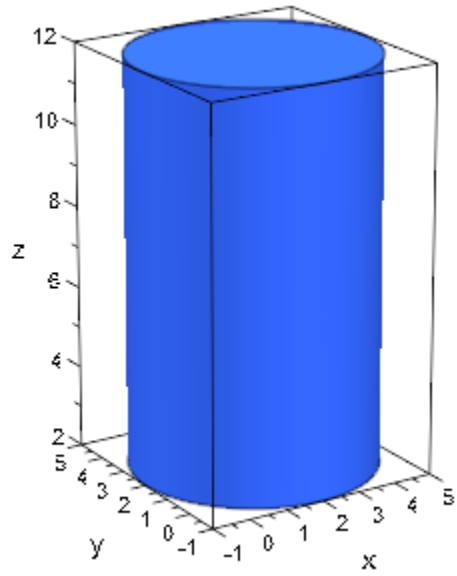



```
delete c:
```

Example 2

The values of `Bottom` and `Top` can be animated:

```
plot(plot::Cylinder(3, [2, 2, 2], [2, 2, a], a = 7..12)):
```



See Also

MuPAD Functions

`BaseRadius` | `TopRadius`

BaseRadius, TopRadius

Base and top radius of cones/conical frustums and pyramids/frustums of pyramids

Value Summary

`BaseRadius`, `TopRadius`	Mandatory	MuPAD expression

Graphics Primitives

Objects	Default Values
`plot::Cone`, `plot::Pyramid`	`BaseRadius`: 1 `TopRadius`: 0

Description

`BaseRadius` defines the radius of the base of a cone or the radius of the circumcircle of the regular base of a pyramid. `TopRadius` defines the radius r of the top of a conical frustum and the radius of the circumcircle of the top of a frustum of pyramid. With the default $r = 0$, a cone or pyramid, respectively, is created. You get a frustum for $r > 0$.

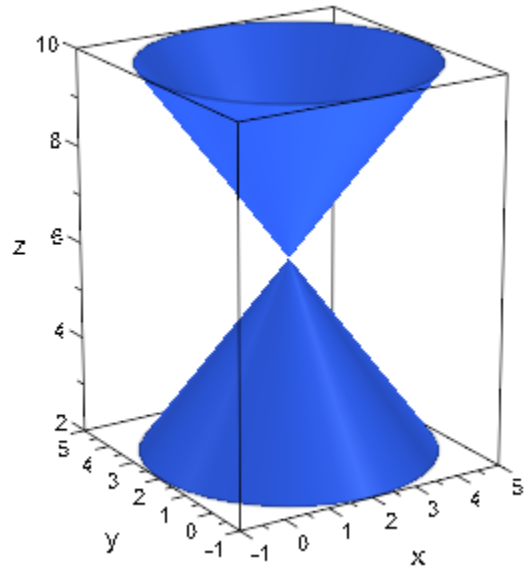
The values of these attributes can be animated.

Examples

Example 1

We draw two cones forming an hour glass:

```
c := plot::Cone(3, [2, 2, 2], [2, 2, 6]):
d := plot::Cone(3, [2, 2, 10], [2, 2, 6]):
plot(c, d)
```

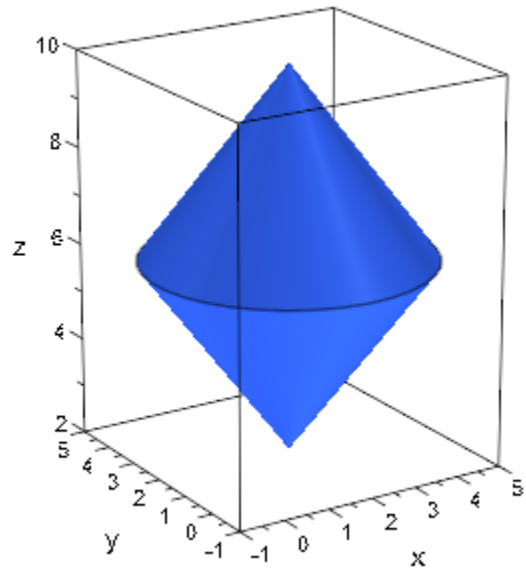


The first argument is the base radius of the cone. Internally, is is stored as the attribute `BaseRadius`. We can access the objects' attributes and change them:

```
c::BaseRadius, d::TopRadius
```

```
3, 0
```

```
c::BaseRadius := 0: c::TopRadius := 3:
d::BaseRadius := 0: d::TopRadius := 3:
plot(c, d):
```

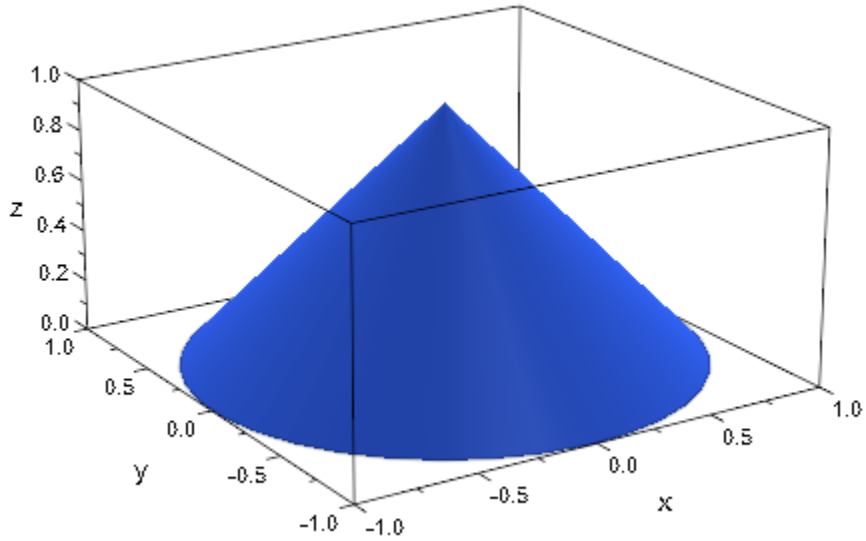



```
delete c, d:
```

Example 2

The values of `BaseRadius` and `TopRadius` can be animated:

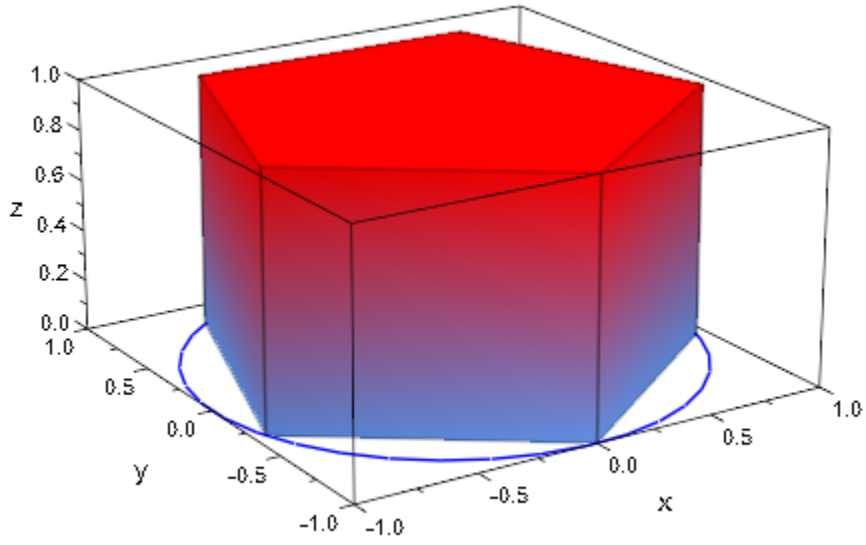
```
plot(plot::Cone(a, [0, 0, 0], 1 - a, [0, 0, 1], a = 0..1)):
```



Example 3

For a pyramid and a frustum of pyramid, the attributes `BaseRadius` and `TopRadius` determine the radius of the circumcircle of its regular base and top:

```
plot(plot::Prism(1,Edges=5), plot::Circle3d(1)):
```



See Also

MuPAD Functions
Base | Top

Cells

Classes of histogram plots

Value Summary

Mandatory	List of arithmetical expressions

Graphics Primitives

Objects	Cells Default Values
`plot::Histogram2d`	[7]

Description

`Cells` determines the number and position of the classes used in a histogram.

`Cells` accepts either a single positive integer (or, equivalently, a list of one positive integer) or a list of cells given as ranges or lists of two elements.

A single integer n in the specification `Cells = n` or `Cells = [n]` is interpreted as "subdivide the range of data into n cells of equal size."

The number n can be animated. In this case, n may be a symbolic expression of the animation parameter.

The cells may be specified directly as in `Cells = [[a_1, b_1], [a_2, b_2], …]` or `Cells = [a_1..b_1, a_2..b_2, Symbol::dots]`.

Note: The i-th cell is the semi-open interval $(a_i, b_i]$, i.e., a datum x is tallied into the i-th cell if $a_i < x \le b_i$ is satisfied.

The cell boundaries must satisfy $a_1 < b_1 \le a_2 < b_2 \le a_3 < \ldots$. In most applications, $b_1 = a_2$, $b_2 = a_3$ etc. is appropriate.

If giving cells directly, the leftmost border may be `- infinity` and the rightmost border may be `infinity`. These rectangles will then be adjusted according to the average widths of the other rectangles for display purposes.

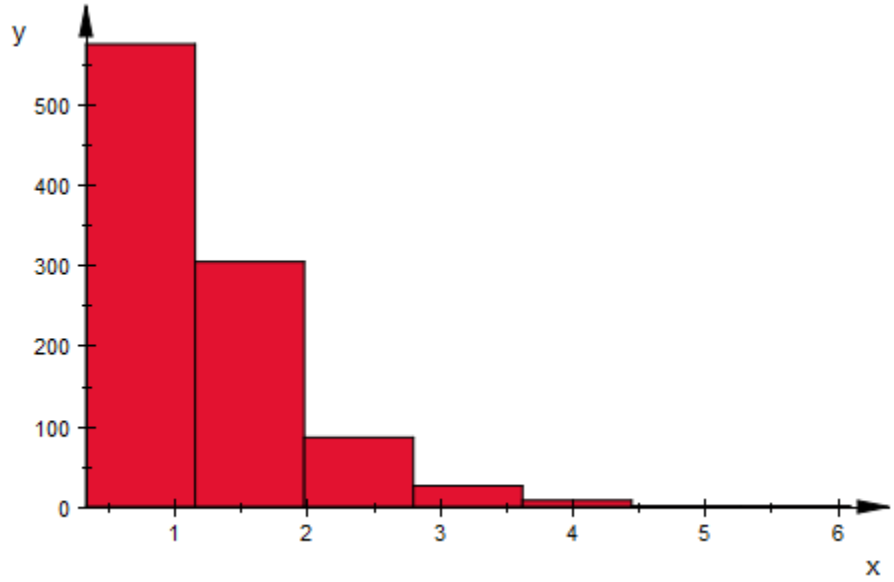
With the attribute `CellsClosed` = `Left`, the cells `[a_1..b_1, a_2..b_2, Symbol::dots]` are interpreted as the semi-open intervals $[a_i, b_i)$.

Examples

Example 1

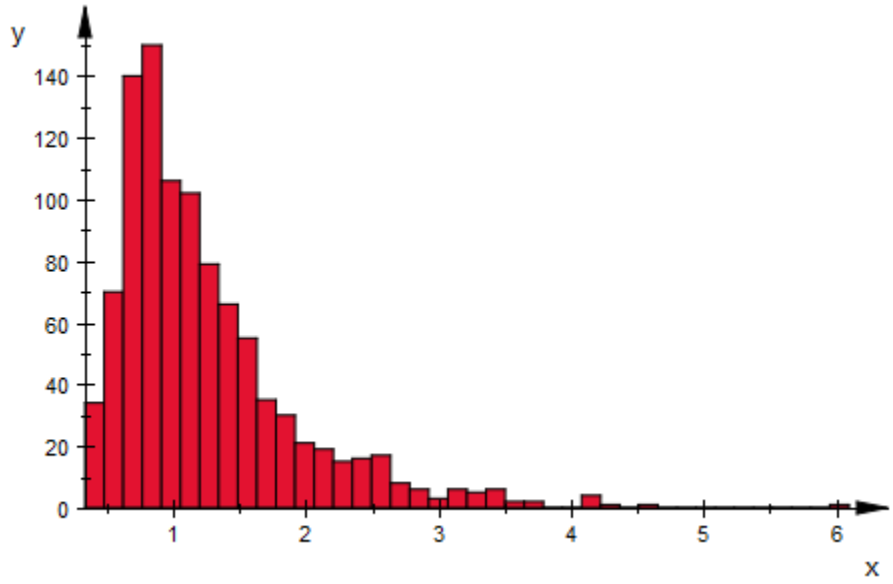
We create a sample of 1000 data points and plot a histogram of them:

```
X := stats::fRandom(100, 10):
data := [X() $ i = 1..1000]:
plot(plot::Histogram2d(data))
```

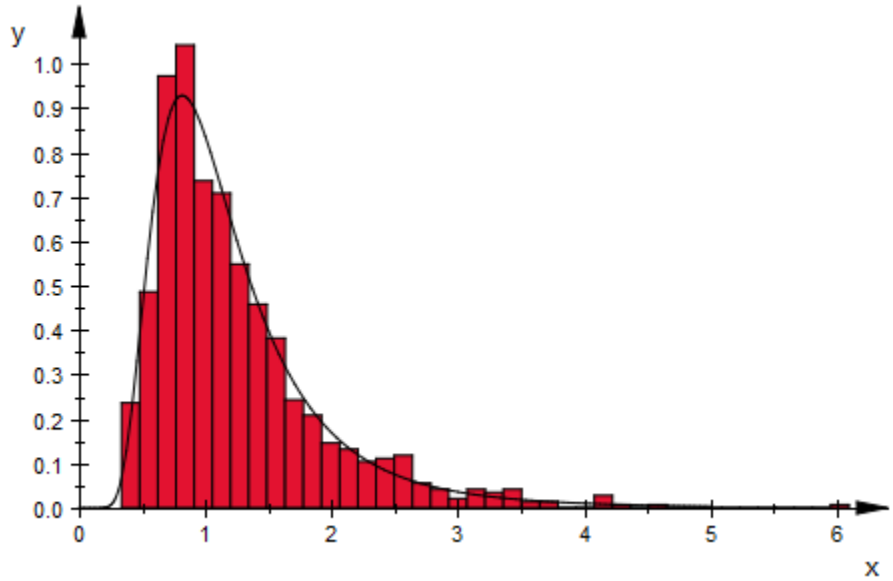


The shape of the distribution becomes much better visible when we increase the number of cells:

```
plot(plot::Histogram2d(data, Cells = 40))
```

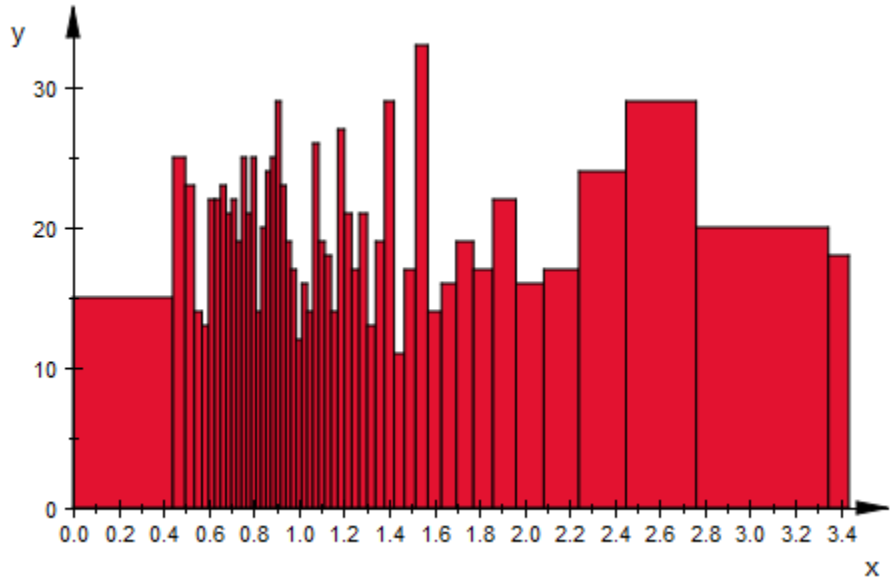



```
plot(plot::Histogram2d(data, Cells = [40], Area = 1),
     plot::Function2d(stats::fPDF(100,10)(x), x = 0 .. 5,
                      Color = RGB::Black))
```

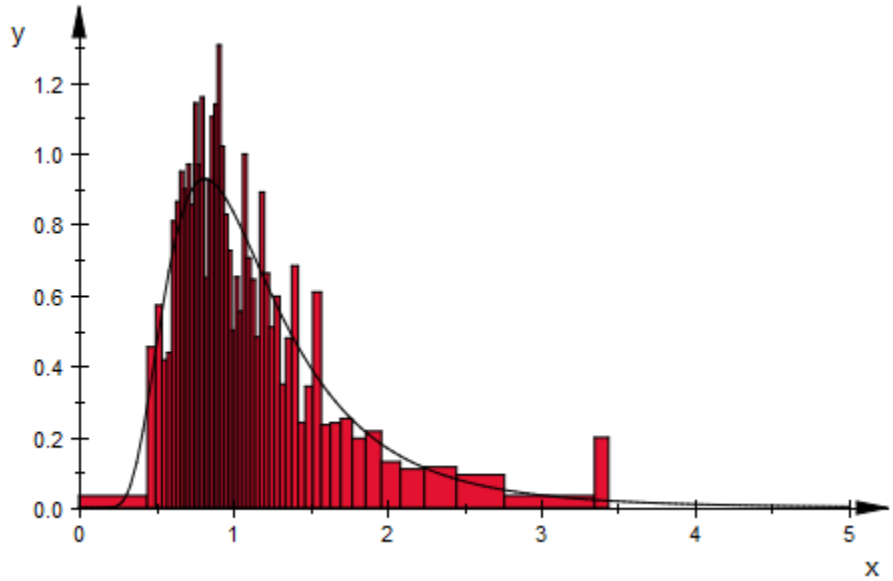


With cells of different widths, setting `Area` to a positive value is highly recommended, to still have the histogram follow the probability distribution:

```
cells :=  stats::equiprobableCells(50,
                  stats::fQuantile(100, 10))

plot(plot::Histogram2d(data, Cells = cells)):
```




```
plot(plot::Histogram2d(data, Cells = cells, Area = 1),
     plot::Function2d(stats::fPDF(100, 10)(x), x = 0 .. 5,
                      Color = RGB::Black))
```



See Also

MuPAD Functions
`Area` | `CellsClosed`

CellsClosed, ClassesClosed

Interpretation of the classes in histogram plots

Value Summary

`CellsClosed`	Optional	`Left`, or `Right`
`ClassesClosed`	[[`CellsClosed`]]	See below

Graphics Primitives

Objects	Default Values
`plot::Histogram2d`	`CellsClosed`, `ClassesClosed`: `Right`

Description

`CellsClosed` determines whether the classes used in a histogram are intepreted as semi-open intervals that are closed at the left or the right boundary.

The graphical primitive plot::Histogram2d tallies numerical data into cells ("classes") that are defined by the attribute `Cells` = `[a_1 .. b_1, a_2 .. b_2, dots]`. By default, these classes are interpreted as a collection of semi-open intervals $(a_i, b_i]$ that are closed at the right boundary. A data item `x` is tallied into the i-th cell if it satisfies $a_i < x \le b_i$. With the option `CellsClosed` = `Left` or the equivalent `ClassesClosed` = `Left` the classes are interpreted as the semi-open intervals $[a_i, b_i)$ that are closed at the left boundary.

Examples

Example 1

We create a sample of 15 data points:

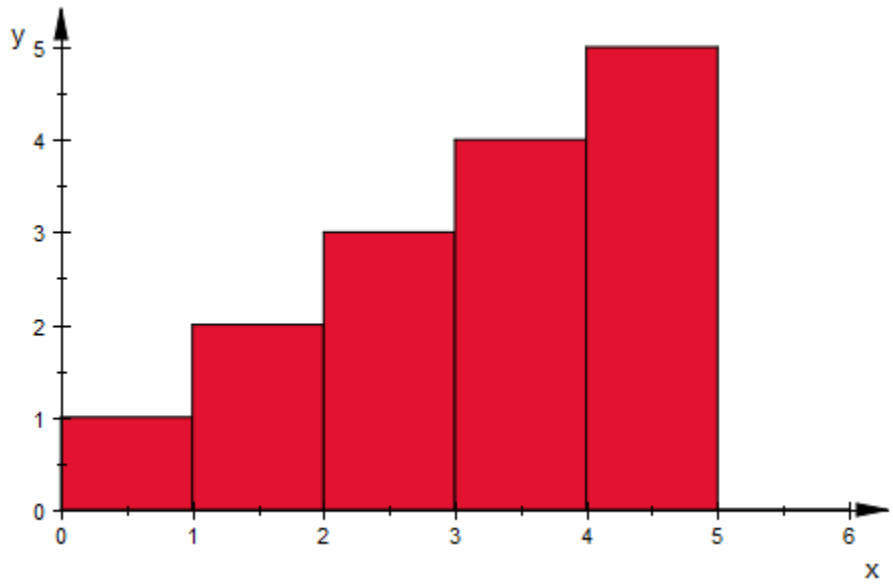
```
data := [1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5]:
```

These data are to be tallied into the following cells (classes):

```
cells := [0 .. 1, 1 .. 2, 2 .. 3, 3 .. 4, 4 .. 5, 5 .. 6]:
```

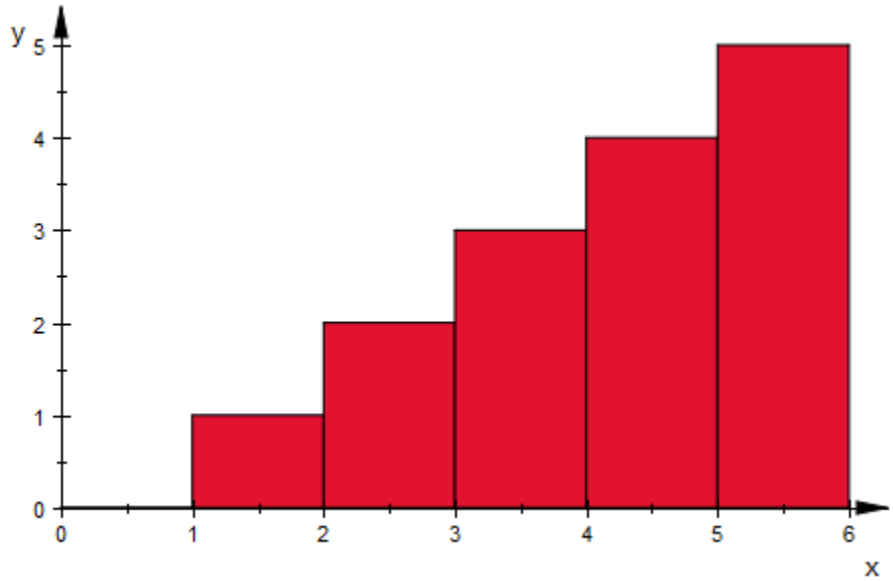
With the default setting `CellsClosed` = `Right`, the 6 classes are the intervals $(0, 1]$, $(1, 2]$ $(2, 3]$ etc. The interval $(0, 1]$ contains one of the data items, the interval $(1, 2]$ contains two, etc.:

```
plot(plot::Histogram2d(data, Cells = cells))
```



Using `CellsClosed` = `Left`, the 6 classes are interpreted as the intervals $[0, 1)$, $[1, 2)$ $[2, 3)$ etc. Now, the first class $[0, 1)$ contains none of the data items, the second class $[1, 2)$ contains one item, etc.:

```
plot(plot::Histogram2d(data, Cells = cells, CellsClosed = Left))
```




```
delete data, cells:
```

See Also

MuPAD Functions
`Area` | `Cells`

Center, CenterX, CenterY, CenterZ

Center of objects, rotation center

Value Summary

`Center`	Library wrapper for "[`CenterX`, `CenterY`]" (2D), "[`CenterX`, `CenterY`, `CenterZ`]" (3D)	List of 2 or 3 expressions, depending on the dimension
`CenterX`, `CenterY`, `CenterZ`	Mandatory	MuPAD expression

Graphics Primitives

Objects	Default Values
`plot::Arc3d`, `plot::Circle3d`, `plot::Dodecahedron`, `plot::Ellipse3d`, `plot::Ellipsoid`, `plot::Hexahedron`, `plot::Icosahedron`, `plot::MuPADCube`, `plot::Octahedron`, `plot::Parallelogram3d`, `plot::Piechart3d`, `plot::Rotate3d`, `plot::Sphere`, `plot::Tetrahedron`, `plot::Waterman`	`Center`: [0, 0, 0] `CenterX`, `CenterY`, `CenterZ`: 0
`plot::Arc2d`, `plot::Circle2d`, `plot::Ellipse2d`, `plot::Parallelogram2d`, `plot::Piechart2d`, `plot::Rotate2d`	`Center`: [0, 0] `CenterX`, `CenterY`: 0

Description

The vector `Center` determines the center of various objects such a circles, spheres, pie charts etc. In rotation objects, it refers to the center of rotation.

Depending on the dimension of the object, it is given by a list or vector of 2 or 3 components.

`CenterX` etc. refer to the x, y, z components of this vector.

`Center`, `CenterX` etc. also denotes the rotation center in rotation objects of type `plot::Rotate2d` or `plot::Rotate3d`.

The values of these attributes can be animated.

Examples

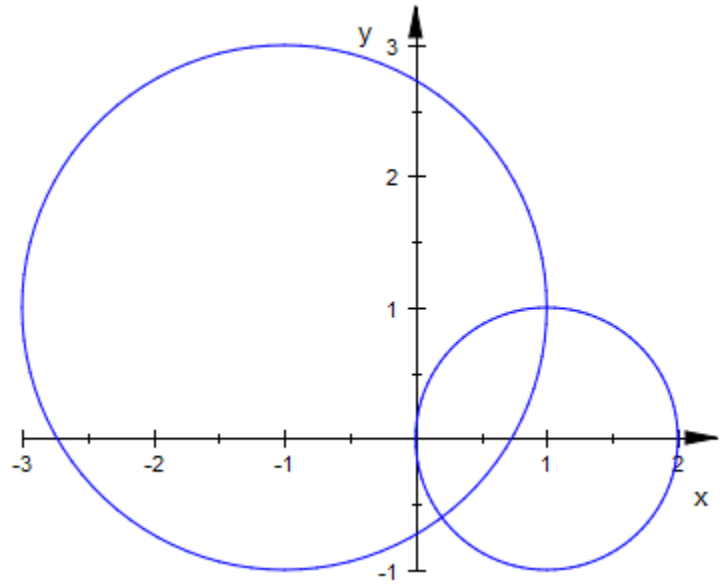
Example 1

We create circles around the origin:

```
c1 := plot::Circle2d(1, [0, 0]):
c2 := plot::Circle2d(2, [0, 0]):
```

The second argument in `plot::Circle2d` is the center. Internally, it is stored as the attribute `Center` and can be changed by assigning a new value:

```
c1::Center := [1, 0]:
c2::Center := [-1, 1]:
plot(c1, c2):
```

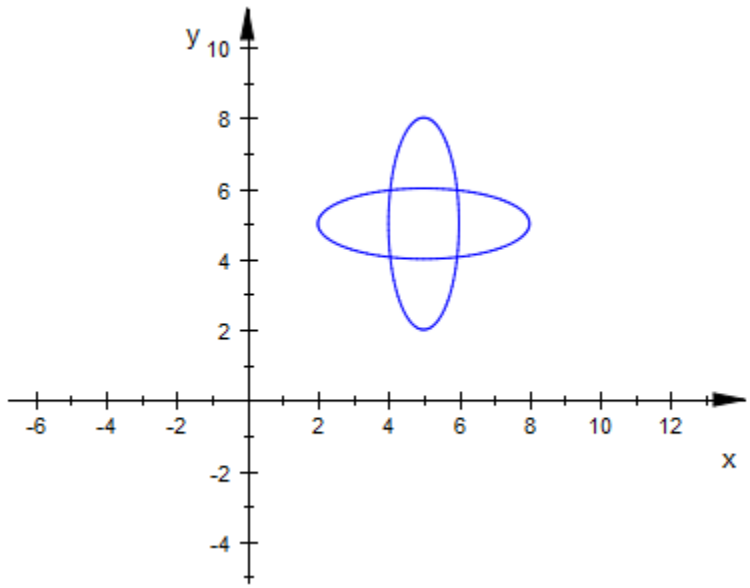



```
delete c1, c2:
```

Example 2

We create an ellipse with an animated center. A copy of it is rotated around this center:

```
e1 := plot::Ellipse2d(1, 3, Center = [a, a], a = 0..5):
e2 := plot::Rotate2d(e1, Angle = a*PI/2,
                     Center = e1::Center, a = 0..5):
plot(e1, e2)
```

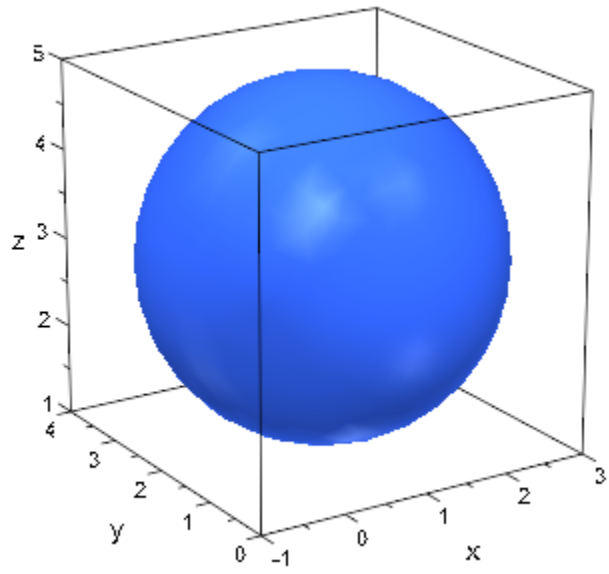



```
delete e1, e2:
```

Example 3

We create a sphere of radius 2 and change the default center `[0, 0, 0]` to `[1, 2, 3]`:

```
s := plot::Sphere(2):
s::Center := [1, 2, 3]:
plot(s)
```




```
delete s:
```

See Also

MuPAD Functions
`Radius` | `SemiAxes`

Closed

Open or closed polygons

Value Summary

Inherited	FALSE, or TRUE

Graphics Primitives

Objects	Closed Default Values
`plot::Arc2d`, `plot::Arc3d`, `plot::Polygon2d`, `plot::Polygon3d`	FALSE

Description

`Closed` switches between open and closed polygons.

`Closed` determines whether objects of type `plot::Polygon2d` and `plot::Polygon3d` are drawn as “real” polygons (i.e., closed) or as broken lines (open polygons).

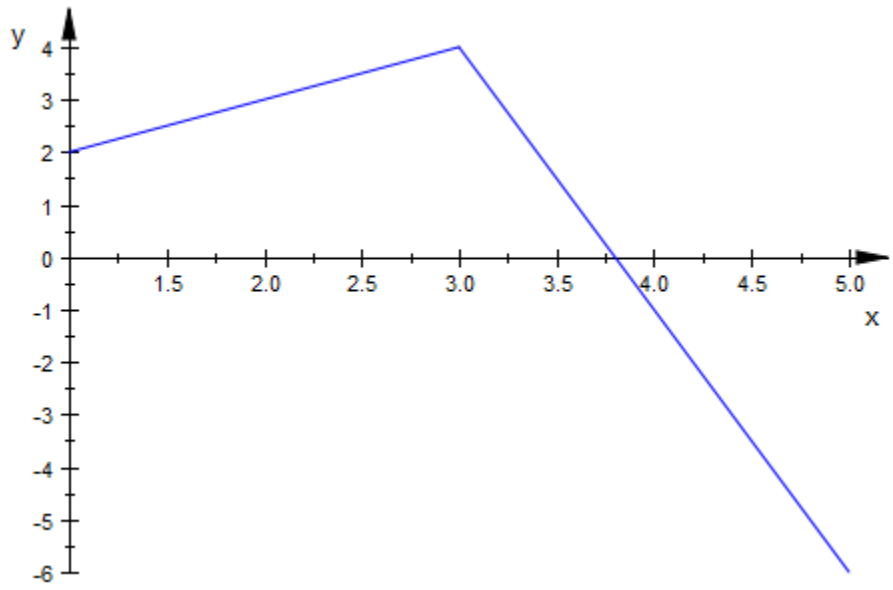
Open polygons can be filled, too. The filled area is exactly the same as if the polygon were closed.

Examples

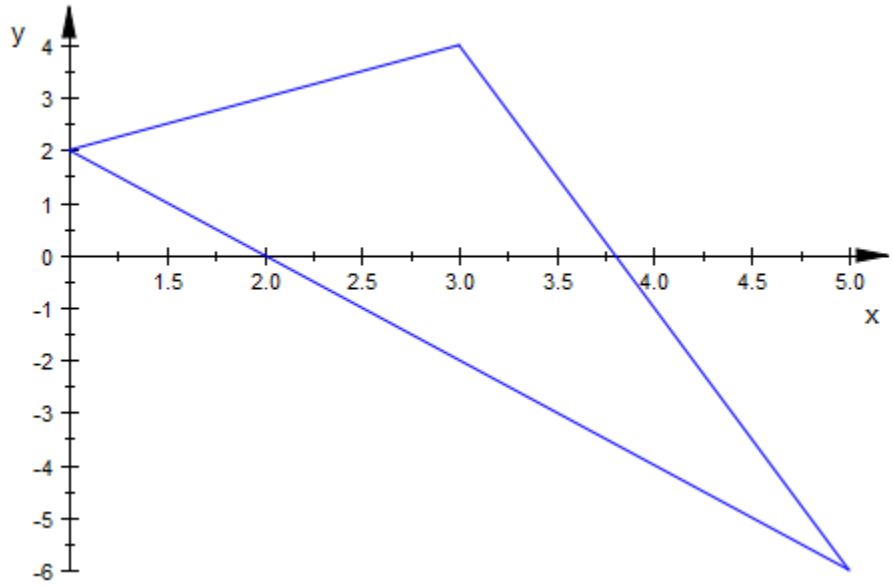
Example 1

By default, polygons are not closed automatically:

```
p := plot::Polygon2d([[1, 2], [3, 4], [5, -6]]):
plot(p)
```

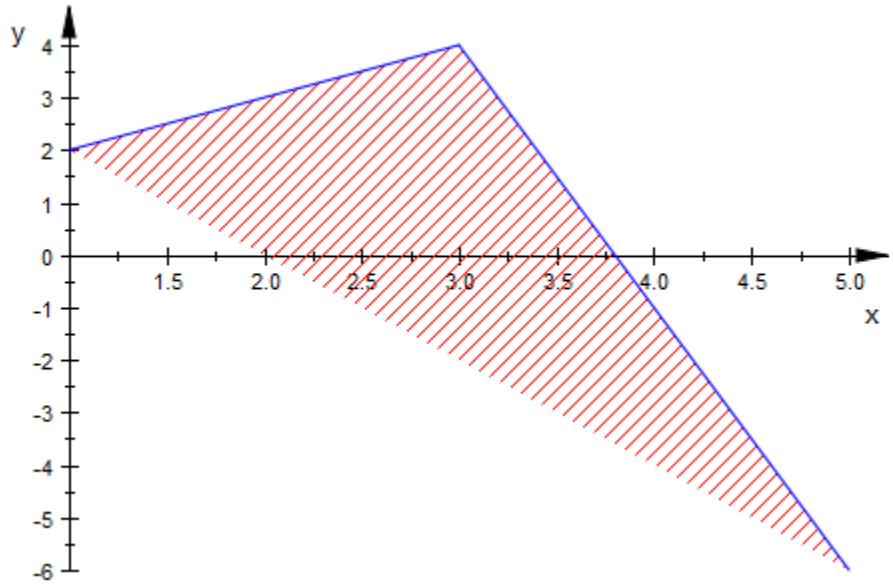



```
p := plot::Polygon2d([[1, 2], [3, 4], [5, -6]], Closed = TRUE):
plot(p)
```



Note that `Filled` and `Closed` are independent:

```
p::Closed := FALSE:
p::Filled := TRUE:
plot(p)
```




```
delete p:
```

See Also

MuPAD Functions
`Filled`

ColorData

Color values of a raster plot

Value Summary

Mandatory	List of arithmetical expressions

Graphics Primitives

Objects	ColorData Default Values
`plot::Raster`	

Description

`ColorData` is a nested list of RGB color values visualized by a `plot::Raster` object.

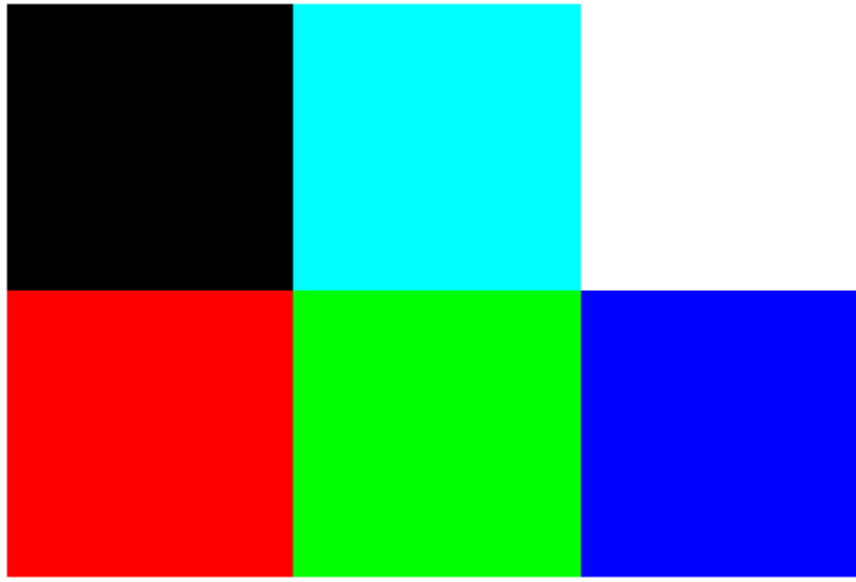
The internal reresentation of the `ColorData` entry of a `plot::Raster` object is a list of lists of color values. Also a matrix or a 2-dimensional array of color values can be assigned to this entry: they are converted to a list of lists.

Examples

Example 1

We create a raster plot object:

```
colordata := [[RGB::Red,   RGB::Green, RGB::Blue ],
              [RGB::Black, RGB::Cyan,  RGB::White]]:
r := plot::Raster(colordata, x = 0..3, y = 0..2):
plot(r):
```



The color data of the raster object can be accessed via the `ColorData` slot:

```
colordata := r::ColorData
```

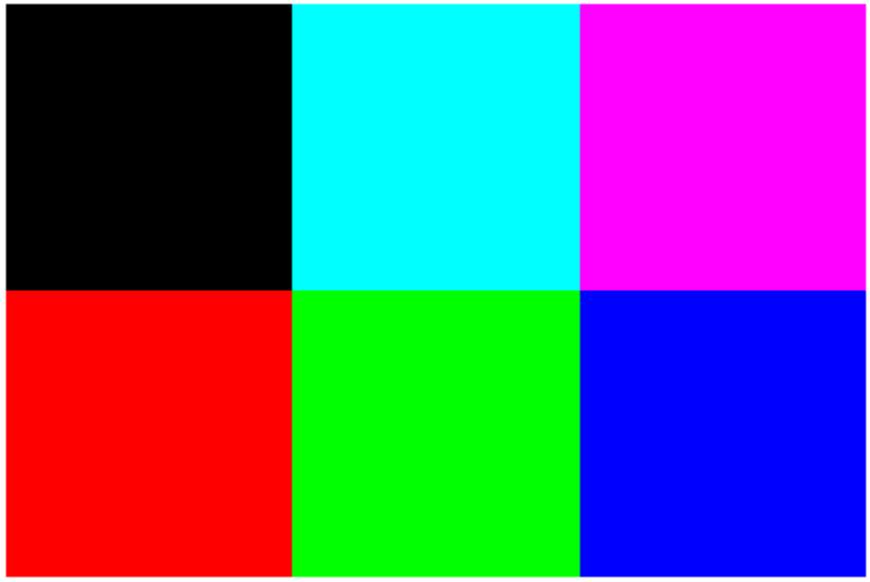
$$[[[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]], [[0.0, 0.0, 0.0], [0.0, 1.0, 1.0], [1.0, 1.0, 1.0]]]$$

The list of list of color values is turned into an array. After changing one entry, the new colors are written back into the raster object:

```
colordata := array(1..2, 1..3, colordata):
colordata[2, 3] := RGB::Magenta:
colordata
```

$$\begin{pmatrix} [1.0, 0.0, 0.0] & [0.0, 1.0, 0.0] & [0.0, 0.0, 1.0] \\ [0.0, 0.0, 0.0] & [0.0, 1.0, 1.0] & [1.0, 0.0, 1.0] \end{pmatrix}$$

```
r::ColorData := colordata:
plot(r)
```



Although the color values were assigned as an array, they are internally stored as a list of lists:

```
r::ColorData
```

```
[[[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]], [[0.0, 0.0, 0.0], [0.0, 1.0, 1.0], [1.0, 0.0, 1.0]]]
```

```
delete colordata, r:
```

CommandList

Turtle movement commands

Value Summary

Mandatory	List of arithmetical expressions

Graphics Primitives

Objects	CommandList Default Values
`plot::Turtle`	[]

Description

`CommandList` stores the command sequence of a `plot::Turtle`. See the documentation of `plot::Turtle` for admissible commands and examples.

Contours

Contours of an implicit function

Value Summary

Mandatory	List of arithmetical expressions

Graphics Primitives

Objects	Contours Default Values
`plot::Implicit2d`, `plot::Implicit3d`	[0]

Description

With `Contours`, you can set the contour(s) of an implicit function.

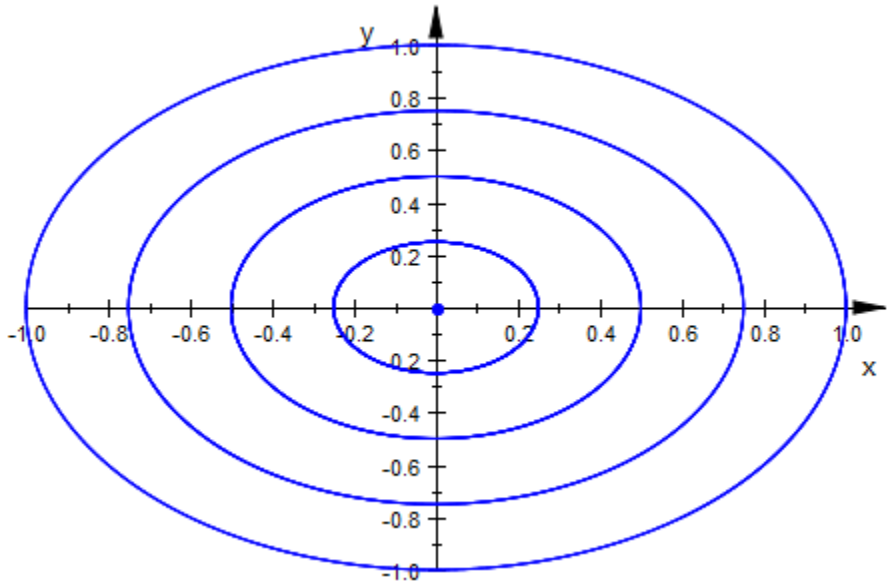
By default, `plot::Implicit2d` and `plot::Implicit3d` plot the set $f^{-1}(0) = \{x \mid f(x) = 0\}$. Using `Contours`, you can instead plot the set $f^{-1}(c)$ for any real c or for a sequence of such values.

Examples

Example 1

The following command plots a series of cuts through a sphere:

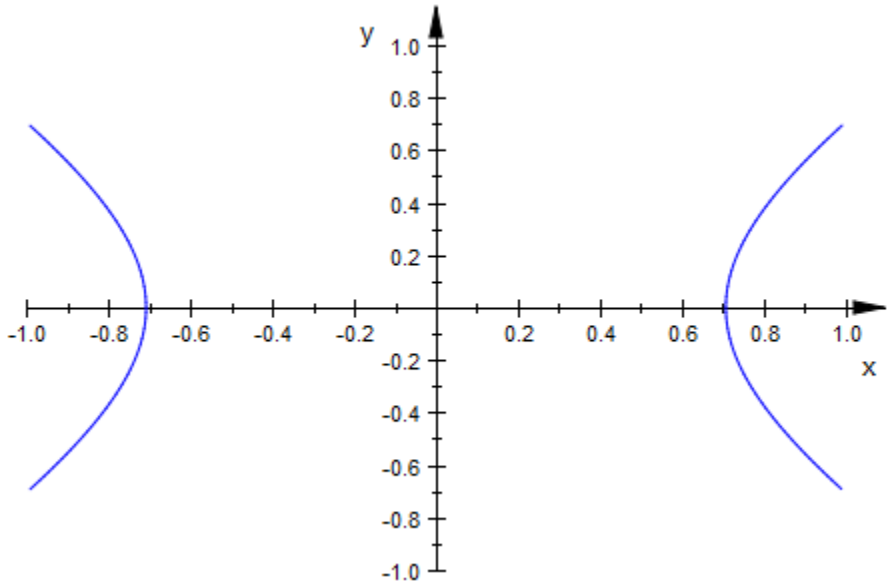
```
plot(plot::Implicit2d(x^2 + y^2, x = -1..1, y = -1..1,
                      Contours = [0, 0.25^2, 0.5^2,
                                  0.75^2, 1.0])):
```



Example 2

Being an expression attribute, `Contours` can be animated:

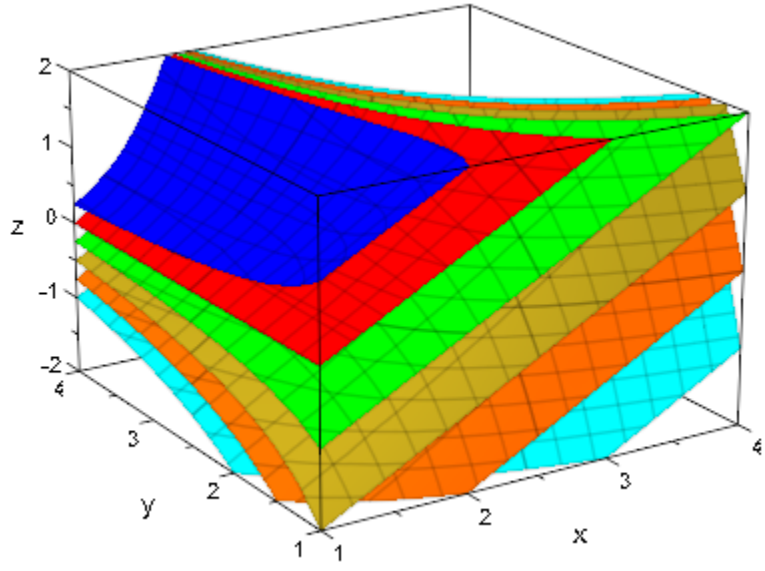
```
plot(plot::Implicit2d(x^2-y^2, x = -1..1, y = -1..1,
                Contours = [1/2*cos(a)], a = 0..2*PI)):
```



Example 3

The contour currently plotted is passed to the color functions and can be used to distinguish them visually:

```
plot(
  plot::Implicit3d(x^y - y*z, x = 1..4, y = 1..4, z = -2..2,
                   Contours = [$0..5],
                   FillColorFunction = ((x,y,z,dx,dy,dz,c) ->
                                RGB::ColorList[round(c)+1]))
    )
```


2
1
0
-1
-2
z
4
3
2
1
y
1
2
3
4
x

CoordinateType

Linear versus logarithmic plots in 2D

Value Summary

Inherited	See below

Graphics Primitives

Objects	CoordinateType Default Values
`plot::CoordinateSystem2d`	`LinLin`

Description

`CoordinateType` allows to switch between linear and logarithmic 2D plots.

By default, a linear (Cartesian) scaling of all coordinate axes is used in 2D. This corresponds to `CoordinateType = LinLin`. Logarithmic plots are created by choosing a `CoordinateType` different from `LinLin`.

In 2D, the following coordinate types are available:

- `LinLin`: Straight lines given by $y = c_1 x + c_2$ are rendered as straight lines on the screen.
- `LinLog`: Linear coordinates are plotted along the horizontal axis, logarithmic coordinates along the vertical axis. The curves $y = e^{c_1 x + c_2}$ are rendered as straight lines on the screen.
- `LogLin`: Logarithmic coordinates are plotted along the horizontal axis, linear coordinates along the vertical axis. The curves $y = c_1 ln(x) + c_2$ are rendered as straight lines on the screen.

- `LogLog`: Logarithmic coordinates are plotted along both axes. The curves $y = c_1 x^{c_2}$ are rendered as straight lines on the screen.

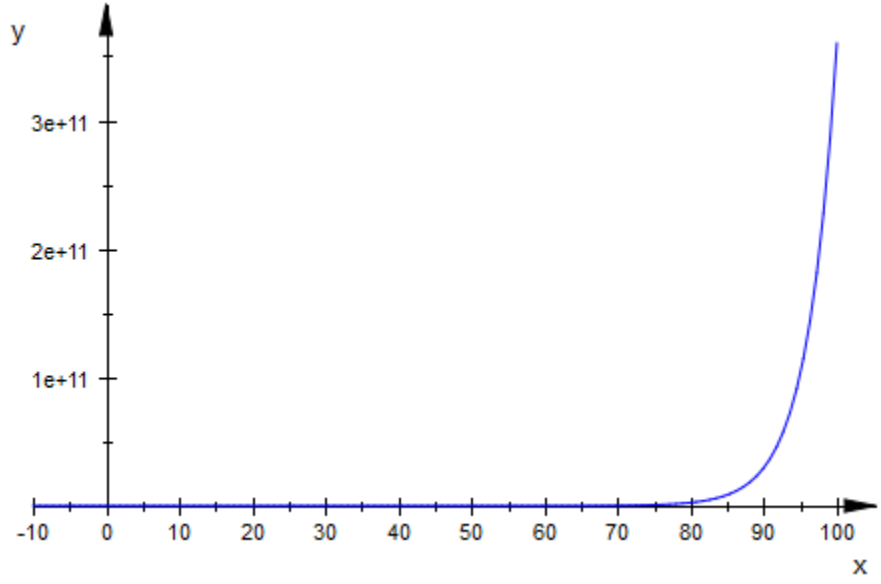
The objects to be plotted must have strictly positive coordinate values in "logarithmic directions".

Examples

Example 1

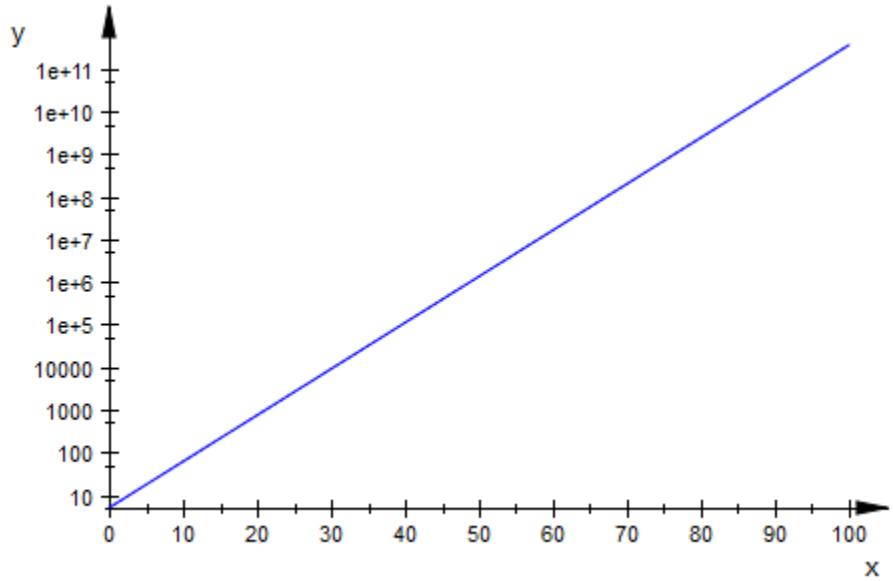
We consider an exponential function:

```
plot(plot::Function2d(5*exp(x/4), x = -10 .. 100),
     CoordinateType = LinLin):
```



In a singly logarithmic plot, the graph is a straight line:

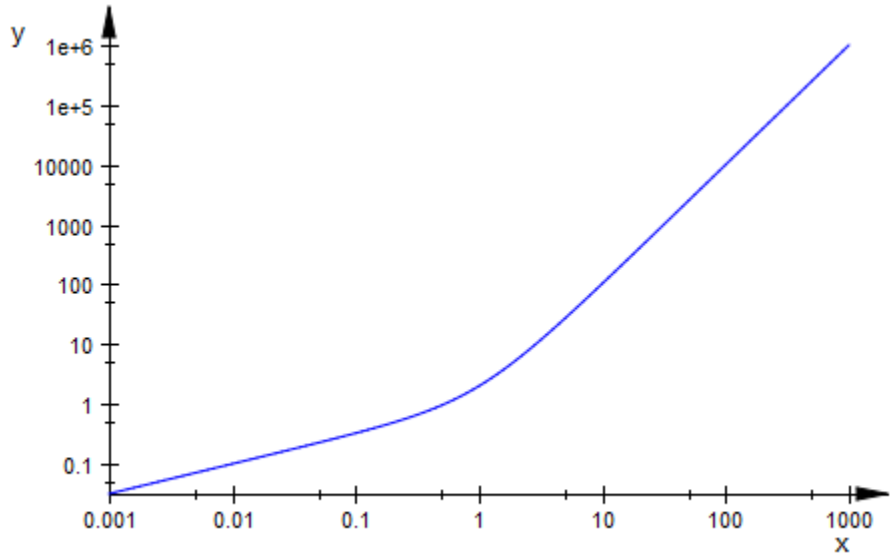
```
plot(plot::Function2d(5*exp(x/4), x = 0 .. 100),
     CoordinateType = LinLog):
```



Example 2

We render the function $y = \sqrt{x} + x^2$ in a log-log plot:

```
plot(plot::Function2d(sqrt(x) + x^2, x = 10^(-3) .. 10^3),
     CoordinateType = LogLog):
```


y
1e+6
1e+5
10000
1000
100
10
1
0.1
0.001
0.01
0.1
1
10
100
1000
x

Data

The (statistical) data to plot

Value Summary

Mandatory	List of arithmetical expressions

Graphics Primitives

Objects	Data Default Values
`plot::Bars2d`, `plot::Bars3d`, `plot::Boxplot`, `plot::Histogram2d`, `plot::Listplot`, `plot::Matrixplot`, `plot::QQplot`, `plot::Scatterplot`, `plot::SparseMatrixplot`	
`plot::Piechart2d`, `plot::Piechart3d`	[1]

Description

`Data` is used internally to store the statistical data displayed, for example, in a pie-chart diagram. While it is possible to manipulate this data (as shown in “Example 2” on page 24-1357), `Data` is mostly seen as a storage space irrelevant to the user.

For speed and clarity, `Data` will be displayed in the object inspector only if the amount of data is small. This may cause problems when using the “recalculate” feature. In such a case, the remedy is to assign the plot object in question to an identifier before plotting.

Examples

Example 1

All object types listed above store the data given in `Data`:

```
X := stats::normalRandom(0, 1):
h := plot::Histogram2d([X() $ i = 1..30])
```

plot::Histogram2d(...)

```
h::Data
```

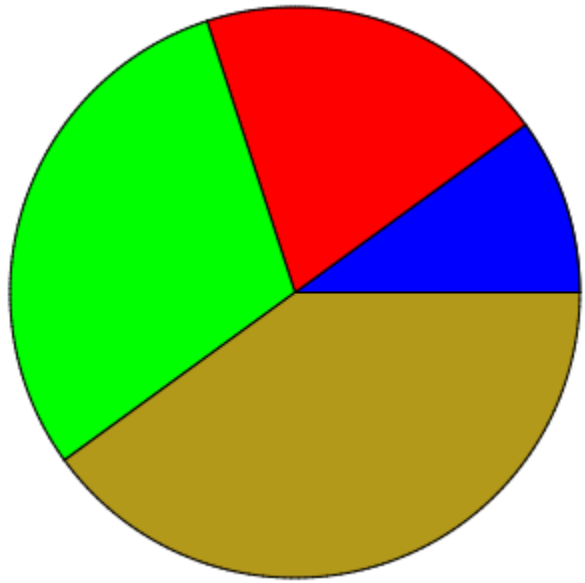
$$[-0.5297400457, -0.5694234147, -0.5161446272, -1.090814471, \ldots, -0.3119111074, 0.1868437371, -0.7818045527]$$

[-0.5297400457, -0.5694234147, -0.5161446272, -1.090814471, 1.782520584, 0.6370330472, 0.6902341601, 0.3399758858, 1.177699186, -0.5970692982, -1.386247581, -0.9783222199, -0.7891413081, 0.2090732178, 0.2186783746, -0.7392138209, 0.6496128588, 0.6258699055, 3.606896706, -0.3319378999, 0.4727169669, 0.4443759372, 0.1735552584, -0.1748302292, -1.468420962, -0.6711676724, 0.6600121852, -0.3119111074, 0.1868437371, -0.7818045527]

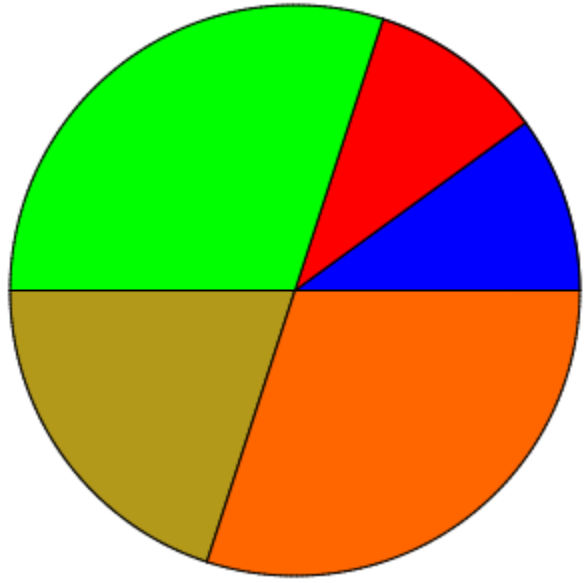
Example 2

It is possible to change the data in an object using `Data`:

```
p := plot::Piechart2d([1, 2, 3, 4]):
plot(p)
```



```
p::Data := [1, 1, 3, 2, 3]:
plot(p)
```



See Also

MuPAD Functions
`Cells`

DensityData, DensityFunction

Density values for a density plot

Value Summary

`DensityData`, `DensityFunction`	Optional	List of arithmetical expressions

Graphics Primitives

Objects	Default Values
`plot::Density`	

Description

`DensityData` is a nested list of "density values" visualized by a `plot::Density` object.

`DensityFunction` is a symbolic expression or a procedure defining the "density values" of a `plot::Density` object.

Density objects of type `plot::Density` can be defined either by discrete density data or by a density function. In the first case, the object has the slot `DensityData`. In the latter case, the function describing the densities is stored in the slot `DensityFunction`.

The internal representation of the `DensityData` entry of a `plot::Density` object is a list of lists of density values. Also a matrix or a 2-dimensional array of density values can be assigned to this entry: they are converted to a list of lists.

The `DensityFunction` of a density object can be a symbolic expression, a procedure or a `piecewise` object.

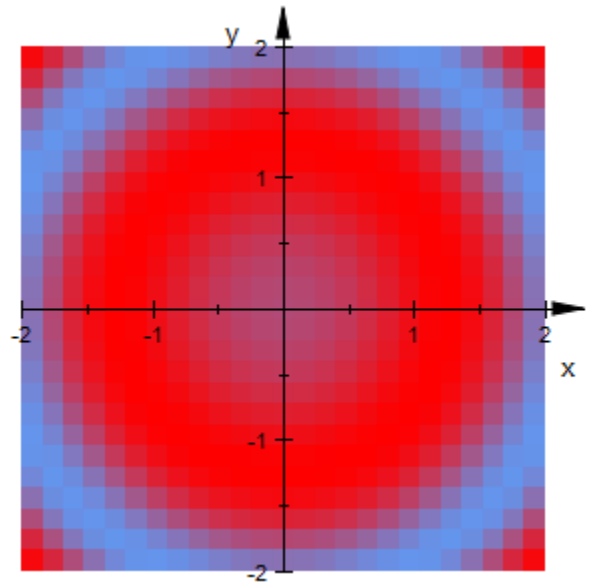
Assigning a value to the `DensityData` entry deletes an existing `DensityFunction` entry and vice versa.

Examples

Example 1

We create a density plot object defined by a density function:

```
d := plot::Density(sin(x^2 + y^2), x = -2..2, y = -2..2):
plot(d, Scaling = Constrained):
```



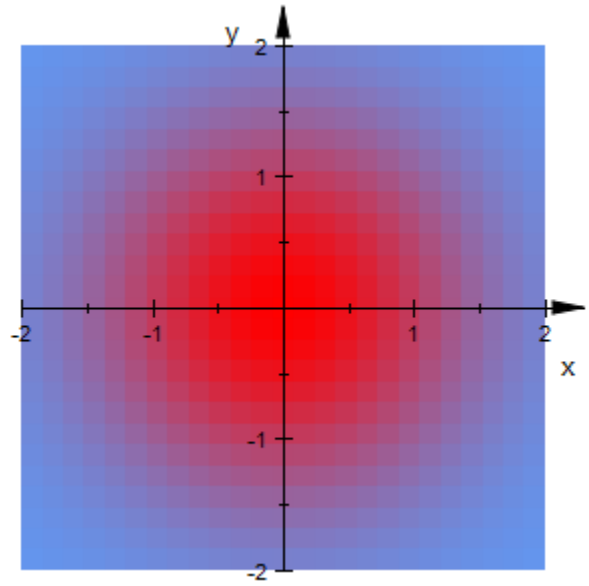
The density function of this object can be accessed via the `DensityFunction` slot:

```
d::DensityFunction
```

$$\sin(x^2 + y^2)$$

We change the density function by assigning a new value to the `DensityFunction` slot:

```
d::DensityFunction := exp(-(x^2 + y^2)/2):
plot(d, Scaling = Constrained)
```

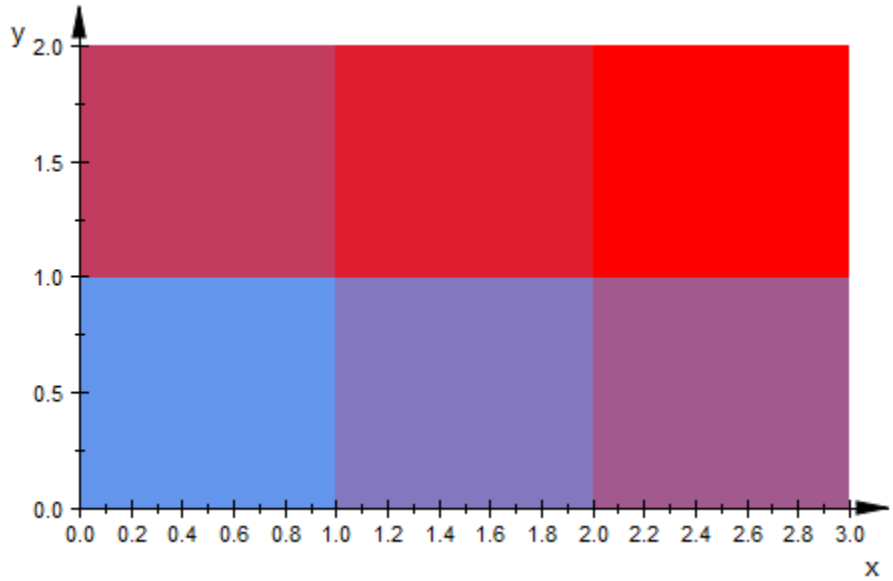



```
delete d:
```

Example 2

We create a density plot object defined by discrete density data:

```
densitydata := [[0.1, 0.2, 0.3],
                [0.4, 0.5, 0.6]]:
d := plot::Density(densitydata, x = 0..3, y = 0..2):
plot(d):
```



The density data of the density object can be accessed via the `DensityData` slot:

```
densitydata := d::DensityData
```

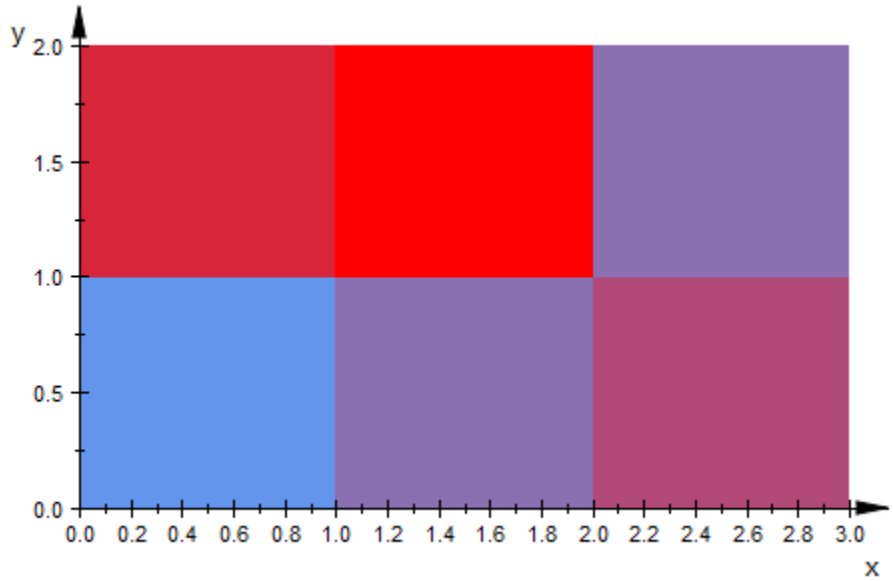
$$[[0.1, 0.2, 0.3], [0.4, 0.5, 0.6]]$$

The list of list of density values is turned into a matrix. After changing one entry, the new density values are written back into the density object:

```
densitydata := matrix(densitydata):
densitydata[2, 3] := 0.2:
densitydata
```

$$\begin{pmatrix} 0.1 & 0.2 & 0.3 \\ 0.4 & 0.5 & 0.2 \end{pmatrix}$$

```
d::DensityData := densitydata:
plot(d)
```



Although the density values were assigned as a matrix, they are internally stored as a list of lists:

```
d::DensityData
```

```
[[0.1, 0.2, 0.3], [0.4, 0.5, 0.2]]
```

```
delete densitydata, d:
```

Edges

Number of Edges

Value Summary

Mandatory	MuPAD expression

Graphics Primitives

Objects	Edges Default Values
`plot::Pyramid`	4
`plot::Prism`	3

Description

`Edges` determines the number of edges for the regular base plane of a prism or pyramid. `Edges` is a positive integer number.

Examples

Example 1

The default values for the attribute `Edges` are:

```
plot::Prism();
plot::Pyramid();
```

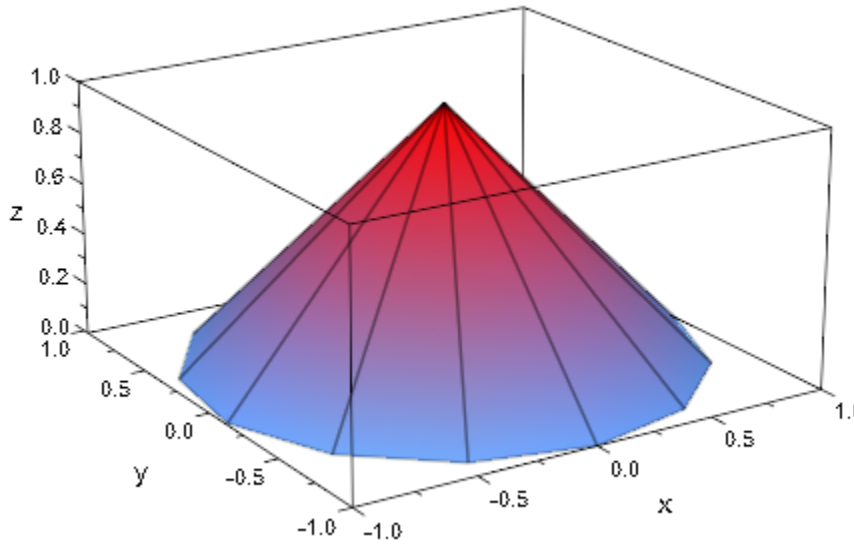
```
plot::Prism(1, [0, 0, 0], [0, 0, 1], Edges = 3)
```

```
plot::Pyramid(1, [0, 0, 0], 0, [0, 0, 1], Edges = 4)
```

Example 2

The attribute `Edges` can be animated:

```
plot(plot::Pyramid(Edges=a, a=3..13)):
```



Extension

Line extensions

Value Summary

Inherited	`Finite`, `Infinite`, or `SemiInfinite`

Graphics Primitives

Objects	Extension Default Values
`plot::Line2d`, `plot::Line3d`	`Finite`

Description

`Extension` allows to extent a line segment to an infinite ray or an infinite line.

Lines of type `plot::Line2d` and `plot::Line3d` are defined by specifying two points through which the line passes. For example: `plot::Line2d([x1, y1], [x2, y2])`. The first point `[x1, y1]` corresponds to the attribute `From`, the second point `[x2, y2]` corresponds to the attribute `To`.

With `Extension` = `Finite`, a line segment from `From` to `To` is drawn.

With `Extension` = `SemiInfinite`, an infinite ray is drawn starting at `From` passing through `To`. The ray extends to the border of the `ViewingBox`.

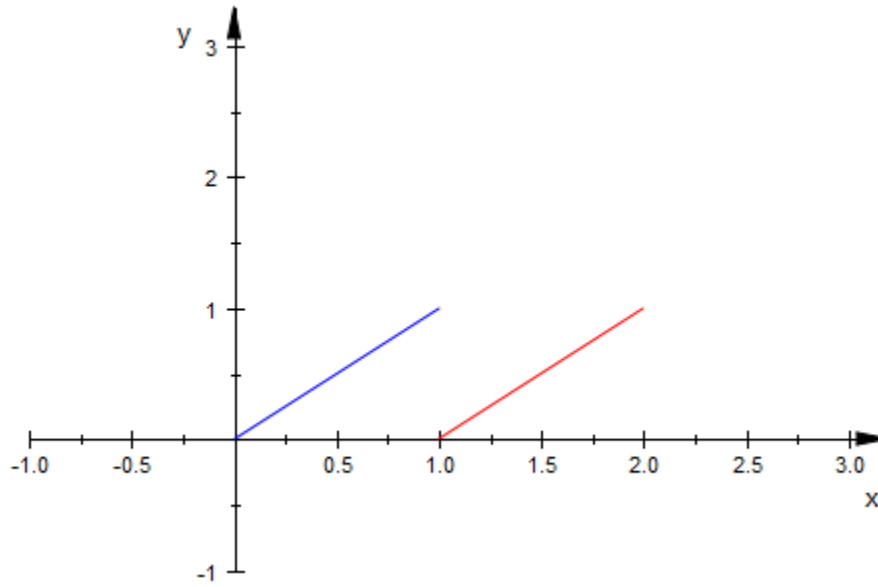
With `Extension` = `Infinite`, an infinite line is drawn passing through `From` and `To`. The line extends in both directions to the border of the `ViewingBox`.

Examples

Example 1

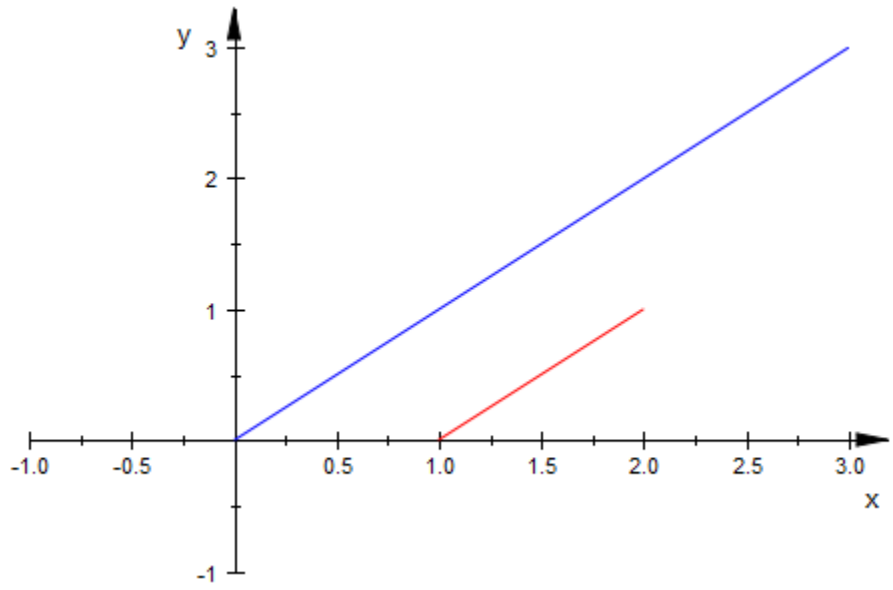
We plot two lines with the default value `Extension` = `Finite`:

```
plot(plot::Line2d([0, 0], [1, 1], Color = RGB::Blue),
     plot::Line2d([1, 0], [2, 1], Color = RGB::Red),
     ViewingBox = [-1..3, -1..3])
```



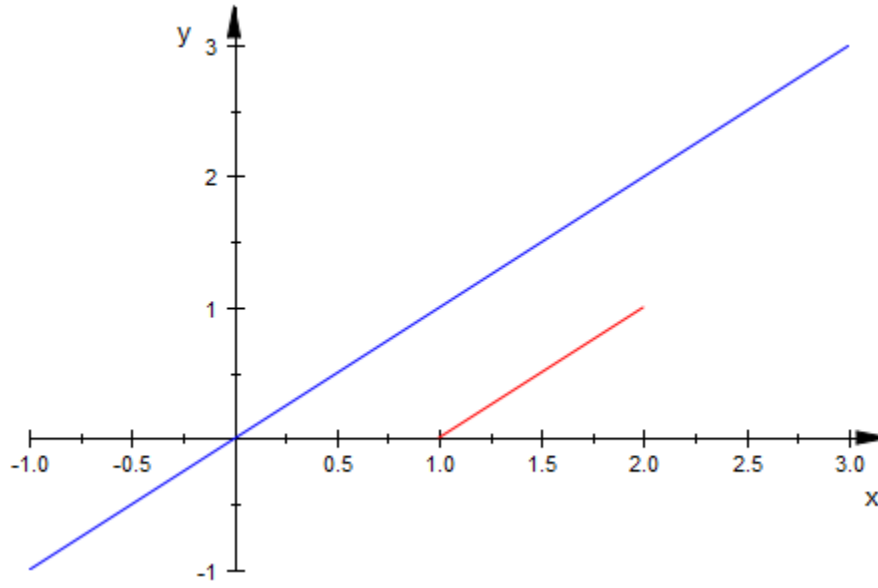
Now, with `Extension` = `SemiInfinite`, the blue line becomes a ray extending to the `ViewingBox` in one direction:

```
plot(plot::Line2d([0, 0], [1, 1], Color = RGB::Blue,
                  Extension = SemiInfinite),
     plot::Line2d([1, 0], [2, 1], Color = RGB::Red),
     ViewingBox = [-1..3, -1..3])
```



With `Extension` = `Infinite`, the blue line extends to the `ViewingBox` in both directions:

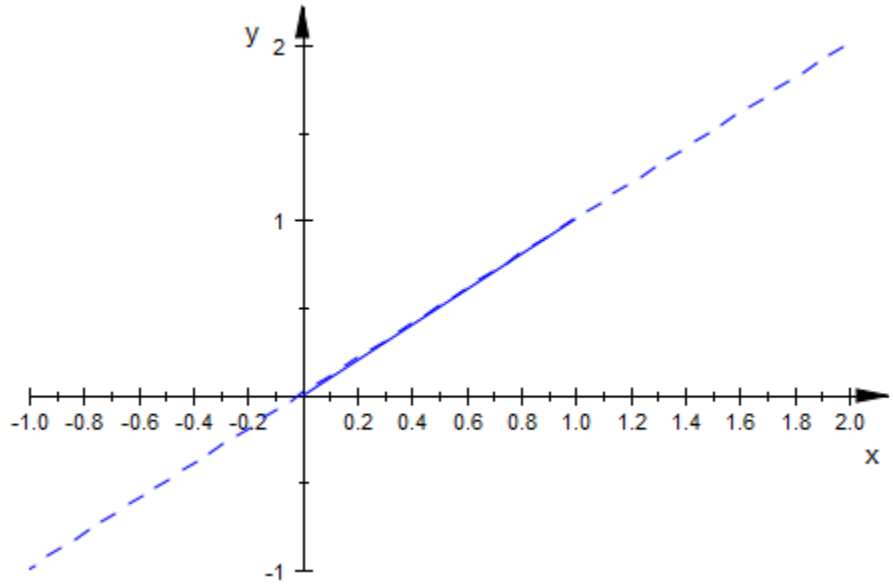
```
plot(plot::Line2d([0, 0], [1, 1], Color = RGB::Blue,
                  Extension = Infinite),
     plot::Line2d([1, 0], [2, 1], Color = RGB::Red),
     ViewingBox = [-1..3, -1..3])
```



Example 2

Here we define a finite line segment and use `plot::modify` to create an extended copy. It is drawn as an infinite dashed line:

```
line := plot::Line2d([0, 0], [1, 1]):
plot(plot::modify(line, Extension = Infinite,
                  LineStyle = Dashed),
     line, ViewingBox = [-1..2, -1..2]):
```




```
delete line:
```

See Also

MuPAD Functions

`AffectViewingBox` | `From` | `To`

From, To, FromX, FromY, FromZ, ToX, ToY, ToZ

Starting point of arrows and lines

Value Summary

`From`	Library wrapper for "[`FromX`, `FromY`]" (2D), "[`FromX`, `FromY`, `FromZ`]" (3D)	List of 2 or 3 expressions, depending on the dimension
`To`	Library wrapper for "[`ToX`, `ToY`]" (2D), "[`ToX`, `ToY`, `ToZ`]" (3D)	List of 2 or 3 expressions, depending on the dimension
`FromX`, `FromY`, `FromZ`, `ToX`, `ToY`, `ToZ`	Mandatory	MuPAD expression

Graphics Primitives

Objects	Default Values
`plot::Arrow2d`, `plot::Line2d`	`From`: [0, 0] `To`: [1, 0] `FromX`, `FromY`, `ToY`: 0 `ToX`: 1
`plot::Arrow3d`, `plot::Line3d`	`From`: [0, 0, 0] `To`: [1, 0, 0] `FromX`, `FromY`, `FromZ`, `ToY`, `ToZ`: 0 `ToX`: 1
`plot::Reflect2d`	

Description

The vectors `From` and `To` determine the starting point and the end point, respectively, of arrows and lines.

`From` is a vector determining the position of the starting point of arrows and lines. Depending on the dimension, it is given by a list or vector of 2 or 3 components.

`FromX` etc. refer to the x, y, z components of this vector.

`To` is a vector determining the position of the end point of arrows and lines. Depending on the dimension, it is given by a list or vector of 2 or 3 components.

`To` etc. refer to the x, y, z components of this vector.

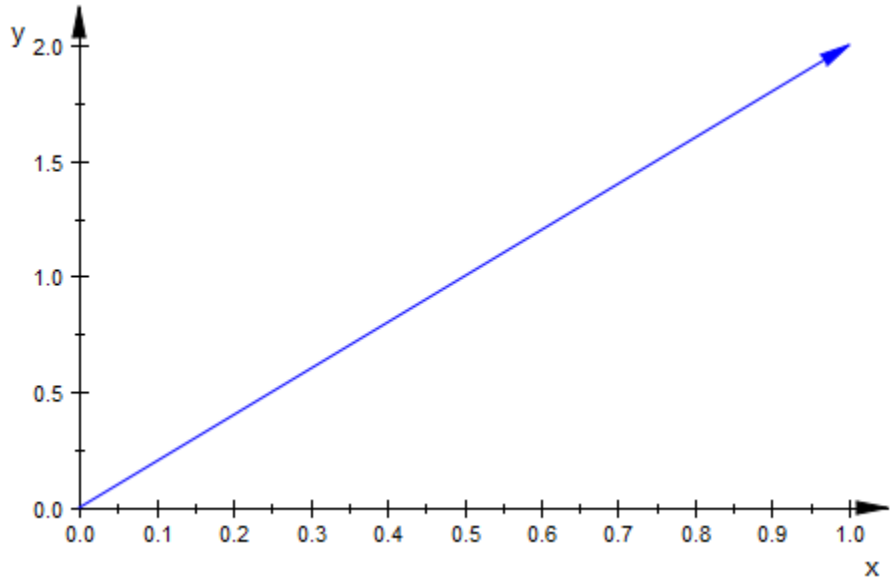
The values of these attributes can be animated.

Examples

Example 1

We define an arrow:

```
p := plot::Arrow2d([0, 0], [1, 2]):
plot(p):
```

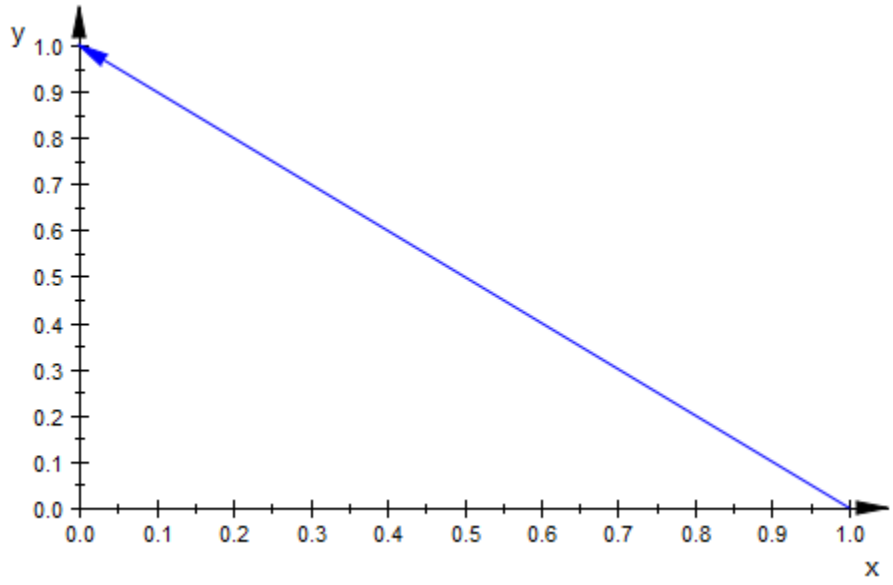


The arguments are the starting point and the end point of the arrow. Internally, they are stored as the attributes `From` and `To`. We can access the object's attributes and change them:

```
p::From, p::To
```

$$[0, 0], [1, 2]$$

```
p::From := [1, 0]:
p::To := [0, 1]:
plot(p):
```

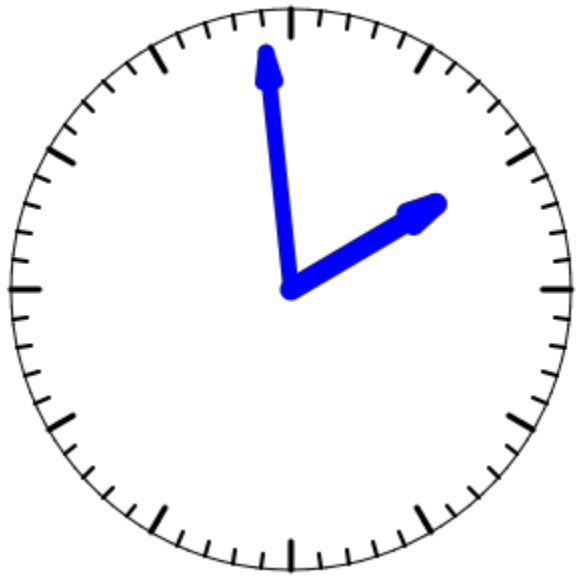



```
delete p:
```

Example 2

The values of `From` and `To` can be animated. Here is a simple clock:

```
plot(plot::Circle2d(1, [0, 0], Color = RGB::Black),
     plot::Line2d([0.9*cos(a*PI/6), 0.9*sin(a*PI/6)],
                  [1.0*cos(a*PI/6), 1.0*sin(a*PI/6)],
                  Color = RGB::Black, LineWidth = 0.8*unit::mm)
                 $ a = 0 .. 11,
     plot::Line2d([0.95*cos(a*PI/30), 0.95*sin(a*PI/30)],
                  [1.0*cos(a*PI/30), 1.0*sin(a*PI/30)],
                  Color = RGB::Black, LineWidth = 0.5*unit::mm)
                 $ a = 0 .. 59,
     plot::Arrow2d([0, 0], [0.85*sin(12*a), 0.85*cos(12*a)],
                   a = 0 .. 2*PI, LineWidth = 2*unit::mm),
     plot::Arrow2d([0, 0], [0.6*sin(a), 0.6*cos(a)],
                   a = 0 .. 2*PI, LineWidth = 3*unit::mm),
     Axes = None, Frames = 600, TimeRange = 0..120):
```



Function, XFunction, YFunction, ZFunction

Function expression or procedure

Value Summary

`Function`, `XFunction`, `YFunction`, `ZFunction`	Mandatory	MuPAD expression

Graphics Primitives

Objects	Default Values
`plot::Conformal`, `plot::Curve2d`, `plot::Curve3d`, `plot::Cylindrical`, `plot::Function2d`, `plot::Function3d`, `plot::Implicit2d`, `plot::Implicit3d`, `plot::Iteration`, `plot::Ode2d`, `plot::Ode3d`, `plot::Polar`, `plot::Sequence`, `plot::Spherical`, `plot::Streamlines2d`, `plot::Sum`, `plot::Surface`, `plot::Tube`, `plot::VectorField2d`, `plot::VectorField3d`, `plot::XRotate`, `plot::ZRotate`	

Description

`Function`, `XFunction`, `YFunction`, `ZFunction` correspond to function expressions or procedures in various plot objects given by a mathematical function.

The attribute `Function` is used for graphs of functions in 2D and 3D, implicit plots, conformal plots etc. which are characterized by a single function.

The attributes `XFunction` etc. refer to the parametrization of the x, y or z-coordinate of parametrized curves and surfaces. In vector field plots they correspond to the components of the vector field.

When defining a graphical primitive such as a function plot, the mathematical expression defining the function is passed directly to the plot routine generating this object. E.g., one calls `plot::Function2d(x*sin(x), x = -5 .. 5)` to define the graph of $f(x) = x\,sin(x)$. Internally, the attribute `Function = x*sin(x)` is associated with the graphical object.

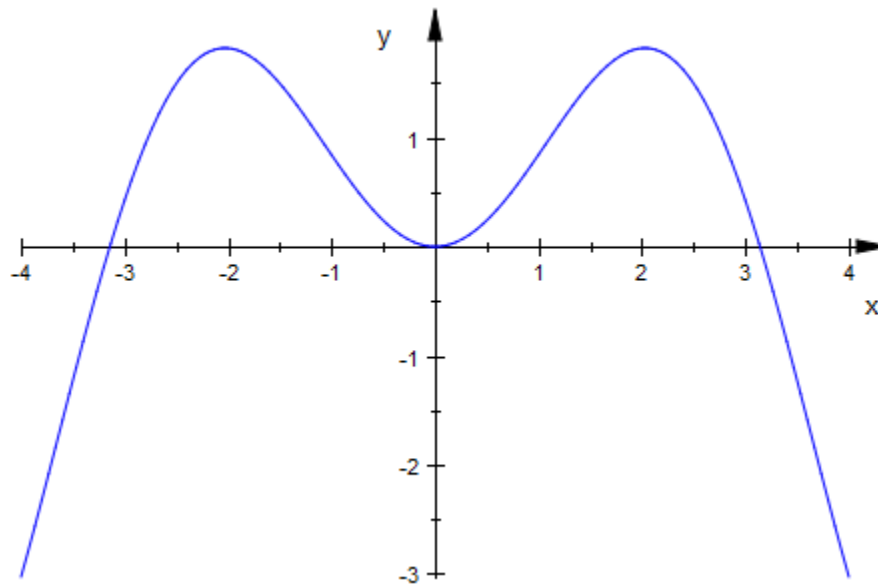
Wherever function expressions are expected, also `piecewise` objects or MuPAD procedures can be used. E.g., the calls `plot::Function2d(sin(x), x = 0..PI)` and `plot::Function2d(x -> sin(x), x = 0..PI)` are equivalent and associate the attributes `Function = sin(x)` or `Function = x -> sin(x)`, respectively, with the plot objects.

Examples

Example 1

We define an object of type `plot::Function2d` representing the graph of $f(x) = x\,sin(x)$:

```
f := plot::Function2d(x*sin(x), x = -4 .. 4):
plot(f)
```



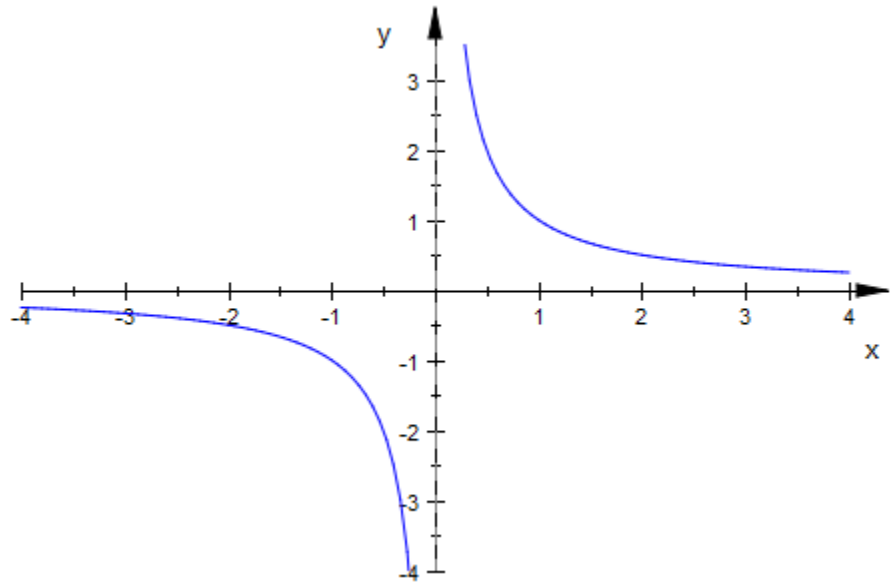
Internally, the expression defining the function is turned into the attribute `Function = x*sin(x)`. It is accessible via a corresponding `slot` of the object:

```
f::Function
```

$$x \sin(x)$$

One can change the object by reassigning a new value to this attribute:

```
f::Function := 1/x:
plot(f):
```




```
delete f:
```

Example 2

For implicit plots as produced by `plot::Implicit2d` and `plot::Implicit3d`, the attribute `Function` refers to the function whose zero set is to be plotted:

```
p := plot::Implicit2d(y*sin(x) - x*cos(y), x=-5..5, y=-5..5):
```

Internally, the expression defining the function is turned into the attribute `Function = y*sin(x) - x*cos(y)`. It is accessible via a corresponding `slot` of the object:

```
p::Function
```

$$y \sin(x) - x \cos(y)$$

```
delete p:
```

Example 3

For parametrized curves and surfaces, the attributes `XFunction`, `YFunction` etc. correspond to the parametrization of the coordinates x, y etc:

```
c2 := plot::Curve2d([u*cos(u), u*sin(u)], u = 0..5*PI):
c2::XFunction, c2::YFunction
```

$$u \cos(u),\ u \sin(u)$$

```
c3 := plot::Curve3d([u*cos(u), u*sin(u), u^2], u = 0..5*PI):
c3::XFunction, c3::YFunction, c3::ZFunction
```

$$u \cos(u),\ u \sin(u),\ u^2$$

```
s := plot::Surface([u*cos(v), u*sin(v), u^2*sin(2*v)],
                   u = 0..1, v = 0..2*PI):
s::XFunction, s::YFunction, s::ZFunction
```

$$u \cos(v),\ u \sin(v),\ u^2 \sin(2\ v)$$

```
delete c2, c3, s:
```

Example 4

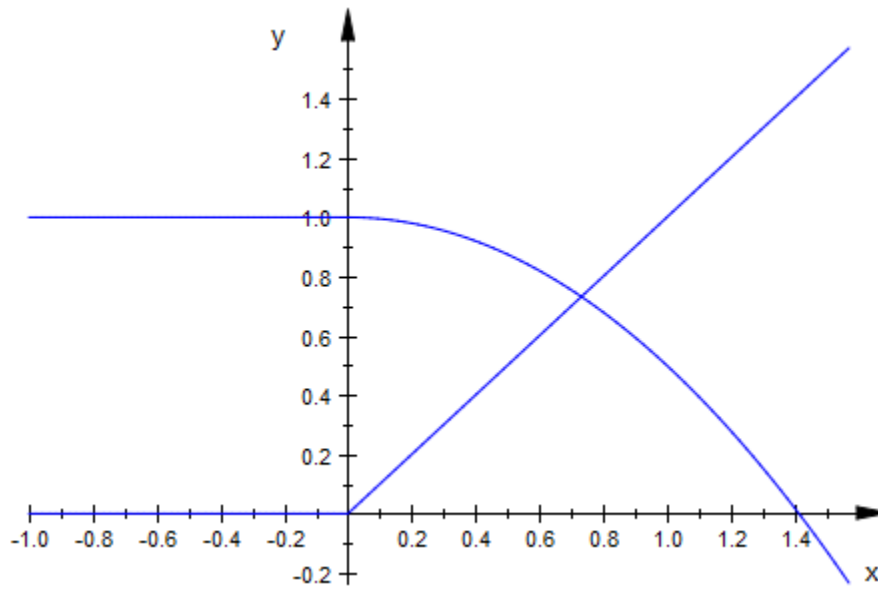
Wherever a function expression is expected, also a `piecewise` object or a procedure can be used:

```
f1 := piecewise([x < 0, 0], [x >= 0, x]):
f2 := proc(x) begin
```

```
        if x < 0 then
             1
        else 1 - x^2/2
        end_if;
      end_proc:
F1 := plot::Function2d(f1, x = -1..PI/2):
F2 := plot::Function2d(f2, x = -1..PI/2):
F1::Function, F2::Function
```

$$\begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } 0 \le x \end{cases}, \text{ proc f2(x) ... end}$$

```
plot(F1, F2)
```




```
delete f1, f2, F1, F2:
```

See Also

MuPAD Functions
Function1 | Function2

Function1, Function2, Baseline

First function/curve delimiting hatch

Value Summary

`Baseline`, `Function1`, `Function2`	Mandatory	Text string

Graphics Primitives

Objects	Default Values
`plot::Hatch`, `plot::Integral`	

Description

`Function1` and `Function2` refer to the functions that define the borders of a hatched 2D region of type `plot::Hatch`.

`Baseline` is the *y* value of a straight horizontal border line of a hatch.

`Function1`, `Function2` are very technical attributes that a user will hardly ever use.

If f_1, f_2 are function objects of type `plot::Function2d`, the hatch object `h := plot::Hatch(`f_1`, `f_2`)` stores references to the objects f_1, f_2 as the slots `h::Function1`, `h::Function2`. These are text references (i.e., strings) by which the function objects f_1, f_2 can be identified, but not the function objects themselves.

`Function1` points to a function object of type `plot::Function2d` or a curve object of type `plot::Curve2d`.

`Function1` is usually set implicitly by `plot::Hatch` to the `Name` attribute of its first argument.

When `Function1` refers to a curve of type `plot::Curve2d`, `Function2` and `Baseline` are ignored.

`Function2` is the (optional) second border function of a hatch. In the plot, the hatched area is bounded by the two functions referred to by `Function1` and `Function2`.

If `Function2` is given, `Function1` must refer to a function graph of type `plot::Function2d`, too.

`Function2` is usually set implicitly by `plot::Hatch` to the `Name` attribute of its second argument.

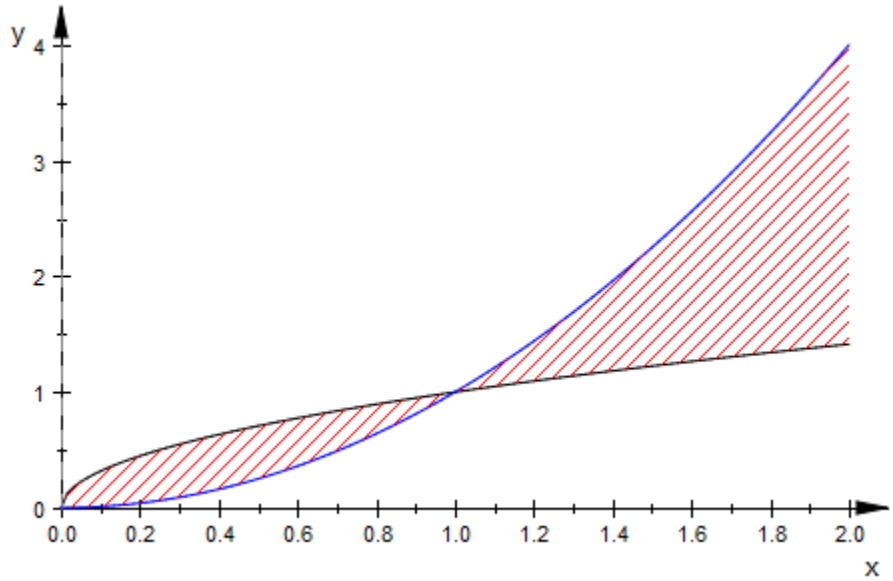
`Baseline` is an alternative second delimiter of a hatch. It defines a horizontal border line of the hatch with a *y*-value given by `Baseline`. The `Baseline` expression may be animated.

Examples

Example 1

We hatch the area between the functions $\sqrt{x}$ and x^2:

```
f1 := plot::Function2d(sqrt(x), x = 0..2, Color = RGB::Black):
f2 := plot::Function2d(x^2, x = 0..2, Color = RGB::Blue):
h := plot::Hatch(f1, f2):
plot(f1, f2, h)
```



The references to the border functions are stored a strings in the hatch object `h`:

```
h::Function1, h::Function2
```

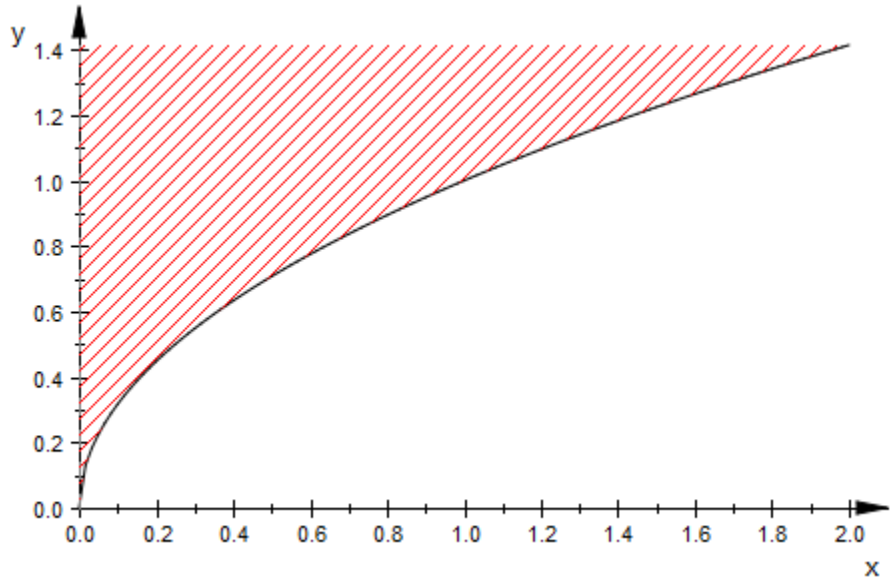
```
"Function2d(LineColor = [0.0, 0.0, 0.0], XMin = 0, XMax = 2, XName = x, Function = x^(1/2), XA\
  xisTitle = "x")", "Function2d(LineColor = [0.0, 0.0, 1.0], XMin = 0, XMax = 2, XName = x, Funct\
  ion = x^2, XAxisTitle = "x")"
```

`Baseline` serves as an alternative for the special case of a constant border function. The `Baseline` value can be animated:

```
h := plot::Hatch(f1, sqrt(2)*a, a = 0 .. 1):
h::Baseline
```

$$\sqrt{2}\, a$$

```
plot(f1, h)
```

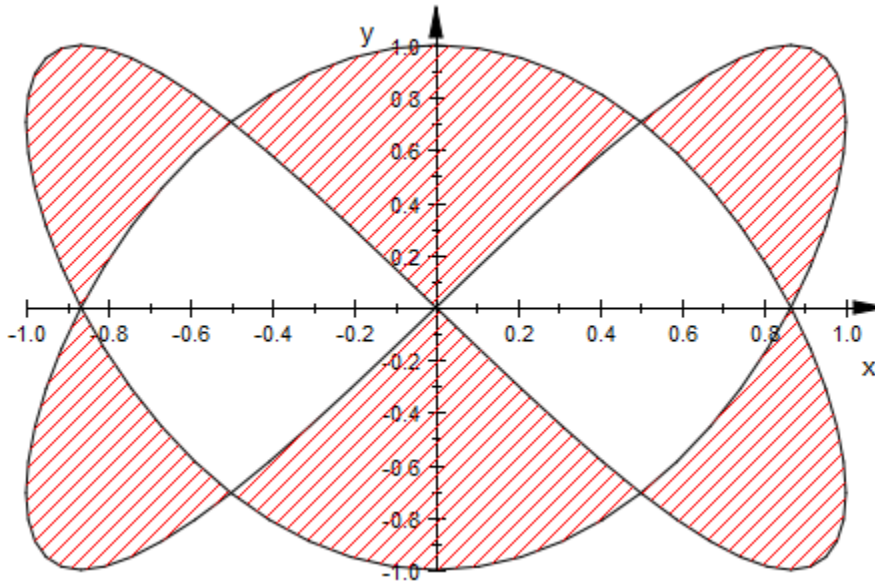


A (closed) curve of type `plot::Curve2d` may be used as the boundary of the hatch:

```
f1 := plot::Curve2d([sin(2*x), cos(3*x)], x = 0..2*PI,
                    Color = RGB::Black):
h := plot::Hatch(f1):
h::Function1
```

```
"Curve2d(LineColor = [0.0, 0.0, 0.0], UMin = 0, UMax = 2*PI, UName = x, XFunction = sin(2*x),\
  YFunction = cos(3*x))"
```

```
plot(f1, h)
```




```
delete f1, f2, h, c:
```

See Also

MuPAD Functions
Name

InitialConditions, TimeMesh

Initial conditions of the ODE

Value Summary

`InitialConditions`, `TimeMesh`	Mandatory	List of arithmetical expressions

Graphics Primitives

Objects	Default Values
`plot::Ode2d`, `plot::Ode3d`	

Description

`InitialConditions = [`$y_1(t_0)$`, `$y_2(t_0)$`, …]` sets the initial conditions for the initial value problem

$$\frac{\partial}{\partial t}\begin{pmatrix} y_1(t) \\ y_2(t) \\ . \\ . \\ . \end{pmatrix} = \begin{pmatrix} f_1(t, y_1(t), y_2(t), \ldots) \\ f_2(t, y_1(t), y_2(t), \ldots) \\ . \\ . \\ . \end{pmatrix}.$$

`TimeMesh = [`t_0`, `t_1`, `t_2`, …]` sets the values of the independent variable t (the "time") of the ODE at which graphical points of the solution curve are plotted. The first entry t_0 is the initial time for which initial conditions are set by `InitialConditions`.

Internally, `plot::Ode2d` and `plot::Ode3d` call the routine `numeric::odesolve` for solving the given ODE numerically.

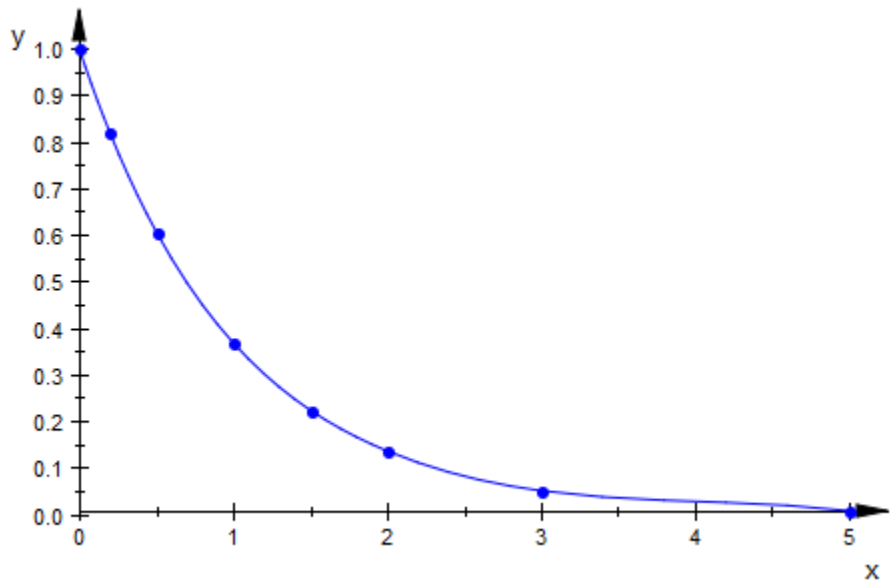
The list of initial conditions set by `InitialConditions` is forwarded to `numeric::odesolve`. See the corresponding help page for further details.

Examples

Example 1

We solve the initial value problem $y'(t) = -y(t)$, $y(0) = 1$ numerically:

```
f := (t, Y) -> [-Y[1]]:
Y0 := [1]:
timemesh:= [0, 0.2, 0.5, 1, 1.5, 2, 3, 5]:
plot(plot::Ode2d(f, InitialConditions = Y0,
                 TimeMesh = timemesh))
```




```
delete f, Y0, timemesh:
```

See Also

MuPAD Functions

`AbsoluteError` | `ODEMethod` | `Projectors` | `RelativeError` | `Stepsize`

IntMethod

Method for integral approximation

Value Summary

Optional	`Exact`, `RiemannLeft`, `RiemannRight`, `RiemannLower`, `RiemannUpper`, `RiemannMiddle`, `RiemannLowerAbs`, `RiemannUpperAbs`, `Simpson`, or `Trapezoid`

Graphics Primitives

Objects	IntMethod Default Values
`plot::Integral`	`Exact`

Description

`IntMethod` determines the method of the visualization of `plot::Integral` objects.

Following methods are implemented:

- Exact

  the area between x-axis and function graph is colored
- RiemannLower

  display boxes between x-axis and function graph using the smallest value of the function in each subinterval
- RiemannLowerAbs

  display boxes between x-axis and function graph using the smallest absolut value of the function in each subinterval

- RiemannUpper

  display boxes between x-axis and function graph using the greatest value of the function in each subinterval

- RiemannUpperAbs

  display boxes between x-axis and function graph using the greatest absolut value of the function in each subinterval

- RiemannLeft

  display boxes between x-axis and function graph using the function value of the left border in each subinterval

- RiemannMiddle

  display boxes between x-axis and function graph using the function value of the middle in each subinterval

- RiemannRight

  display boxes between x-axis and function graph using the function value of the right border in each subinterval

- Trapezoid

  display an approximation of the integral using the Trapezoidal rule

- Simpson

  interpolate the graph of the function using Simpsons rule

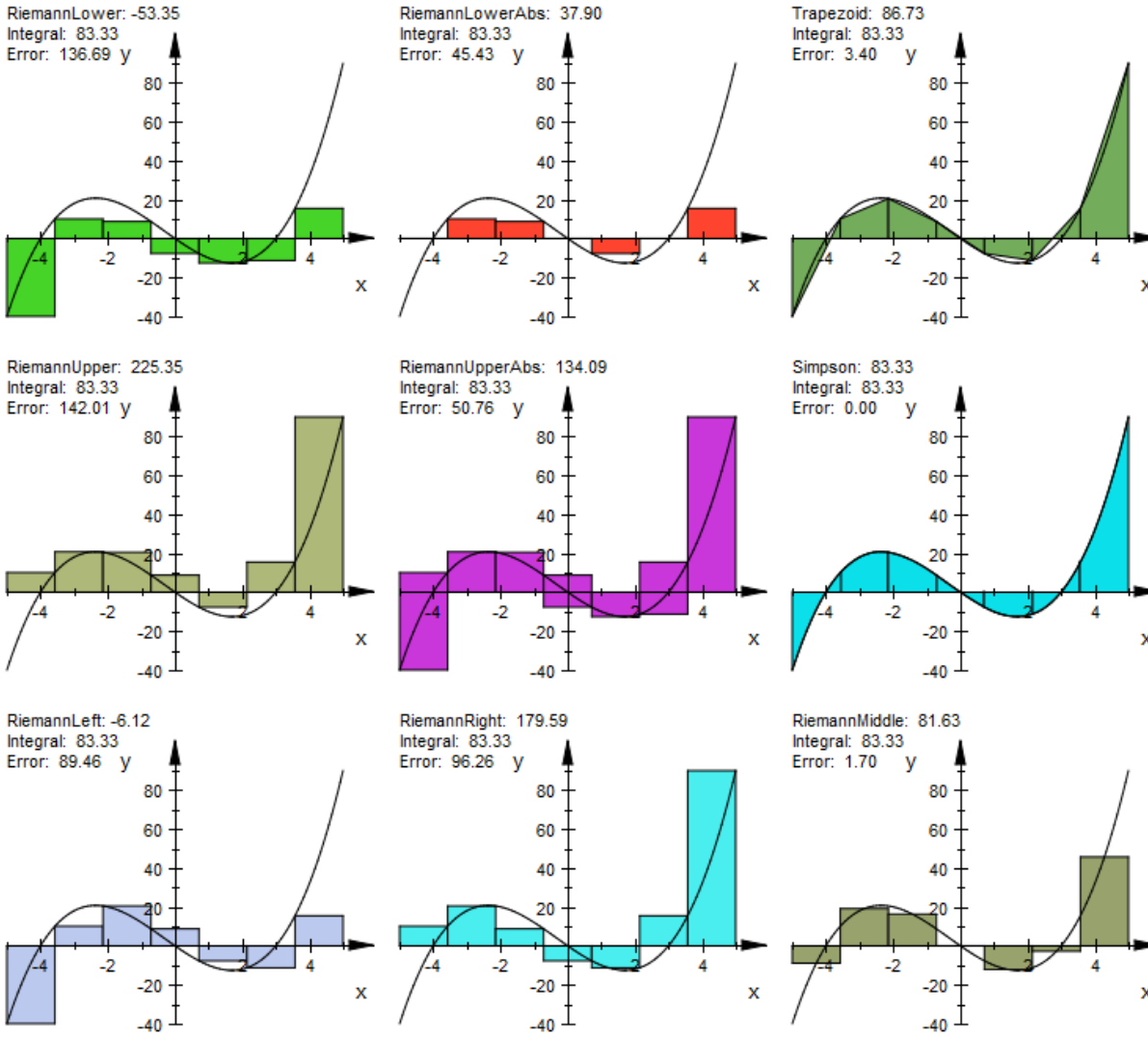
Examples

Example 1

The following example shows all implemented methods:

```
f := plot::Function2d(x*(x-3)*(x+4), Color = RGB::Black):
plot(plot::Scene2d(plot::Integral(f, 7, IntMethod = method,
                                  Color = [frandom() $ i=1..3],
                                  ShowInfo = [IntMethod, Integral,
                                  Error, Position = [-5,90]]), f)
```

```
$ method in [RiemannLower, RiemannLowerAbs, Trapezoid,
             RiemannUpper, RiemannUpperAbs, Simpson,
             RiemannLeft,  RiemannRight,    RiemannMiddle],
Columns = 3, TextFont = [8], Width = 200, Height = 180)
```



Generations, RotationAngle, IterationRules, StartRule, StepLength, TurtleRules

Number of iterations of L-system rules

Value Summary

`Generations`, `IterationRules`, `RotationAngle`, `StartRule`, `StepLength`, `TurtleRules`	Optional	MuPAD expression

Graphics Primitives

Objects	Default Values

`plot::Lsys`	`Generations`: `5`
	
	`StepLength`: `1.0`

Description

`Generations`, `RotationAngle`, `IterationRules`, `StartRule`, `StepLength`, and `TurtleRules` define a Lindenmayer system. The attribute meanings and examples of their use can be found in the documentation of `plot::Lsys`.

Ground

Base value

Value Summary

Optional	MuPAD expression

Graphics Primitives

Objects	Ground Default Values
`plot::Bars3d`, `plot::Sweep`	`0`

Description

In bar charts, the attribute `Ground = g` determines the vertical coordinate value of one end of the bars. Data values $m > g$ are displayed as bars stretching in the vertical direction from the lower end `g` up to the upper end `m`. Data values $m < g$ are displayed as bars stretching in the vertical direction from the upper end `g` down to the lower end `m`.

In sweep surfaces of type `plot::Sweep`, a parametrized space curve $(x(u), y(u), z(u))$ is projected to the x-y-plane with constant $z = g$, where g is set by the `Ground` attribute.

The parameter `g` has to be a numerical real value or an expression of the animation parameter.

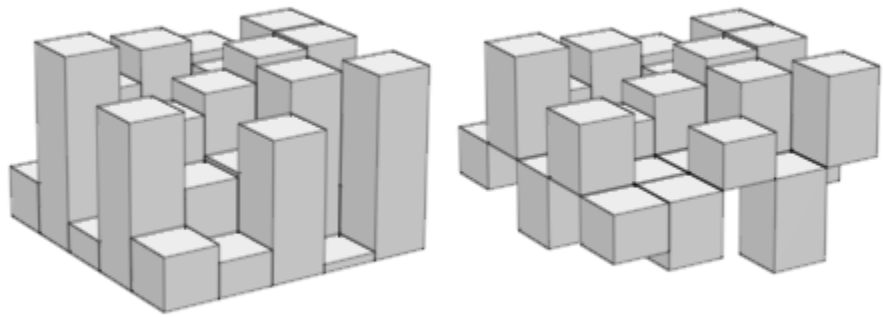
If the attribute `Ground = g` is not specified, the default value `g = 0` is used.

Examples

Example 1

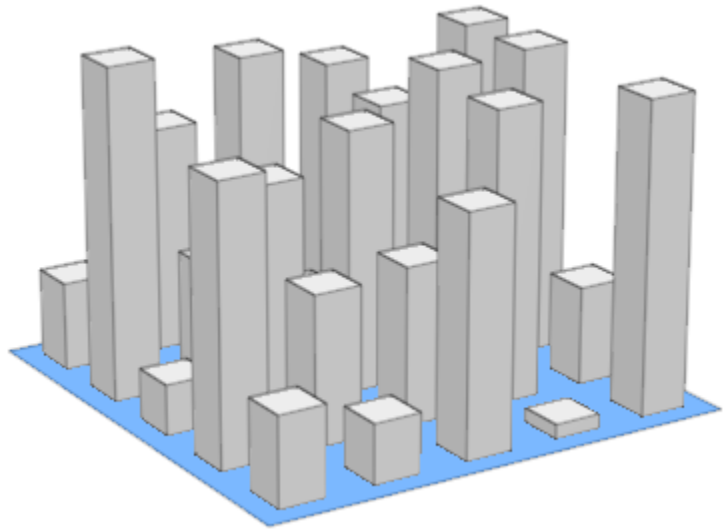
We plot the same data with different `Ground` values:

```
A := matrix::random(5, 5, frandom):
plot(plot::Scene3d(plot::Bars3d(A, Ground = 0,
                                Color = RGB::Grey)),
     plot::Scene3d(plot::Bars3d(A, Ground = 0.5,
                                Color = RGB::Grey)),
     Layout = Horizontal):
```



In the next call, the ground level is animated. Note that in animations one must specify ranges for the *x* and *y* coordinates. We include a transparent plane visualizing the ground level:

```
plot(plot::Bars3d(A, x = 0 .. 1, y = 0 .. 1, a = 0 .. PI,
                  Color = RGB::Grey,
                  Gap = [0.5, 0.5],
                  Ground = sin(a)),
     plot::Surface([x, y, sin(a) + 0.001],
                   x = 0 .. 1, y = 0 .. 1, a = 0 .. PI,
                   Mesh = [2, 2], Color = RGB::Blue.[0.5])
):
```

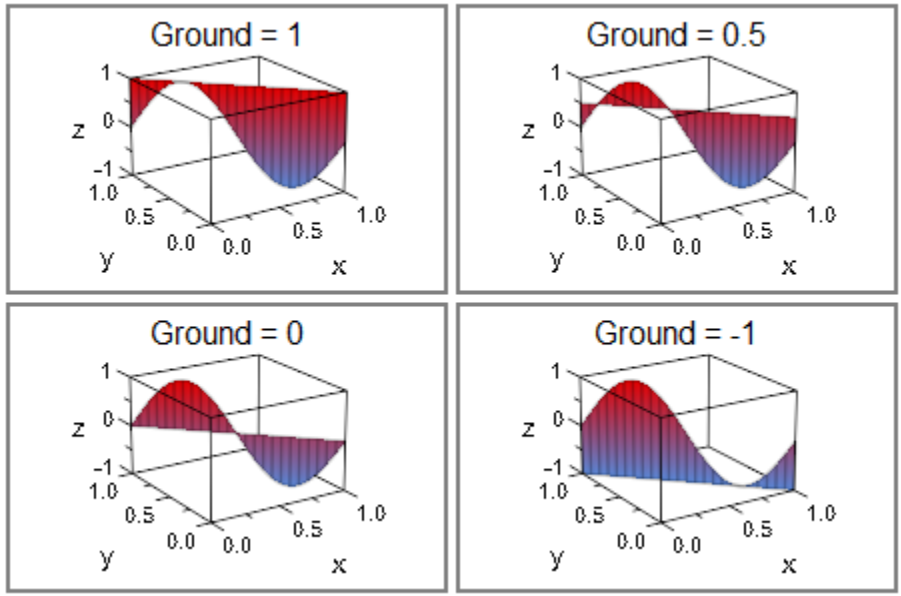


```
delete A:
```

Example 2

We use different `Ground` values to project a space curve to the x-y-plane:

```
plot(plot::Scene3d(plot::Sweep([u, 1-u, sin(2*PI*u)], u = 0..1,
                        Ground =  1), Header = "Ground = 1"),
     plot::Scene3d(plot::Sweep([u, 1-u, sin(2*PI*u)], u = 0..1,
                        Ground = 0.5), Header = "Ground = 0.5"),
     plot::Scene3d(plot::Sweep([u, 1-u, sin(2*PI*u)], u = 0..1,
                        Ground = 0), Header = "Ground = 0"),
     plot::Scene3d(plot::Sweep([u, 1-u, sin(2*PI*u)], u = 0..1,
                        Ground = -1), Header = "Ground = -1"),
     plot::Scene3d::BorderWidth = 0.5*unit::mm,
     Layout = Tabular, Rows = 2)
```


Ground = 1
Ground = 0.5
Ground = 0
Ground = -1
z
y
x
1
0
-1
1.0
0.5
0.0

Heights, Moves

Heights of pieces in pie charts

Value Summary

`Heights`, `Moves`	Optional	List of arithmetical expressions

Graphics Primitives

Objects	Default Values
`plot::Piechart3d`	`Heights`: [`0.3`] `Moves`: [`0`]
`plot::Piechart2d`	`Moves`: [`0`]

Description

`Heights`, `Moves` determine the heights and displacements of the single pieces in a pie chart.

`Heights` determines the heights of the pieces in a `plot::Piechart3d`. If no height value is given for a piece, 0.3 is used. The given values have to be real numbers or expressions of the animation parameter.

`Moves` determines the movements of pieces away from the pie chart center. If no move value is given for a piece, 0 is used. The given values have to be non-negative real numbers or expressions of the animation parameter. The values are fractions of the `Radius` of the pie chart. A value of 1 means a full pie chart radius, 0.5 half the radius of the pie chart etc.

`Heights`, `Moves` accept its input in two formats:

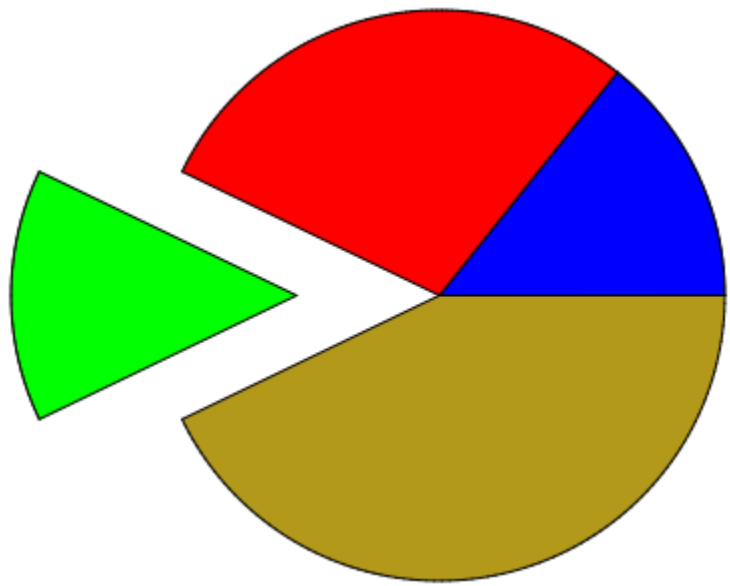
- The values can be given as a flat list of values with entries of the type specified above. The first list entry will be used for the first piece etc. If the list contains less values than the data set of the pie chart, the last value is repeated. Superfluous entries are ignored.
- The values can be given as a list of equations with positive integers on the left hand side and values – as specified above – on the right hand side. The integers are interpreted as indices of the pieces.

Examples

Example 1

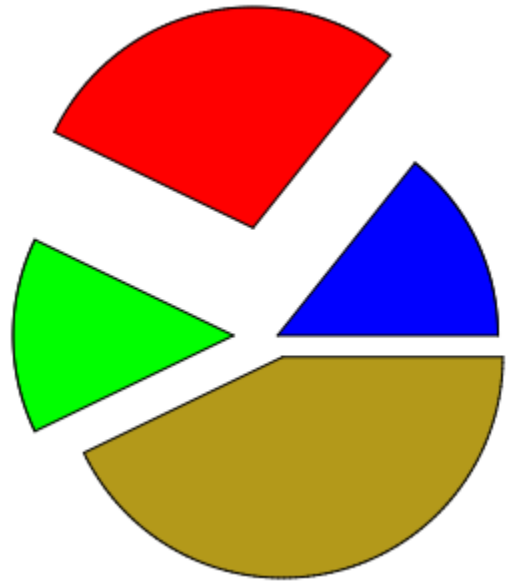
We move the third piece of the following pie chart away from the center by half the radius of the pie chart:

```
plot(plot::Piechart2d([1, 2, 1, 3],
                      Moves = [3 = 0.5]))
```



The pieces are moved away from the center by different amounts:

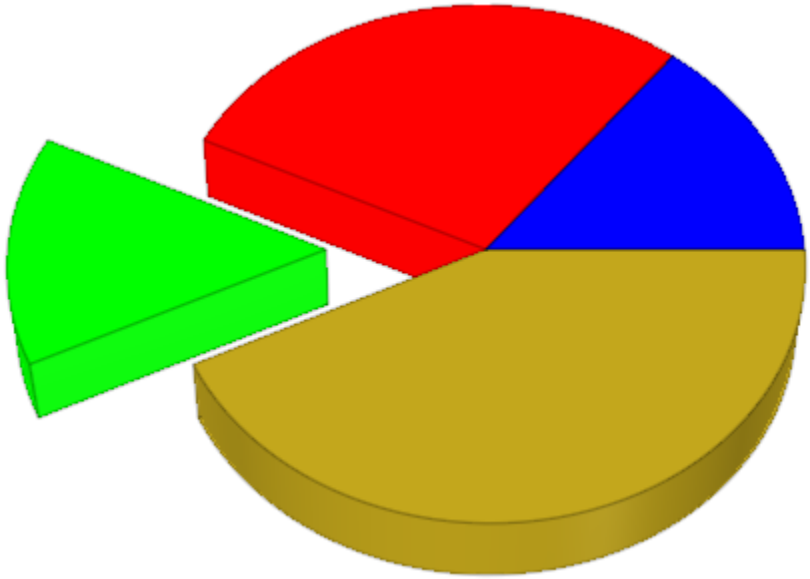
```
plot(plot::Piechart2d([1, 2, 1, 3],
                      Moves = [0, 0.5, 0.2, 0.1]))
```



Example 2

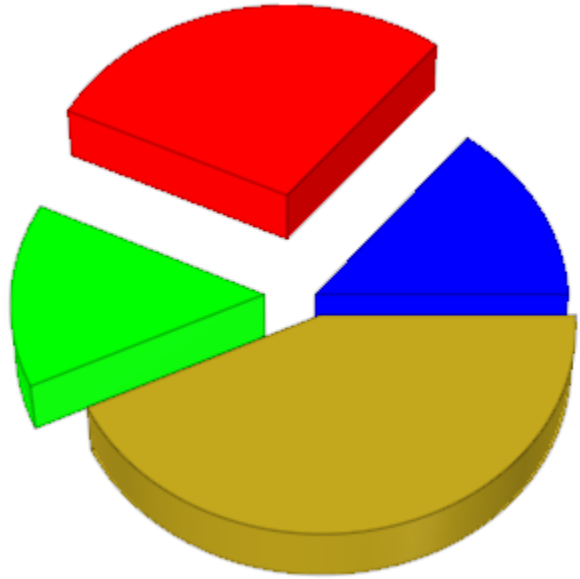
We plot an analogeous 3D pie chart:

```
plot(plot::Piechart3d([1, 2, 1, 3],
                      Moves = [3 = 0.5]))
```



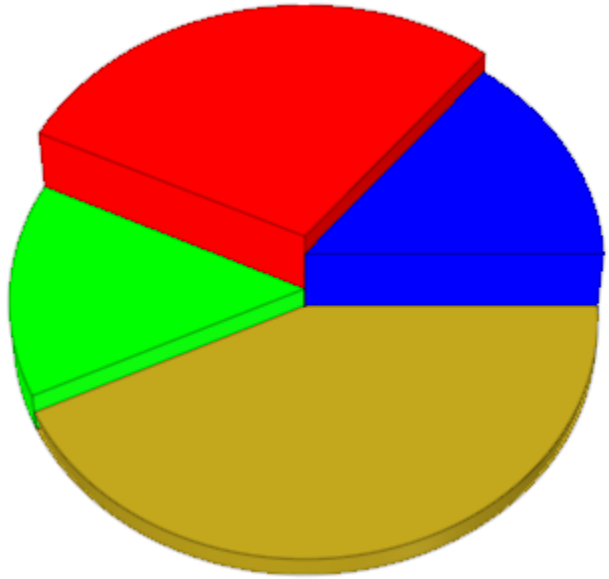
The pieces are moved away from the center by different amounts:

```
plot(plot::Piechart3d([1, 2, 1, 3],
                      Moves = [0, 0.5, 0.2, 0.1]))
```



In 3D, the pieces of a pie chart can have different heights:

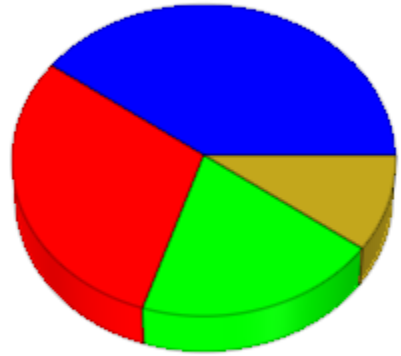
```
plot(plot::Piechart3d([1, 2, 1, 3],
                      Heights = [0.4, 0.5, 0.2, 0.1]))
```



Example 3

Here is a `plot::Piechart3d` with animated `Heights`, `Moves`, and `Radius`:

```
plot(plot::Piechart3d([4, 3, 2, 1],
                      Radius = 3 + sin(a),
                      Heights = [cos(a)^2, cos(2*a)^2,
                                 cos(3*a)^2, cos(4*a)^2],
                      Moves = [0.3*sin(a)^2], a = 0..PI)):
```



See Also

MuPAD Functions
`Data | Radius`

Inequalities

Inequalities displayed in inequality plots

Value Summary

Mandatory	List of arithmetical expressions

Graphics Primitives

Objects	Inequalities Default Values
`plot::Inequality`	

Description

`Inequalities` is the attribute used by `plot::Inequality` to store the inequalities to plot.

`plot::Inequality` is used to plot the areas where one or more inequalities are fulfilled. `Inequalities` is the internal attribute where the inequalities are stored. Most users will never access this attribute directly; it exists for technical reasons only.

InputFile

Input file for import functions

Value Summary

Mandatory	Text string

Graphics Primitives

Objects	InputFile Default Values
`plot::SurfaceSTL`	

Description

`InputFile` specifies the name of the input file for import functions.

`InputFile` can either be an absolute pathname or a pathname relative to the current working directory or one of the directories specified by the MuPAD variable `READPATH`.

Note that some MuPAD functions do not react to the MuPAD variable `READPATH`. In this case it might be necessary to specify `InputFile` as an absolute pathname.

Note that the current working directory of a MuPAD session may depend on how and from where MuPAD has been started. Be careful when making assumptions about this.

See `plot::SurfaceSTL` for examples.

See Also

MuPAD Functions
`OutputFile`

Iterations, StartingPoint

Number of iterations in plot::Iteration

Value Summary

`Iterations`, `StartingPoint`	Optional	MuPAD expression

Graphics Primitives

Objects	Default Values
`plot::Iteration`	`Iterations`: 10

Description

`Iterations` and `StartingPoints` are special attributes for iteration objects of type `plot::Iteration`. `StartingPoint` sets the starting point, `Iterations` sets the number of iteration steps.

The call `it := plot::Iteration(f, x_0, n, x = `x_{min}`..`x_{max}`)` yields a visualization of the iteration $x_i = f(x_{i-1})$ of the starting point x_0 with $i = 1, \ldots, n$. The values x_0 and `n` are stored as the attributes `StartingPoint` = x_0 and `Iterations` = `n` in the iteration object `it`. The values can be accessed and changed as the slots `it::StartingPoint` and `it::Iterations`, respectively.

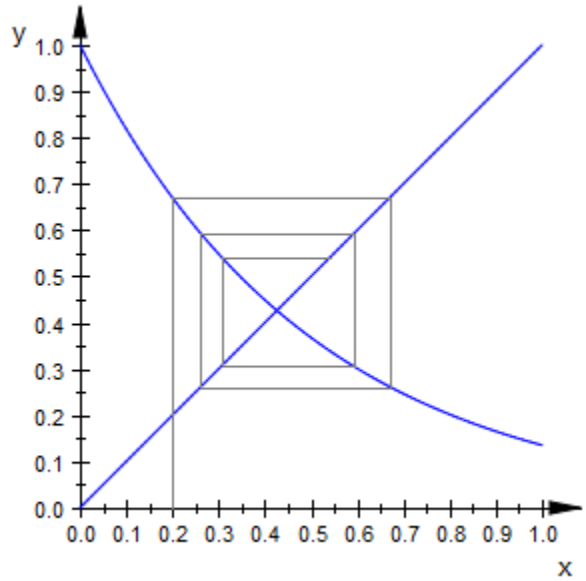
These attributes can be animated.

Examples

Example 1

We define and plot an iteration object:

```
f := plot::Function2d(exp(-2*x), x = 0..1):
g := plot::Function2d(x, x = 0..1):
it := plot::Iteration(exp(-2*x), 0.2, 5, x = 0..1):
plot(f, g, it)
```



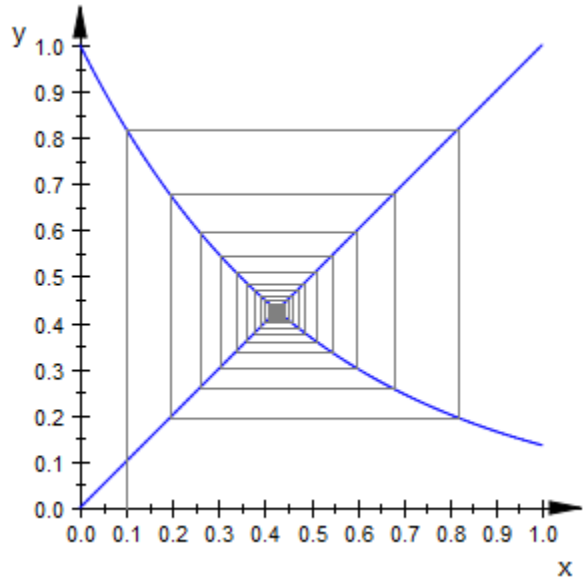
The starting point $x_0 = 0.2$ and the number of iteration steps 5 are stored inside the iteration object:

```
it::StartingPoint, it::Iterations
```

$$0.2,\ 5$$

We change these values:

```
it::StartingPoint := 0.1:
it::Iterations := 30:
plot(f, g, it)
```




```
delete f, g, it:
```

LineColorFunction, FillColorFunction

Functional line coloring

Value Summary

`FillColorFunction`, `LineColorFunction`	Optional	Color function (see below)

Graphics Primitives

Objects	Default Values
`plot::Conformal`, `plot::Curve2d`, `plot::Curve3d`, `plot::Cylindrical`, `plot::Density`, `plot::Dodecahedron`, `plot::Function2d`, `plot::Function3d`, `plot::Hexahedron`, `plot::Icosahedron`, `plot::Implicit2d`, `plot::Implicit3d`, `plot::Integral`, `plot::Listplot`, `plot::Matrixplot`, `plot::Octahedron`, `plot::Polar`, `plot::Polygon2d`, `plot::Polygon3d`, `plot::Prism`, `plot::Pyramid`, `plot::Rootlocus`, `plot::Sequence`, `plot::Spherical`, `plot::Streamlines2d`, `plot::Sum`, `plot::Surface`, `plot::SurfaceSet`, `plot::SurfaceSTL`, `plot::Sweep`, `plot::Tetrahedron`, `plot::Tube`, `plot::VectorField2d`, `plot::VectorField3d`, `plot::Waterman`, `plot::XRotate`, `plot::ZRotate`	

Description

These options accept functions that define the color of a plot at arbitrary points.

Using `FillColorType` and `LineColorType`, the user can control the color of many graphical objects. The setting providing the most detailed (and most complicated) control is `Functional`. In this case, a *color function* must be provided using one of `LineColorFunction`, `FillColorFunction`.

A color function can be a list of three or four expressions.

If three expressions are given, they specify RGB colors. If four expressions are given, they specify RGBA colors. See the introduction for more details on color specifications.

The expressions may contain the identifiers bound in the corresponding object. For example, in a `plot::Function2d(sin(x), x=0..PI)`, the color function may refer to `x`. More formally, the expressions may contain the identifiers found in the attributes `XName`, `YName`, `ZName`, `UName`, `VName`, and `ParameterName` of the plot object they are found in.

All of these expressions must, for values in the given ranges, evaluate to real numbers in the range `0..1`. Real values outside this range do not yield errors, they are simply clipped.

See also “Example 1” on page 24-1412.

Alternatively, a color function can be a procedure or function environment.

A procedure (or a function environment) used as a color function must return lists of three or four real numbers in the range `0..1`. Real values outside this range are clipped. (If this function ever returns a list of four numbers, it must always do so.) A list of three numbers is interpreted as an RGB color, while a list of four values is interpreted as an RGBA color. See the introduction for more details on color specifications.

The number and meaning of arguments a color function is called with depends on the object type. Informally, we have:

Type (abbreviated)	Parameters
`Conformal(f(z))`	`z`, `Re(f(z))`, `Im(f(z))`, `flag` (with `flag = 1` or `flag = 2`)
`Curve2d(x(u),y(u))`	`u`, `x(u)`, `y(u)`
`Curve3d(x(u),y(u),z(u))`	`u`, `x(u)`, `y(u)`, `z(u)`

Type (abbreviated)	Parameters
`Cylindrical(r(u,v),phi(u,v),z(u,v)`	`u`, `v`, `r(u,v)`, `phi(u,v)`, `z(u,v)`, `x(u)`, `y(u)`, `z(u)`
`Density(f(x,y))`	`x`, `y`, `f(x,y)`
`Dodecahedron`	See below.
`Function2d(f(x))`	`x`, `f(x)`
`Function3d(f(x,y))`	`x`, `y`, `f(x,y)`
`Hexahedron`	See below.
`Icosahedron`	See below.
`Implicit2d(f(x,y), Contours=[c])`	`x`, `y`, `D([1],f)(x,y)`, `D([2],f)(x,y)`, `c`
`Implicit3d(f(x,y,z), Contours=[c])`	`x`, `y`, `z`, `D([1],f)(x,y,z)`, `D([2],f)(x,y,z)`, `D([3],f)(x,y,z)`, `c`
`Matrixplot`	`x`, `y`, `z`
`Octahedron`	See below.
`Polar([r(t),phi(t)])`	`t`, `r(t)`, `phi(t)`, `x(t)`, `y(t)`
`Polygon2d([..,[xi,yi],..])`	`xi`, `yi`, `i`
`Polygon3d([..,[xi,yi,zi],..])`	`xi`, `yi`, `zi`, `i`
`Rootlocus(p(z, u))`	`u`, `Re(z)`, `Im(z)`
`Spherical(r(u,v),phi(u,v),thet(u,v`	`u`, `v`, `r(u,v)`, `phi(u,v)`, `thet(u,v)`, `x`, `y`, `z`
`Streamlines2d(v(x,y), w(x, y))`	`x`, `y`, `v(x,y)`, `w(x,y)`, `t`, `l`, `n`
`Surface(x(u,v),y(u,v),z(u,v))`	`u`, `v`, `x(u,v)`, `y(u,v)`, `z(u,v)`
`SurfaceSTL`	See below.
`SurfaceSet`	See below.
`Tetrahedron`	See below.
`Tube`	See below.
`VectorField2d(v(x,y),w(x,y))`	`x`, `y`, `v(x,y)`, `w(x,y)`
`XRotate(f(x))`	`x`, `phi`, `x`, `y(x,phi)`, `z(x,phi)`
`ZRotate(f(t))`	`t`, `phi`, `x(t,phi)`, `y(t,phi)`, `f(t)(=z(t,phi))`

Additionally, for animated objects, the current value of the animation parameter is provided.

`Dodecahedron`, `Hexahedron`, `Icosahedron`, `SurfaceSTL`, `SurfaceSet`, and `Tetrahedron` are built from triangles; the color functions are called once for each vertex of these triangles and are passed the number of the triangle (an integer count starting at 1), the coordinates of the vertex and the animation parameter, if that is used.

For `plot::Tube`, the color functions are given the coordinates of the currently visited point on the central curve, followed by the coordinates of the point on the surface, followed by the animation parameter, if any. (That makes seven arguments altogether.)

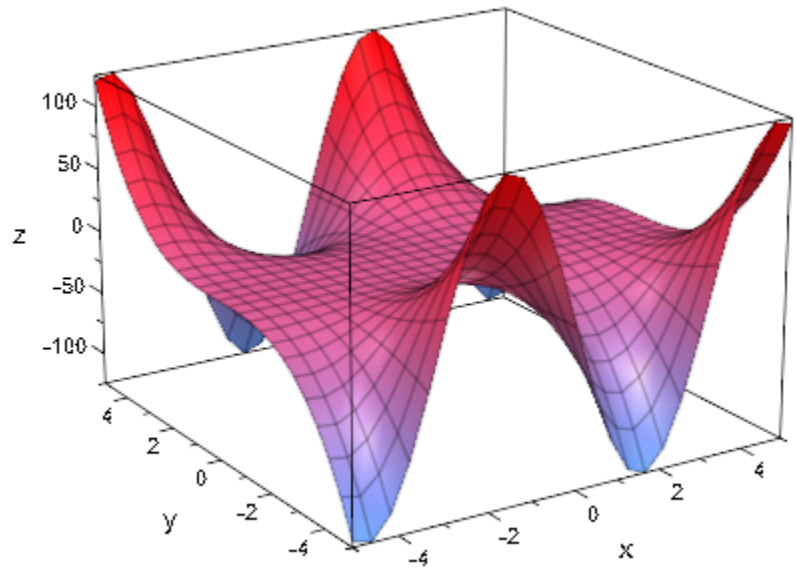
The examples below show different usage environments of color functions for some of the object types listed above.

Examples

Example 1

By default, most 3D-objects in MuPAD get “height coloring”:

```
plot(plot::Function3d(sin(x)*y^3))
```

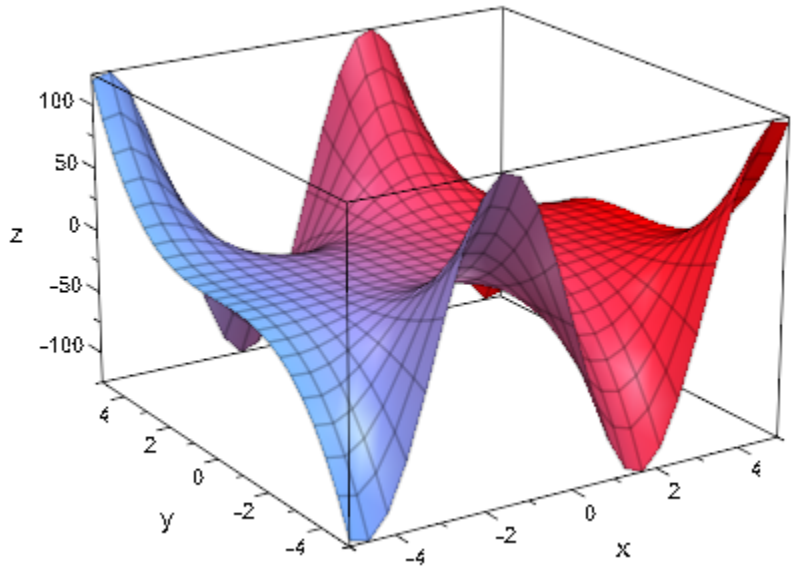


To change the direction of this color, you can use `FillColorFunction`:

```
xmin := -5:
xmax := 5:
color := zip(RGB::Red, RGB::CornflowerBlue,
             (a, b) -> (x-xmin)/(xmax-xmin)*a
                       +(xmax-x)/(xmax-xmin)*b)
```

$$[0.0607807\,x + 0.6960965,\ 0.2921535 - 0.0584307\,x,\ 0.4646975 - 0.0929395\,x]$$

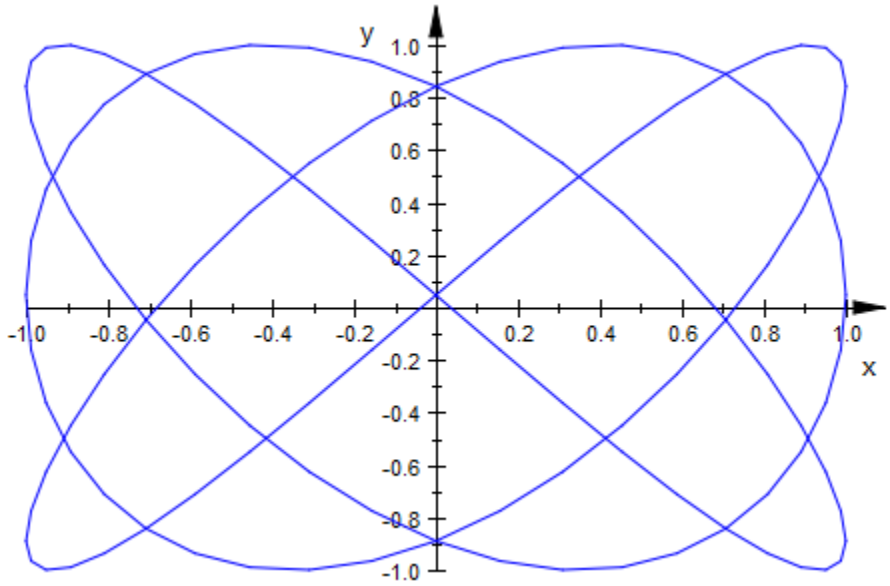
```
plot(plot::Function3d(sin(x)*y^3,
         FillColorFunction = color))
```



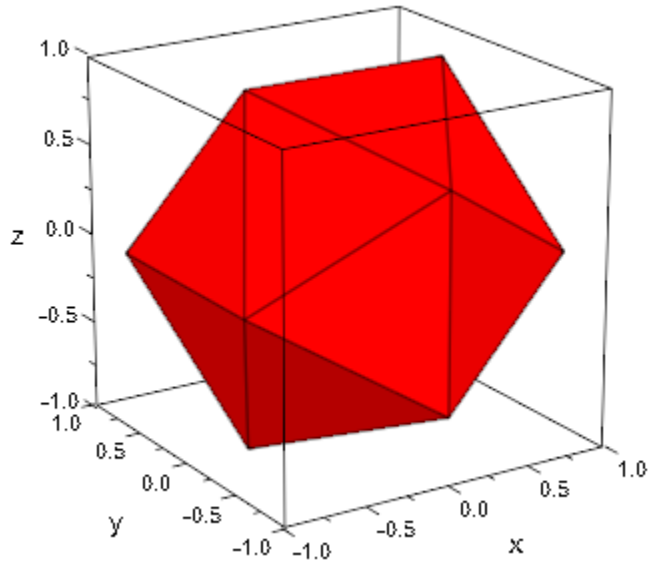
Example 2

Animated color functions can be combined with static objects:

```
plot(
  plot::Curve2d([sin(3*x), sin(4*x + 1)], x = 0..2*PI,
                LineColorFunction = ((u, x, y, a) ->
                                     [(u-a)/5, (u-a)/5, 1]),
                a = -5..6)
    )
```



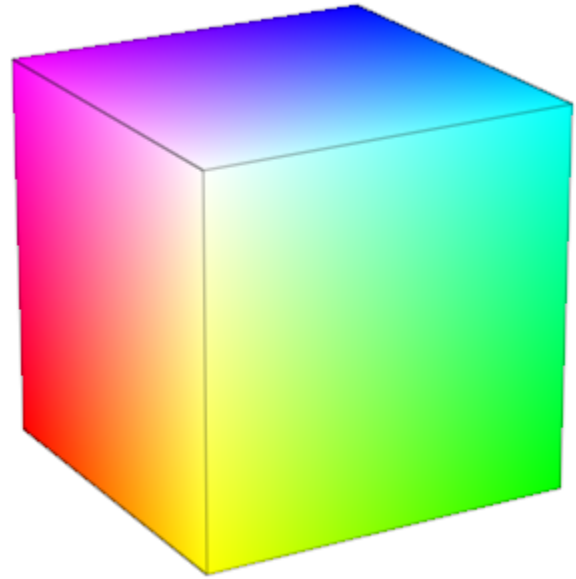

```
cf := (i, x, y, z, a) -> [RGB::Red,
                          RGB::Green,
                          RGB::Blue][(floor(a*i)
                                      mod 3) + 1]:
plot(plot::Icosahedron(FillColorFunction = cf,
                       a = 0..9))
```



Example 3

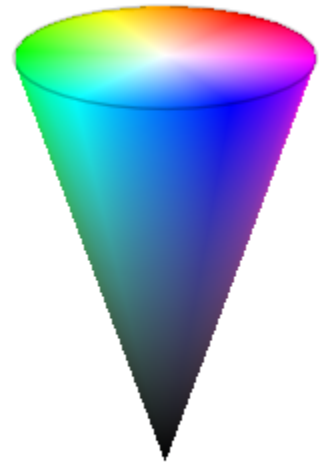
A color function should generate values in RGB color space. Since a `plot::Box` does not allow a `FillColorFunction`, we use six (trivial) `Surface` objects to show the outside of this color space:

```
rgb := (u, v, x, y, z) -> [x, y, z]:
plot(plot::Surface(formula, u = 0..1, v = 0..1,
                   FillColorFunction = rgb)
     $ formula in [[0, u, v], [1, u, v],
                   [u, 0, v], [u, 1, v],
                   [u, v, 0], [u, v, 1]],
     plot::Box(0..1, 0..1, 0..1, Filled = FALSE,
               LineColor = RGB::Black.[0.25]),
     Scaling = Constrained, Axes = None,
     ULinesVisible = FALSE, VLinesVisible = FALSE,
     Lighting = None, CameraDirection = [4, 7, 3])
```



RGB colors are a very technical way of defining a color. The HSV color space is more popular with designers, since there the “hue” (i.e., the perceived color type) is not a combination of three numbers but rather one of the numbers making up a color:

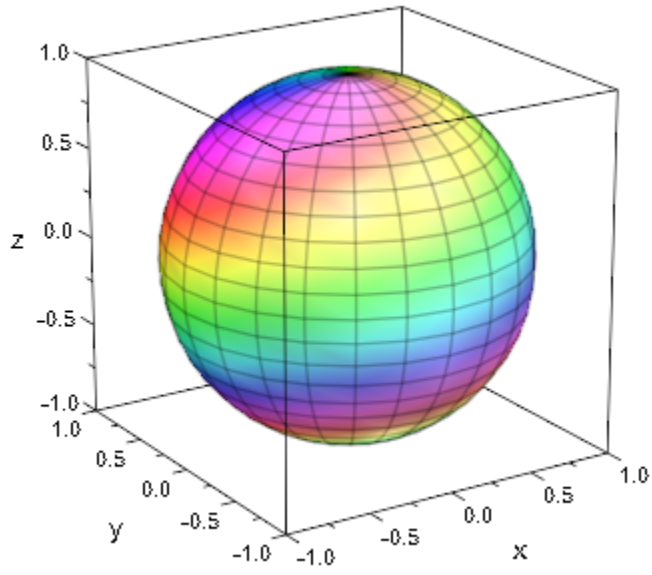
```
hsv := (u, v, r, phi, z) -> RGB::fromHSV([180/PI*phi, r, z]):
plot(plot::Cylindrical([z, phi, z], z = 0..1, phi = 0..2*PI,
                       FillColorFunction = hsv),
     plot::Cylindrical([r, phi, 1], r = 0..1, phi = 0..2*PI,
                       FillColorFunction = hsv),
     plot::Circle3d(1, [0, 0, 1], [0, 0, 1],
                    Color = RGB::Black.[0.25]),
     ZXRatio = 1.5, Scaling = Unconstrained,
     Axes = None, Lighting = None,
     ULinesVisible = FALSE, VLinesVisible = FALSE,
     CameraDirection = [-17, -12, 3])
```



Example 4

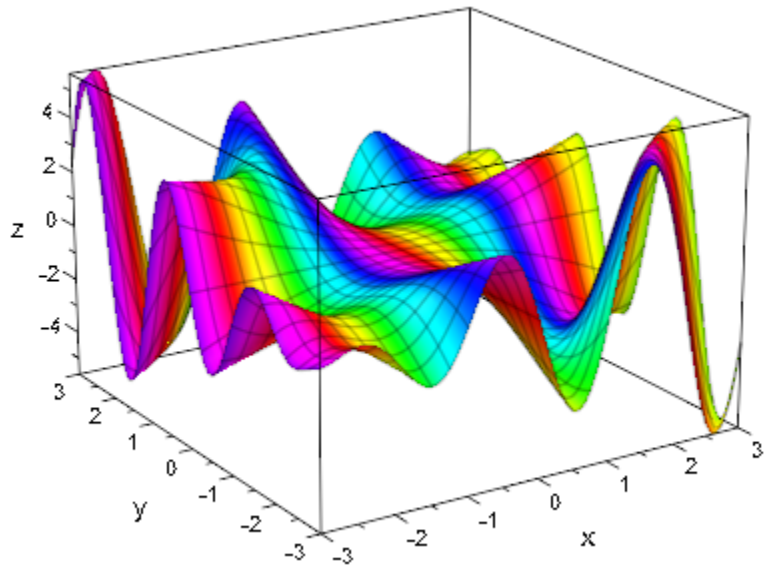
HSV color space is especially suitable for quick coloring of cylindrical,polar, or spherical plots, due to its circular nature:

```
hsv := (u, v, r, phi,thet) ->
    RGB::fromHSV([180/PI*(phi+(thet+2)^3/PI^2),
                  3/4+sin(u)/4, 1]):
plot(plot::Spherical([1, u, v], u = 0..2*PI, v = 0..PI,
                     FillColorFunction = hsv))
```



There are other examples, where the cyclic nature comes in handy, too:

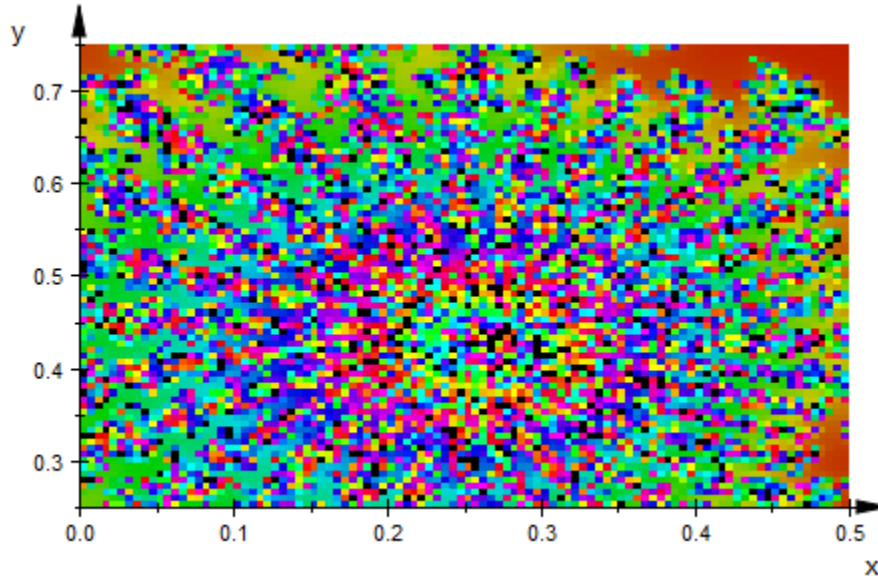
```
hsv := (x, y, z) -> RGB::fromHSV([150*z, 1, 1]):
plot(plot::Function3d(sin(x*y)*(x-y), x = -3..3, y = -3..3,
                      Submesh = [2, 2], FillColorFunction = hsv))
```



The following example takes a long time to compute. Reducing the values set for `Mesh` results in a shorter computation, while higher values lead to an image with finer details:

```
c := 0.377+0.2*I:
julia := proc(x, y)
           local i, z;
         begin
           i := 0;
           z := float(x + I*y);
           while i < 1000 and abs(z) < 4 do
             z := z^2 + c;
             i := i + 1;
           end_while;
           i;
         end_proc:
Jcol := (x, y, i) -> if i >= 1000 then
                       RGB::Black
                     else
                       RGB::fromHSV([i, 1, 3/4+i/2000])
                     end:
plot(plot::Density(julia, x = 0..0.5, y=0.25..0.75,
                   FillColorFunction = Jcol,
```

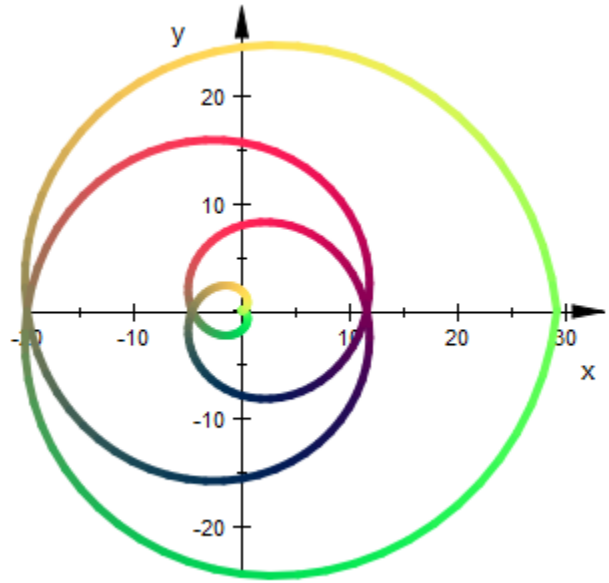
```
                Mesh = [100,75]))
```



Example 5

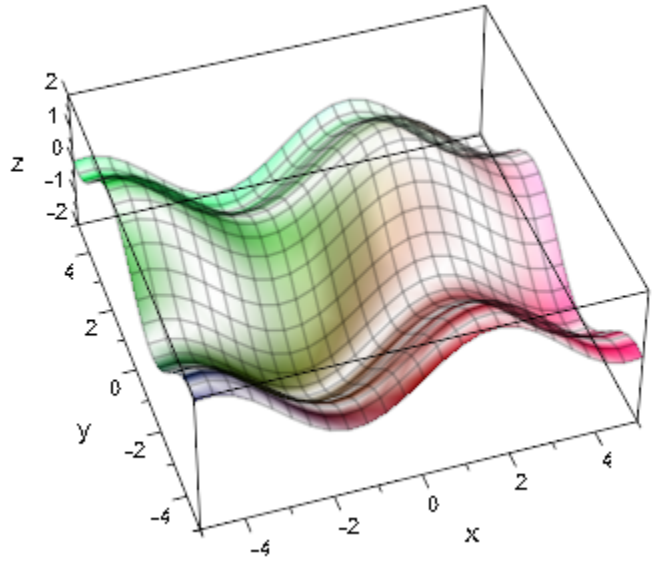
Another way of getting a smooth color transition is to use a periodic function in between, for example trigonometric ones (note the `(1+sin(a))/2`: we need values between 0 and 1):

```
plot(
  plot::Polar([r*surd(r, 3), r], r = -4*PI..4*PI,
              AdaptiveMesh = 2,
              LineColorFunction = [(sin(r) + 1)/2,
                                   (cos(r/2) + 1)/2,
                                   1/3],
              LineWidth = 1*unit::mm)
    )
```



This also applies for cyclic colors in terms of time:

```
plot(plot::Function3d(sin(x)+sin(y), x = -5..5, y = -5..5,
          FillColorFunction = [(x+5)/10, (y+5)/10,
                               abs(x+y+5*cos(a))/15,
                               (1+cos(x+y^2-a))/2],
          a = 0..2*PI),
     CameraDirection = [-1, -3, 3],
     Scaling = Constrained)
```



Algorithms

Animation is handled by the general framework, not the individual objects. Therefore, the framework also supplies the animation parameter to the color functions.

See Also

MuPAD Functions

`FillColor` | `FillColorType` | `LineColor` | `LineColorType`

Matrix2d, Matrix3d

Transformation matrices

Value Summary

`Matrix2d`, `Matrix3d`	Optional	List of four real-valued expressions

Description

`Matrix2d`, `Matrix3d` represent the transformation matrices of transformation objects.

The general transformation objects `plot::Transform2d` and `plot::Transform3d` allow to apply the affine-linear transformation $x \to A\,x + b$ to 2D and 3D objects, respectively. Depending on the dimension, the transformation matrix A can be accessed and changed via the attributes `Matrix2d`, `Matrix3d`, of the transformation object. The shift vector b can be accessed and changed via the attribute `Shift`.

When setting the matrix attribute, matrices, arrays, lists of lists, and plain lists are accepted. Internally, however, the matrix data are always stored as a plain list

$[A_{1,1}, A_{1,2}, A_{2,1}, A_{2,2}]$

in 2D or

$[A_{1,1}, A_{1,2}, \ldots, A_{3,2}, A_{3,3}]$

in 3D, respectively, representing the matrix row by row. When reading the matrix by a slot access, this plain list is returned.

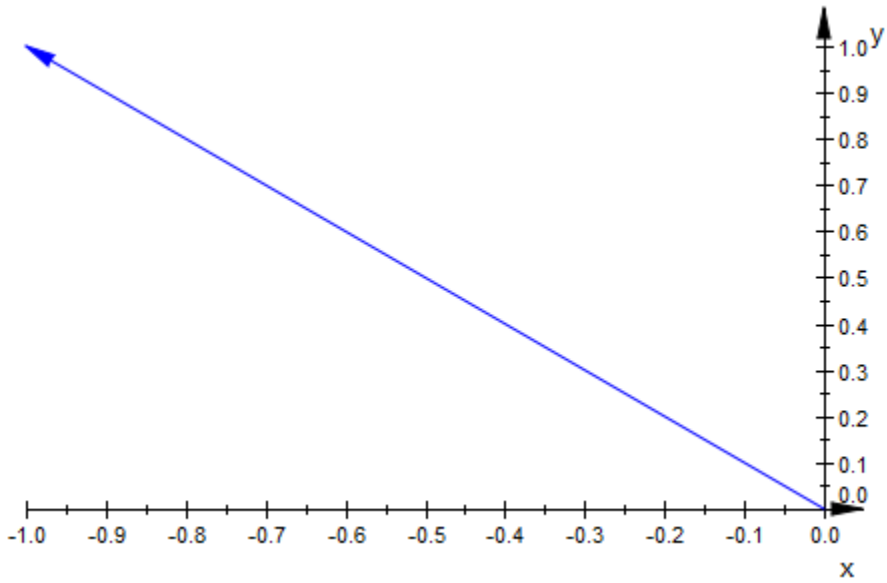
The entries of `Matrix2d`, `Matrix3d` can be animated.

Examples

Example 1

We apply a linear transformation to an arrow:

```
A := matrix([[1, -1], [1, 1]]):
g := plot::Transform2d(A, plot::Arrow2d([0, 0], [0, 1])):
plot(g)
```



The `Matrix2d` corresponding to the transformation is stored as a plain list in the corresponding slot of `g`:

```
g::Matrix2d
```

$$[1.0, -1.0, 1.0, 1.0]$$

```
delete f, g:
```

See Also

MuPAD Functions
`Scale` | `Shift`

MeshList, MeshListType, MeshListNormals

Triangulation data

Value Summary

`MeshList`	Mandatory	List of arithmetical expressions
`MeshListType`	Optional	`ColorQuads`, `Quads`, `QuadStrip`, `Triangles`, `TriangleFan`, or `TriangleStrip`
`MeshListNormals`	Optional	`BeforeFacets`, `BeforePoints`, `BehindFacets`, `BehindPoints`, or `None`

Graphics Primitives

Objects	Default Values

`plot::SurfaceSet`	`MeshListType`: `Triangles`
	
	`MeshListNormals`: `None`

Description

`MeshList` is a list of data defining the triangulation of a 3D surface of type `plot::SurfaceSet`.

`MeshListType` specifies how the data in the list `MeshList` are to be interpreted.

`MeshListNormals` specifies which of the data in the list `MeshList` are to be interpreted as normals.

`MeshList` contains coordinates of points (and optional normals) of either triangles or quads which define a mesh of a 3D surface. The points must be given homogenous: If a normal is given, it must be given for all points or facets, respectively. The attribute `MeshListType` specifies how these points are to be interpreted for plotting the surface. The attribute `MeshListNormals` specifies whether the list contains normal vectors and at which positions they located.

About normals and facet orientation: The facets (triangles or quads) define the surface of a 3D object. As such, each facet is part of the boundary between the interior and the exterior of the object. The orientation of the facets (which way is "out" and which way is "in") is specified redundantly in two ways which should be consistent: First, the direction of the normal is outward. Second, which is most commonly used now-a-day, the facet vertices are listed in counter-clockwise order when looking at the object from the outside (right-hand rule). Normals must be given as unit vectors.

`MeshList` must not contain color values. Use the color functions `LineColorFunction` and `FillColorFunction` instead.

`MeshListType` specifies how the points in `MeshList` are to be interpreted. Supported mesh list types are:

Value	Info	Description
`Triangles`	Set of separate triangles	Each tuple of three points define one new triangle.
`TriangleFan`	Triangle fan	The first triangle is defined by the first three points. The next triangles are defined by the first point, the previous point and the current point.
`TriangleStrip`	Triangle strip	The first triangle is defined by the first three points. The next triangles are defined by the two previous points and the current point.
`Quads`	Set of separate quads	Each tuple of four points define one new quad.
`QuadStrip`	Strip of quads	The first quad is defined by the first four points. The next quads are defined by the two previous points and the next two points.

`MeshListNormals` specifies whether `MeshList` contains normal vectors and at which positions they are located. Valid options are:

Value	Description
`None`	No normals are specified.
`BeforePoints`	A normal is given before each point.
`BehindPoints`	A normal is given behind each point.

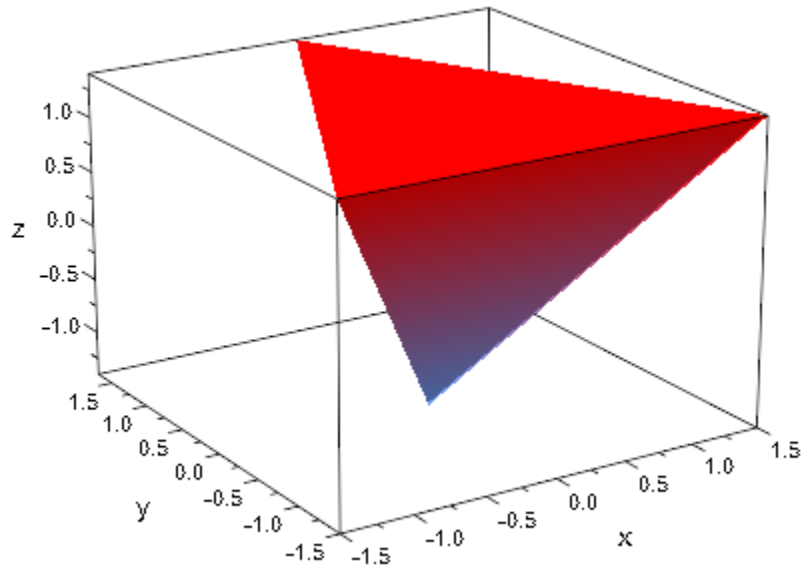
Value	Description
`BeforeFacets`	A normal is given before each triangle or quad, respectively. This option is only valid for `MeshListType = Triangles` and `MeshListType = Quads`.
`BehindFacets`	A normal is given behind each triangle or quad, respectively. This option is only valid for `MeshListType = Triangles` and `MeshListType = Quads`.

Examples

Example 1

We create a triangle set with normals in front of each triangle and plot this object, a tetrahedron, afterwards:

```
meshList:= [
   0.0 ,  0.0 , -1.0 ,
  -1.5 , -1.5 ,  1.4 ,  0.0,  1.7, 1.4, 1.5, -1.5,  1.4,
   0.0 ,  0.88,  0.47,
  -1.5 , -1.5 ,  1.4 ,  1.5, -1.5, 1.4, 0.0,  0.0, -1.4,
  -0.88, -0.41,  0.25,
   1.5 , -1.5 ,  1.4 ,  0.0,  1.7, 1.4, 0.0,  0.0, -1.4,
   0.88, -0.41,  0.25,
   0.0 ,  1.7 ,  1.4 , -1.5, -1.5, 1.4, 0.0,  0.0, -1.4
]:
plot(plot::SurfaceSet(meshList,
                      MeshListType = Triangles,
                      MeshListNormals = BeforeFacets)):
```




```
delete meshList:
```

Example 2

See `plot::SurfaceSet` for further examples.

See Also

MuPAD Functions

`OutputFile` | `UseNormals`

Name

Name of an object

Value Summary

Optional	Text string

Graphics Primitives

Objects	Name Default Values
`plot::AmbientLight`, `plot::Arc2d`, `plot::Arc3d`, `plot::Arrow2d`, `plot::Arrow3d`, `plot::Bars2d`, `plot::Bars3d`, `plot::Box`, `plot::Boxplot`, `plot::Camera`, `plot::Canvas`, `plot::Circle2d`, `plot::Circle3d`, `plot::ClippingBox`, `plot::Cone`, `plot::Conformal`, `plot::CoordinateSystem2d`, `plot::CoordinateSystem3d`, `plot::Curve2d`, `plot::Curve3d`, `plot::Cylinder`, `plot::Cylindrical`, `plot::Density`, `plot::DistantLight`, `plot::Dodecahedron`, `plot::Ellipse2d`, `plot::Ellipse3d`, `plot::Ellipsoid`, `plot::Function2d`, `plot::Function3d`, `plot::Group2d`, `plot::Group3d`, `plot::Hatch`, `plot::Hexahedron`, `plot::Histogram2d`, `plot::Icosahedron`, `plot::Implicit2d`, `plot::Implicit3d`, `plot::Inequality`, `plot::Integral`, `plot::Iteration`, `plot::Line2d`, `plot::Line3d`, `plot::Listplot`,	

Objects	Name Default Values
plot::Lsys, plot::Matrixplot, plot::MuPADCube, plot::Octahedron, plot::Ode2d, plot::Ode3d, plot::Parallelogram2d, plot::Parallelogram3d, plot::Piechart2d, plot::Piechart3d, plot::Plane, plot::Point2d, plot::Point3d, plot::PointLight, plot::PointList2d, plot::PointList3d, plot::Polar, plot::Polygon2d, plot::Polygon3d, plot::Prism, plot::Pyramid, plot::QQplot, plot::Raster, plot::Rectangle, plot::Reflect2d, plot::Reflect3d, plot::Rootlocus, plot::Rotate2d, plot::Rotate3d, plot::Scale2d, plot::Scale3d, plot::Scatterplot, plot::Scene2d, plot::Scene3d, plot::Sequence, plot::SparseMatrixplot, plot::Sphere, plot::Spherical, plot::SpotLight, plot::Streamlines2d, plot::Sum, plot::Surface, plot::SurfaceSet, plot::SurfaceSTL, plot::Sweep, plot::Tetrahedron, plot::Text2d, plot::Text3d, plot::Transform2d, plot::Transform3d, plot::Translate2d, plot::Translate3d, plot::Tube, plot::Turtle, plot::VectorField2d, plot::VectorField3d, plot::Waterman, plot::XRotate, plot::ZRotate	

Description

The attribute `Name` sets the name of a graphical object. The object is displayed by this name in the legend and the interactive object browser of the graphics tool.

Giving a name to a graphical object has no significance whatsoever for the graphical appearance of the object. The main purpose of the name is to make it easier to identify the object in the interactive "object browser" of the MuPAD graphics tool (see section Viewer, Browser, and Inspector: Interactive Manipulation of this document).

If no name is specified, the type of the object is diplayed in the object browser.

If the legend is "switched on" by setting `LegendVisible` to `TRUE`, the name slot of an object is used (if if exists), unless the object has a specific `LegendText`.

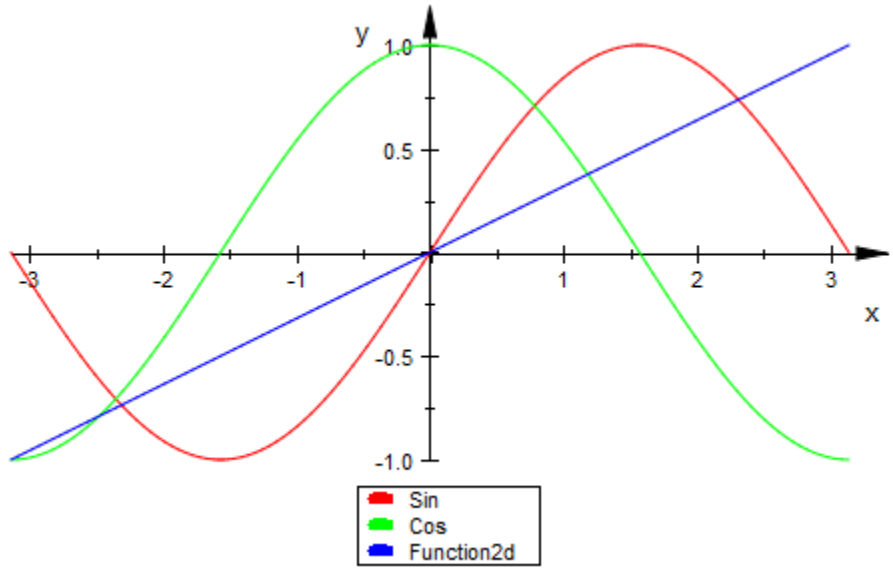
`Name` has a special technical semantics for objects of type `plot::Hatch`. The bounding functions or curves of the hatch are referenced via their name slot. If the bounding function or curve has no name slot, it is set implicitly by `plot::Hatch` to the output of `expr2text` of the function/curve. Cf. "Example 2" on page 24-1433Example 2.

Examples

Example 1

The first two of the following function objects are given the names `Sin` and `Cos`, respectively. Generate the graphics and doubleclick on the plot. The two function objects are visible under their names in the object browser. The third function is just labeled as a 'Function2d' object. Also the legend uses this labeling:

```
plot(plot::Function2d(sin(x), x = -PI..PI, Name = "Sin",
                      Color = RGB::Red),
     plot::Function2d(cos(x), x = -PI..PI, Name = "Cos",
                      Color = RGB::Green),
     plot::Function2d(x/PI , x = -PI..PI, Color = RGB::Blue),
     LegendVisible = TRUE)
```



Example 2

By default, an object does not have a `Name` entry:

```
f := plot::Function2d(cos(x), x=0..PI):
f::Name
```

FAIL

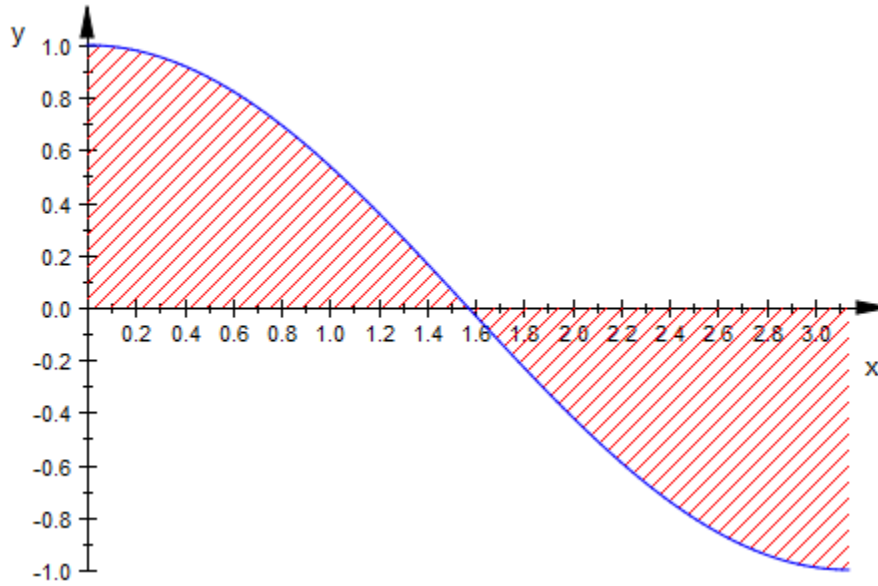
When creating a `plot::Hatch` object, the `Name` slot is set implicitly for the border function(s):

```
h := plot::Hatch(f):
f::Name
```

"Function2d(XMin = 0, XMax = PI, XName = x, Function = cos(x), XAxisTitle = "x")"

We plot the hatch with its bounding function:

```
plot(h, f)
```



Doubleclick on the graphics and observe the way the objects are labeled in the object inspector.

```
delete f, h:
```

See Also

MuPAD Functions

`Function1 | Function2 | LegendText | Title`

Nodes

Number of subintervals or list of x-values for subintervals

Value Summary

Optional	List of arithmetical expressions

Graphics Primitives

Objects	Nodes Default Values
`plot::Integral`	[10]

Description

`Nodes` is a positive number of subintervals for numeric approximation of integrals. The given interval for approximation is divided into the given number of subintervals, all of the same width.

Otherwise, `Nodes` can be a list of x-values for dividing the given interval. The interval is divided into subintervals at the given x-values.

When a number is given for `Nodes`, the number can be given as a list with this one number, too.

When a list with x-values is given, the left and right border of the whole (approximation) interval can be omitted. In this case, the number of subintervals is the number of given x-values plus one.

Nodes outside the approximation interval are ignored. Duplicate values are ignored.

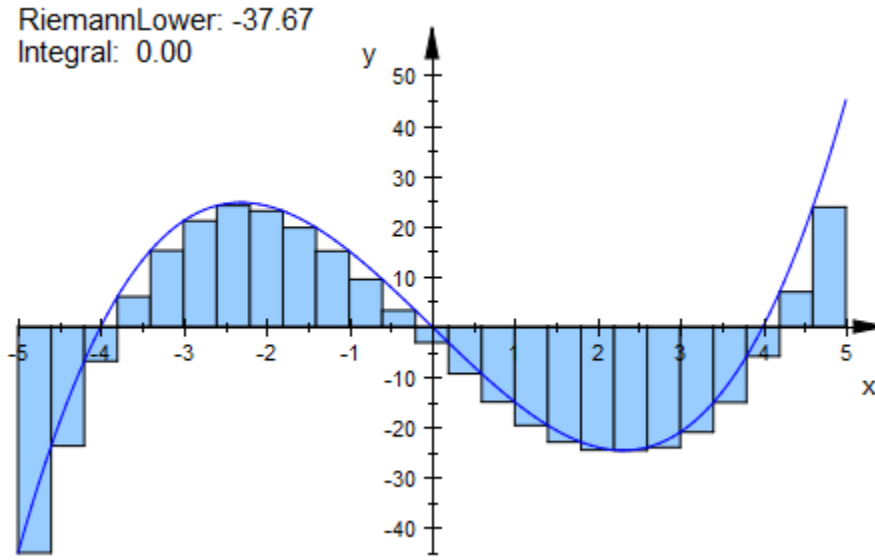
The nodes need not be ordered.

Examples

Example 1

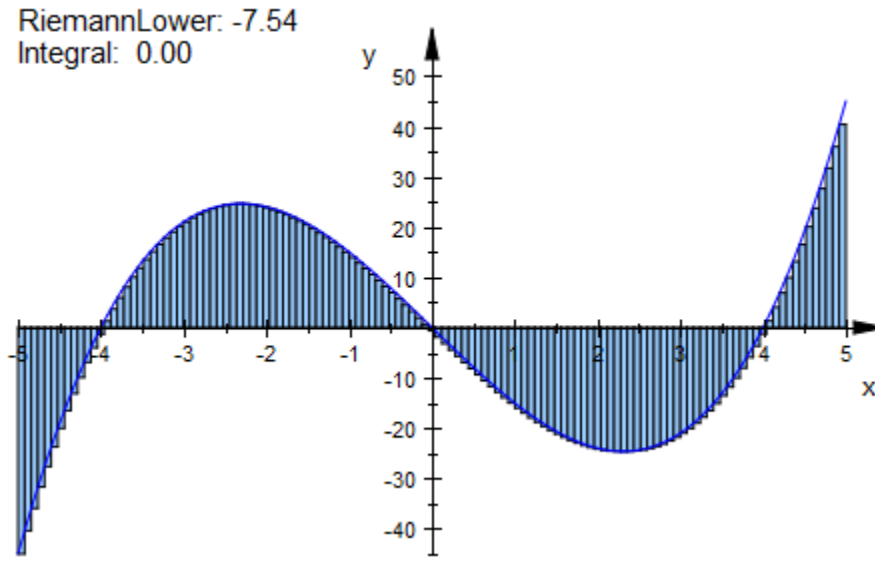
`Nodes` determines the number of rectangles for Riemann sums:

```
f := plot::Function2d(x*(x+4)*(x-4)):
plot(plot::Integral(f, Nodes = 25, IntMethod = RiemannLower), f)
```



Increasing of `Nodes` decreases the error or the approximation:

```
plot(plot::Integral(f, Nodes = 125, IntMethod = RiemannLower), f)
```

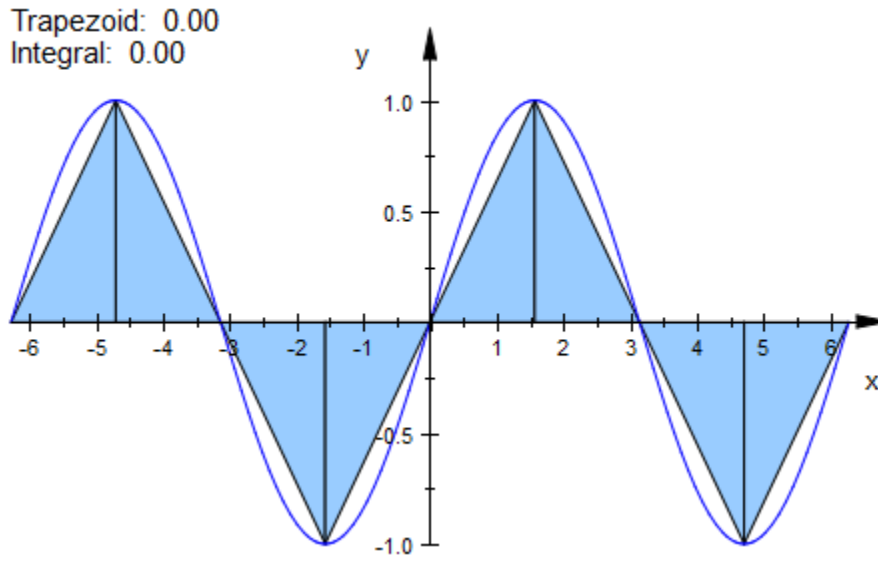



```
delete f:
```

Example 2

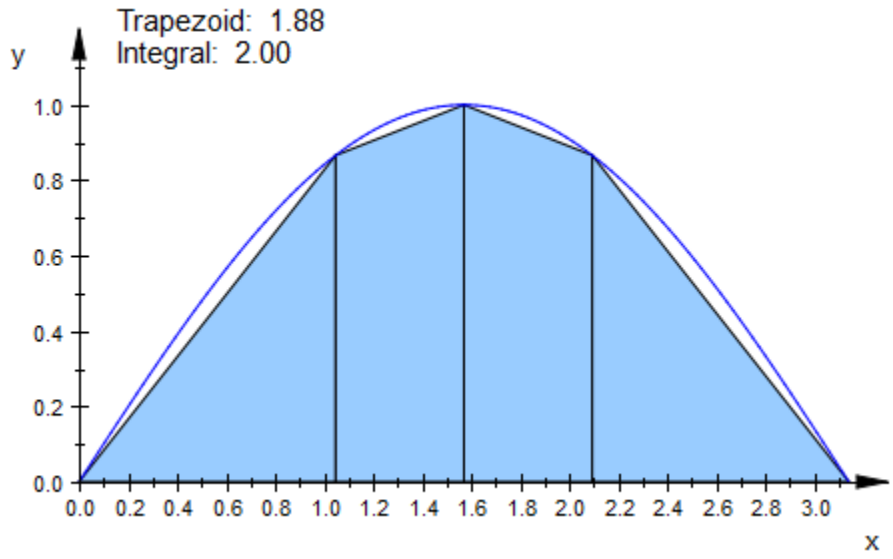
We request a specific division into subintervals:

```
f := plot::Function2d(sin(x), x = -2*PI..2*PI):
plot(
  plot::Integral(f, Nodes = [i*PI/2 $ i = -4..4],
                    IntMethod = Trapezoid),
    f)
```



The subintervals do not need to be of equal width:

```
f := plot::Function2d(sin(x), x = 0..PI):
plot(
  plot::Integral(f, [PI/3, PI/2, 2*PI/3],
                 IntMethod = Trapezoid),
   f)
```




```
delete f:
```

See Also

MuPAD Functions
`IntMethod`

Normal, NormalX, NormalY, NormalZ

Normal vector of circles and discs, etc. in 3D

Value Summary

`Normal`	Library wrapper for "[`NormalX`, `NormalY`]" (2D), "[`NormalX`, `NormalY`, `NormalZ`]" (3D)	List of 2 or 3 expressions, depending on the dimension
`NormalX`, `NormalY`, `NormalZ`	Mandatory	MuPAD expression

Graphics Primitives

Objects	Default Values
`plot::Prism`, `plot::Pyramid`	`Normal`: [0, 0, 0] `NormalX`, `NormalY`, `NormalZ`: 0
`plot::Arc3d`, `plot::Circle3d`, `plot::Ellipse3d`, `plot::Plane`, `plot::Reflect3d`	`Normal`: [0, 0, 1] `NormalX`, `NormalY`: 0 `NormalZ`: 1

Description

`Normal` determines the normal vector of the plane of the 3D circle, prism or pyramid. It is given by a list or vector of 3 components.

`NormalX` etc. refer to the x, y, z components of this vector.

The values of these attributes can be animated.

With `Filled` = `TRUE`, a circle becomes a disc.

Examples

Example 1

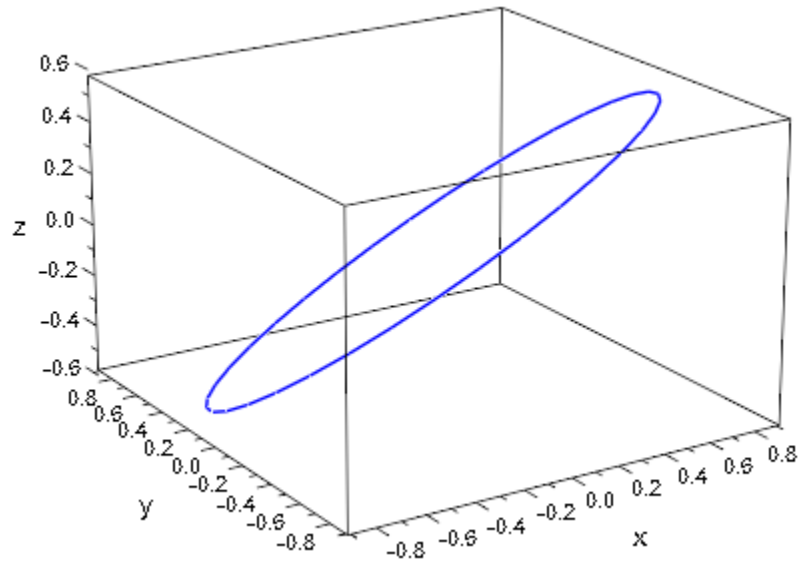
We create a circle around the origin lying in the *x*-*y* plane:

```
c := plot::Circle3d(1, [0, 0, 0], [0, 0, 1])
```

```
plot::Circle3d(1, [0, 0, 0], [0, 0, 1])
```

The second argument in `plot::Circle3d` is the center, the third argument is the normal. Internally, these vectors are stored as the attributes `Center` and `Normal` and can be changed by assigning a new value:

```
c::Normal := [-0.5, 0.5, 1]:
plot(c):
```

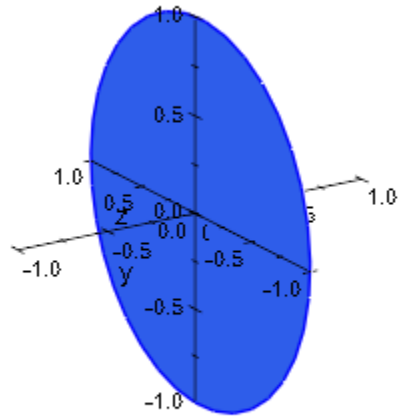



```
delete c:
```

Example 2

`Normal` can be animated:

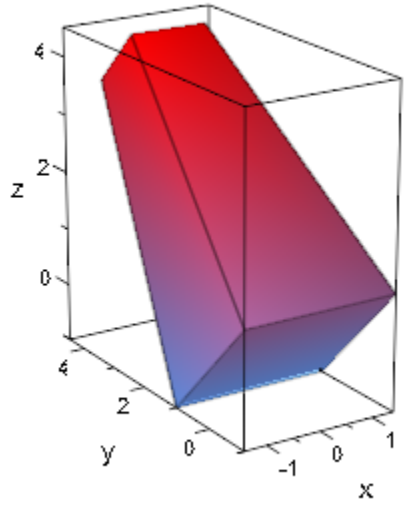
```
plot(plot::Circle3d(1, [0, 0, 0], [cos(a), sin(a), 0],
                    a = 0 .. 2*PI, Filled = TRUE),
     Axes = Origin):
```

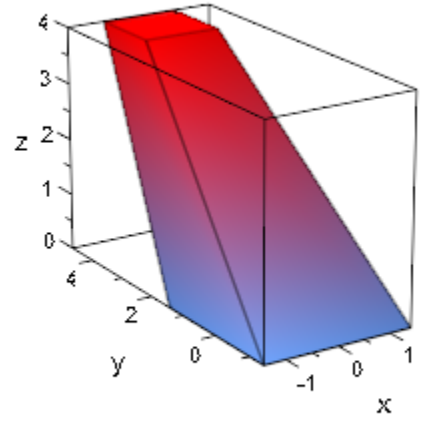


Example 3

`Normal` can be used to create crooked prisms, pyramids and frustums of pyramids. If this attribute is set to [0, 0, 0], the axis between `Base` and `Top` ist used as normal vector:

```
plot(plot::Scene3d(plot::Pyramid(2,[0,0,0],1,[0,4,4], Normal=[0,0,0])),
     plot::Scene3d(plot::Pyramid(2,[0,0,0],1,[0,4,4], Normal=[0,0,1]))):
```


z
4
2
0
y
4
2
0
x
-1
0
1


z
4
3
2
1
0
y
4
2
0
x
-1
0
1

ParameterName, ParameterBegin, ParameterEnd, ParameterRange

Name of the animation parameter

Value Summary

`ParameterBegin`, `ParameterEnd`, `ParameterName`	Optional	MuPAD expression
`ParameterRange`	[`ParameterBegin` .. `ParameterEnd`]	Range of arithmetical expressions

Description

Typically, animations are triggered by passing an equation of the form `a = \`a_{min}\`..\`a_{max}\`` in the definition of an object.

This is equivalent to passing the attributes `ParameterName = a`, `ParameterBegin =` a_{min}, and `ParameterEnd =` a_{max} in the definition of the object.

The attribute `ParamterRange = \`a_{min}\`..\`a_{max}\`` is equivalent to setting both `ParameterBegin =` a_{min} and `ParameterEnd =` a_{max}.

Animations are defined object by object, not frame by frame.

In most cases, the user will define animations by passing an equation of the form `a = \`a_{min}\`..\`a_{max}\`` in the definition of an object. Any equation of this form that is not essential for the definition of a static version of the object is interpreted as an animation parameter and an animation range.

Passing such an equation is equivalent to setting the three attributes

`ParameterName = a`, `ParameterBegin =` a_{min}, `ParameterEnd =` a_{max}.

The attribute `ParamterRange = `a_{min}`..`a_{max}`` serves as a short cut for setting both `ParameterBegin = `a_{min} and `ParameterEnd = `a_{max}.

The values a_{min} and a_{max} are the parameter values at the beginning and the end of the real time range in which an object is animated. This time range is set by the attributes `TimeBegin` and `TimeEnd`, respectively.

The parameter range `a_{min}`..`a_{max}` is mapped linearly to this time interval.

The name of the animation parameter may be an identifier or an indexed identifier. This parameter is a `global variable' that may be present in other quantities or attributes defining the object.

The definition of an object may involve procedures rather than symbolic expressions. E.g., a 2D function plot may be defined by `plot::Function2d(f, x = x_0..x_1)`, where *f* is a procedure accepting one numerical argument *x* from the plot range between x_0 und *x1*.

In an animated version `plot::Function2d(f, x = x_0..x_1, a = `a_{min}`..`a_{max}`)`, the function *f* will be called with two arguments *x* and *a*. Thus, *f* may be defined as a function accepting two parameters *x*, *a* or as a function with one parameter *x*, using the animation parameter *a* as a global variable.

Each animated object has its own animation parameter and range `a_{min}`..`a_{max}`. It is not necessary that several animated objects in a scene use the same parameter name. It is not used to synchronize the animations.

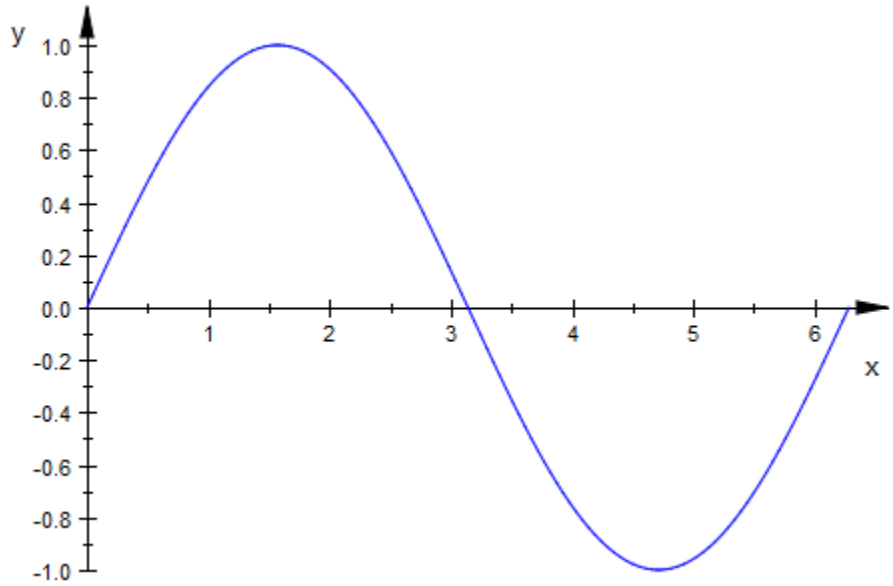
The synchronization is determined uniquely by the linear correspondence between the animation range `a_{min}`..`a_{max}` and the real time span of the animation set by the attributes `TimeBegin` and `TimeEnd` of the object.

Examples

Example 1

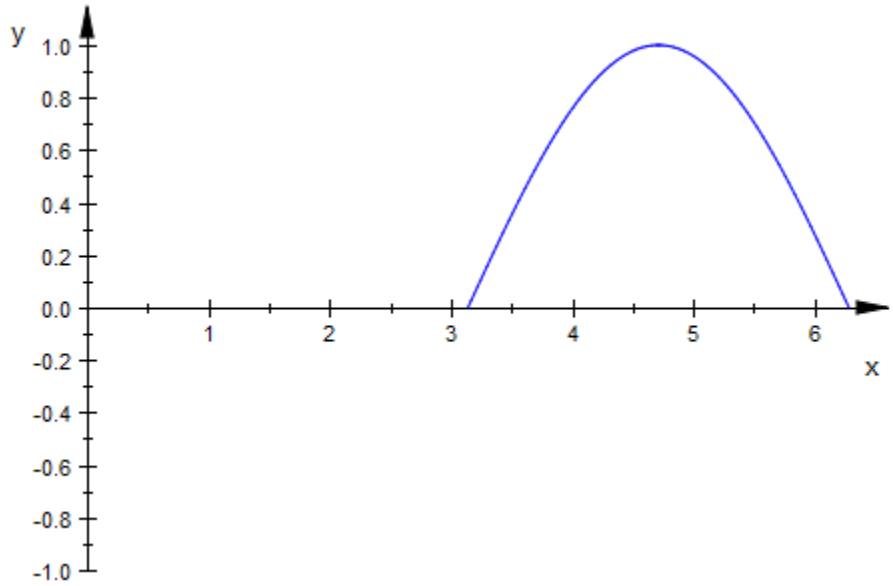
The definition of a static 2D function plot involves the specification of one range (for the *x* variable):

```
plot(plot::Function2d(sin(x), x = 0 .. 2*PI))
```



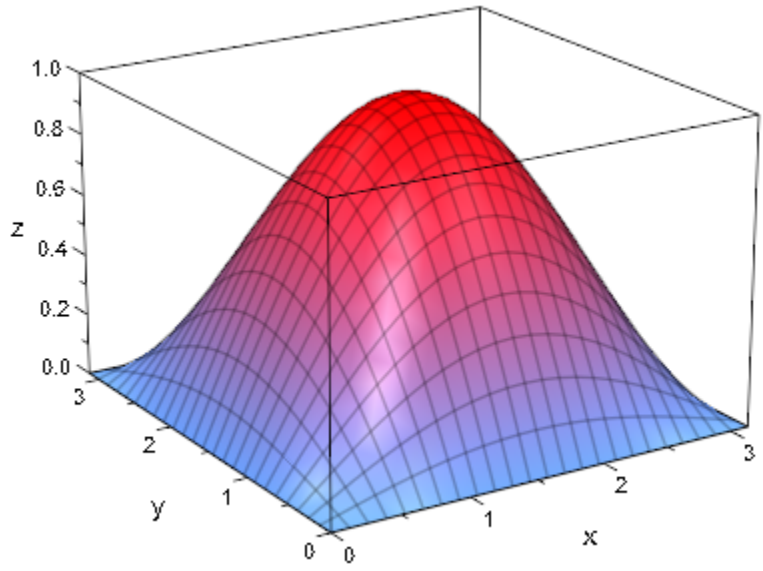
When a "surplus equation" `a = `a_{min}`..`a_{max}`` is passed, this is interpreted as a call to animate the function. The animation parameter may turn up in the expression defining the function:

```
plot(plot::Function2d(sin(x + a), x = a .. 2*PI, a = 0..PI))
```



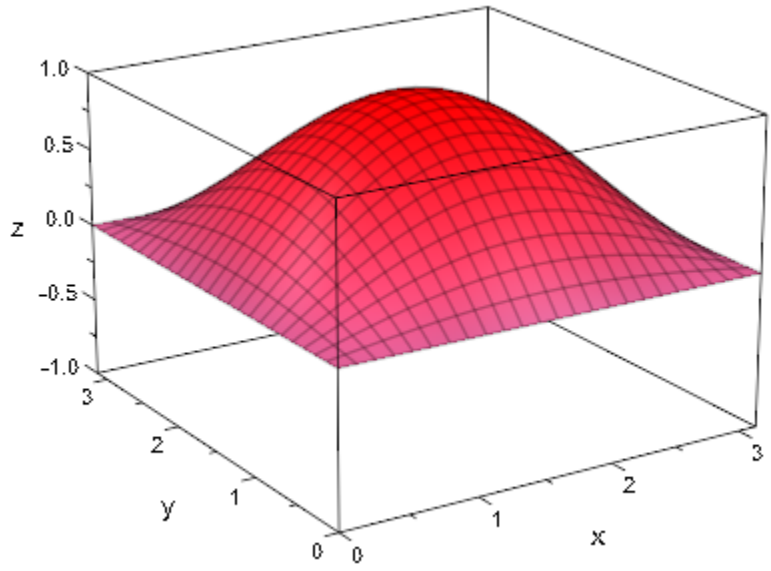
A static function plot in 3D requires two ranges (for the *x* and the *y* variable):

```
plot(plot::Function3d(sin(x)*sin(y), x = 0 .. PI, y = 0..PI))
```



Now, a third equation `a = `a_{min}`..`a_{max}`` triggers an animation:

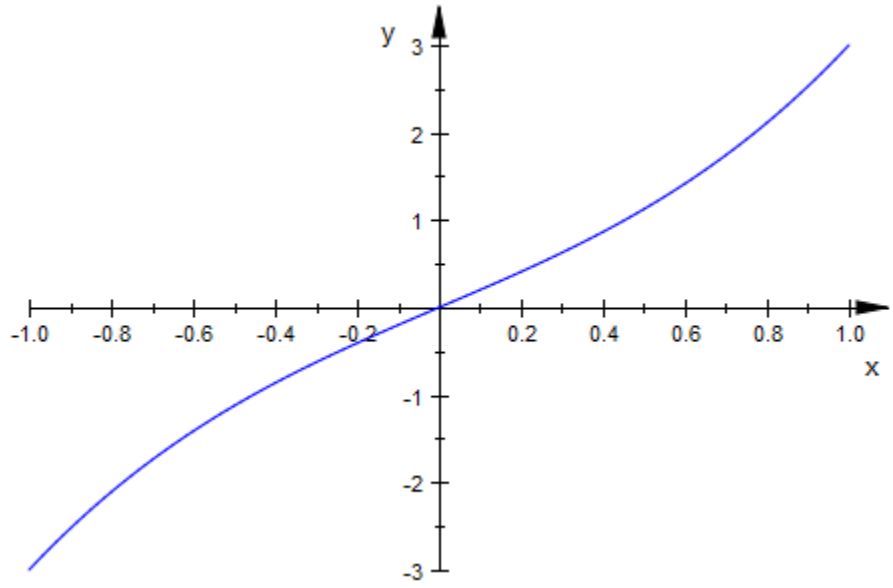
```
plot(plot::Function3d(sin(x + a)*sin(y - a), x = 0 .. PI,
                      y = 0..PI, a = 0..PI))
```



Example 2

We define an animated 2D function plot:

```
f := plot::Function2d(x^3 + a*x, x = -1..1, a = 0..2):
plot(f):
```

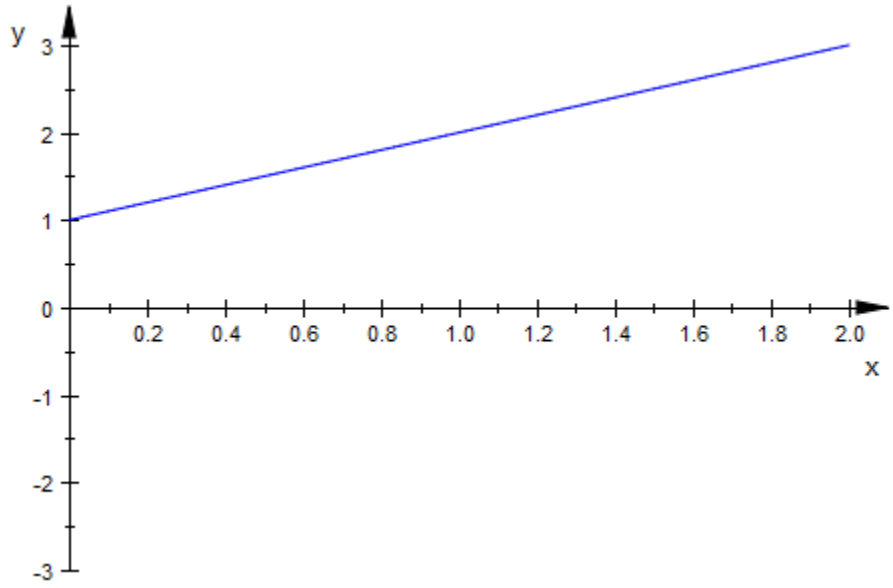


We swap the role of the independent variable x and the animation parameter a:

```
[f::XName, f::ParameterName] := [f::ParameterName, f::XName]:
[f::XRange, f::ParameterRange] :=
                                [f::ParameterRange, f::XRange]:
```

The function now is drawn as a function of a for various values of the "time" x:

```
plot(f)
```




```
delete f:
```

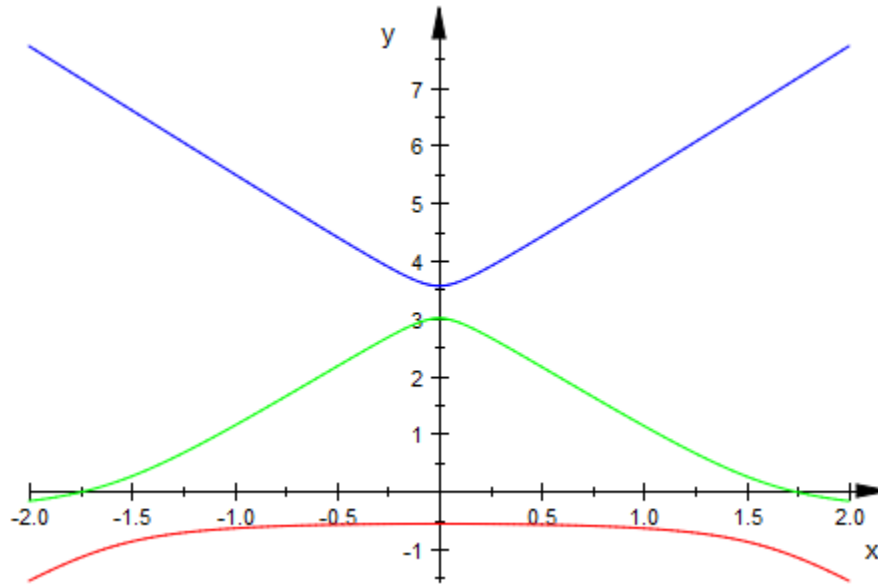
Example 3

We demonstrate the use of procedures in the definition of animated functions.

We wish to plot the eigenvalues of a matrix that depends on two parameters x and a. The eigenvalues are computed numerically in the procedure `eigenvals`. This procedure uses `option remember`, because it is called thrice with the same arguments by the procedures f_1, f_2, f_3 that produce the smallest, the middle, and the largest eigenvalue, respectively, as functions of the parameters x and a:

```
eigenvals :=
   proc(x, a)
     option remember;
     local A;
   begin
     A:= matrix([[1,  a,  x ],
                 [a,  2, a*x],
                 [x, a*x, 3 ]]):
```

```
    sort(numeric::eigenvalues(A)):
  end_proc:
f1:= (x, a) -> eigenvals(x, a)[1]:
f2:= (x, a) -> eigenvals(x, a)[2]:
f3:= (x, a) -> eigenvals(x, a)[3]:

plot(plot::Function2d(f1, x = -2..2, a = 0..2,
                      Color = RGB::Red),
     plot::Function2d(f2, x = -2..2, a = 0..2,
                      Color = RGB::Green),
     plot::Function2d(f3, x = -2..2, a = 0..2,
                      Color = RGB::Blue)):
```




```
delete eigenvals, f1, f2, f3:
```

See Also

MuPAD Functions

Frames | TimeBegin | TimeEnd | TimeRange | VisibleAfter | VisibleAfterEnd | VisibleBefore | VisibleBeforeBegin | VisibleFromTo

More About

- "The Number of Frames and the Time Range"

Points2d, Points3d

List of 2D points

Value Summary

`Points2d`, `Points3d`	Mandatory	List of 2D points

Graphics Primitives

Objects	Default Values
`plot::PointList2d`, `plot::PointList3d`, `plot::Polygon2d`, `plot::Polygon3d`	

Description

`Points2d` is the list of 2D points in objects of type `plot::PointList2d` and `plot::Polygon2d`, respectively.

`Points3d` is the list of 3D points in objects of type `plot::PointList3d` and `plot::Polygon3d`, respectively.

One usually defines such an object `p`, say, via

`p := plot::PointList2d([[x1, y1], [x2, y2], …])` or

`p := plot::Polygon2d([[x1, y1], [x2, y2], …])`, respectively.

Internally, the points are stored as the attribute

`Points2d = [[x1, y1], [x2, y2], …]`

and can be accessed via the slot call `p::Points2d`. Assigning a new list to `p::Points2d` changes the object `p` accordingly.

The corresponding statements hold for 3D point lists and polygons.

The points in the list `Points2d` may consist of lists with 2 elements (the x and y coordinates) or of lists with 3 elements (the x and y coordinates and the RGB color of the point).

The points in the list `Points3d` may consist of lists with 3 elements (the x, y, and z coordinates) or of lists with 4 elements (the x, y, z coordinates and the RGB/RGBa color of the point).

If you specify the color of one point, you must specify the colors of all other points in the list. See "Example 2" on page 24-1457.

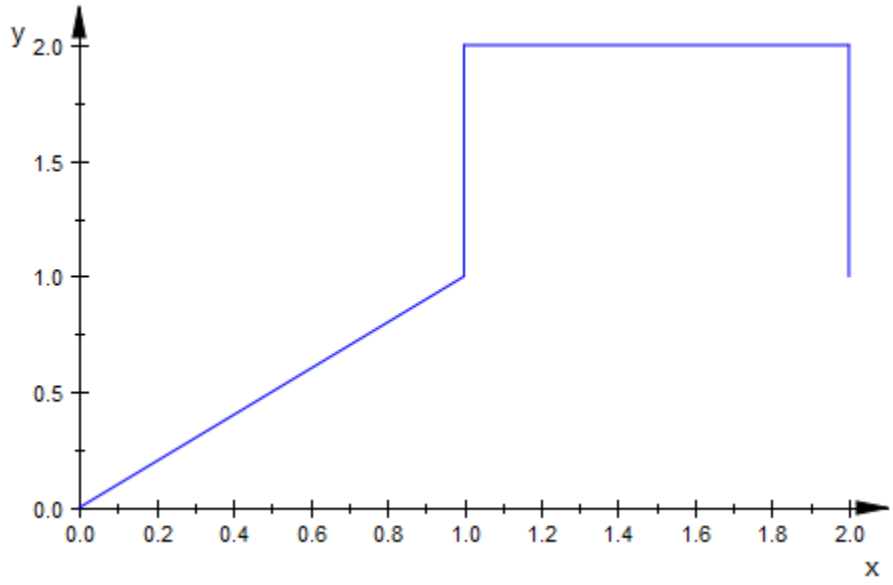
The points in the lists `Points2d` and `Points3d` can be animated.

Examples

Example 1

We define a 2D polygon with 5 points:

```
p := plot::Polygon2d([[0, 0], [1, 1], [1, 2], [2, 2], [2, 1]]):
plot(p):
```



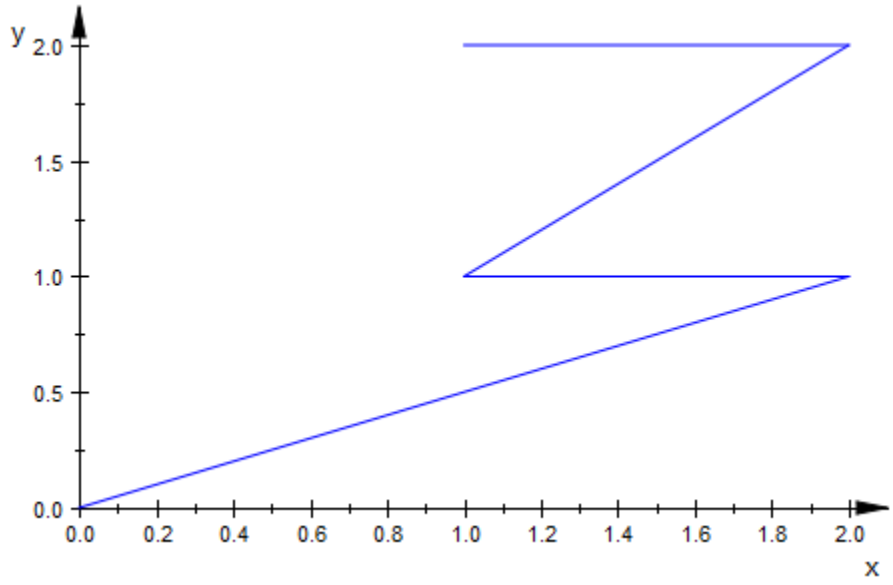
The points in the polygon can be accessed as the `Points2d` attribute:

```
pts := p::Points2d
```

$$[[0, 0], [1, 1], [1, 2], [2, 2], [2, 1]]$$

We change the polygon be assigning a new point list:

```
p::Points2d := [pts[1], pts[5], pts[2], pts[4], pts[3]]:
plot(p):
```



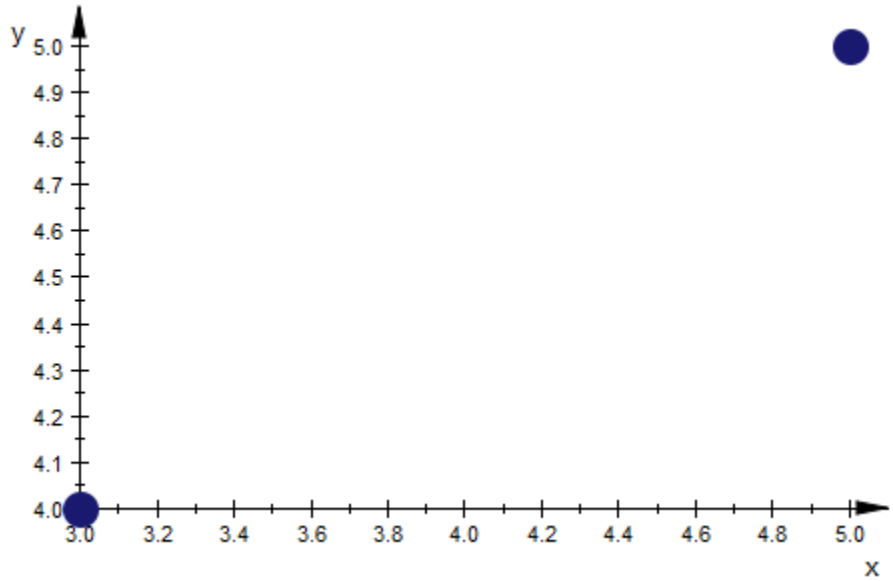

```
delete p, pts:
```

Example 2

`Points2d` and `Points3d` allow you to specify the colors of the points. For example, the following list contains two points. The `plot` function uses the default color for both points on the plot:

```
Coords := [[3, 4], [5, 5]];
plotCoords := plot::PointList2d(Coords):
plot(plotCoords, PointSize = 5)
```

```
[[3, 4], [5, 5]]
```



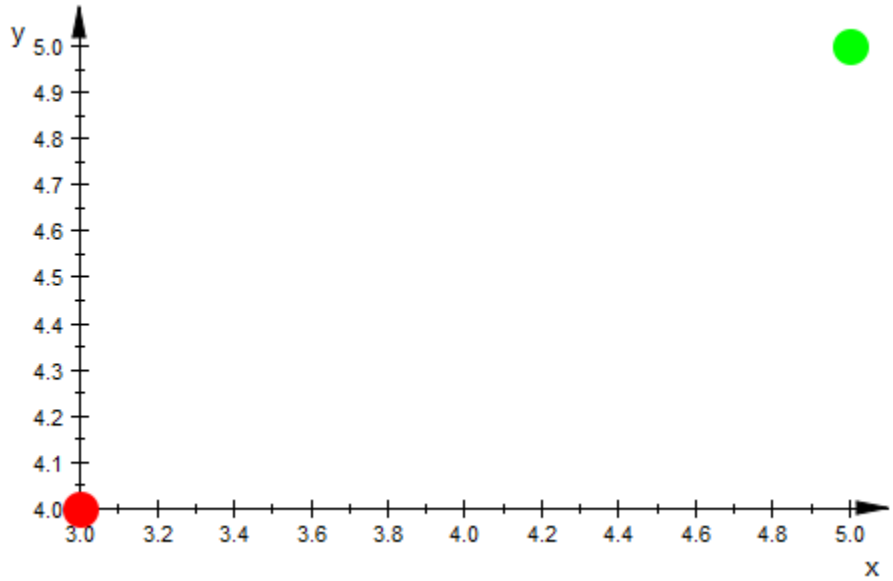
To access and modify the list of points, use `Points2d`. For example, include the color of each point in the list:

```
plotCoords::Points2d := [[3, 4, RGB::Red], [5, 5, RGB::Green]]
```

```
[[3, 4, [1.0, 0.0, 0.0]], [5, 5, [0.0, 1.0, 0.0]]]
```

Now the first point appears in red, and the second point appears in green:

```
plot(plotCoords, PointSize = 5)
```



If you specify the color of one point, you must also specify the colors of all other points in the list:

```
plotCoords::Points2d := [[3, 4, RGB::Red], [5, 5]]
```

```
Error: The attribute 'Points2d' in the 'PointList2d' object must be a list of lists of
```

Position, PositionX, PositionY, PositionZ

Positions of cameras, lights, and text objects

Value Summary

`Position`	Library wrapper for "[`PositionX`, `PositionY`]" (2D), "[`PositionX`, `PositionY`, `PositionZ`]" (3D)	List of 2 or 3 expressions, depending on the dimension
`PositionX`, `PositionY`, `PositionZ`	Mandatory	MuPAD expression

Graphics Primitives

Objects	Default Values
`plot::Camera`, `plot::DistantLight`, `plot::PointLight`, `plot::SpotLight`, `plot::Text2d`, `plot::Text3d`	
`plot::Point2d`	`Position`: [0, 0] `PositionX`, `PositionY`: 0
`plot::Plane`, `plot::Point3d`, `plot::Reflect3d`	`Position`: [0, 0, 0] `PositionX`, `PositionY`, `PositionZ`: 0

Description

`Position` determines the positions of cameras, lights, and text objects.

`PositionX` etc. refer to the single coordinate values of the position.

The attribute `Position` refers to the location of a camera taking pictures of a 3D scene. Its a value is a list or vector of coordinates.

Also the position of light sources illuminating the 3D scene is set by `Position`.

Further, `Position` determines the coordinates where text objects are to be placed.

These attributes can be animated. Animating a camera position one can realize a flight through a 3D scene.

By default, the position of lights is given in model coordinates that have nothing to do with the camera that is used to view the scene.

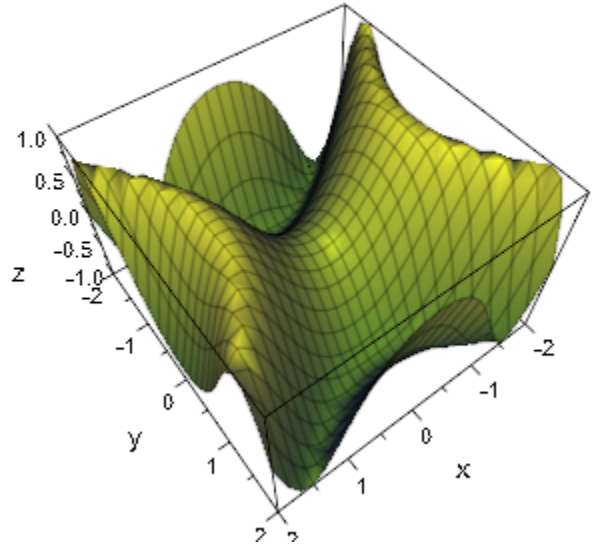
The attribute `CameraCoordinates` also allows to position a light relative to the camera. In particular, the light moves automatically, when the camera is moved.

Examples

Example 1

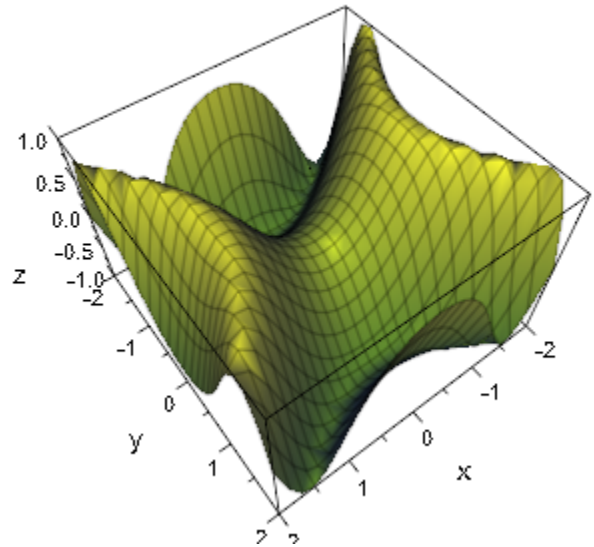
We define a 3D scene consisting of a function, a distant light, and a camera. The light shines from the direction of the camera:

```
f := plot::Function3d(sin(x^2 - y^2), x = -2..2, y = -2..2,
                      Color = RGB::White):
light := plot::DistantLight([3, 4, 5], [0, 0, 0], 0.75,
                            Color = RGB::Yellow):
camera := plot::Camera([3, 4, 5], [0, 0, 0], 0.25*PI):
plot(f, light, camera)
```



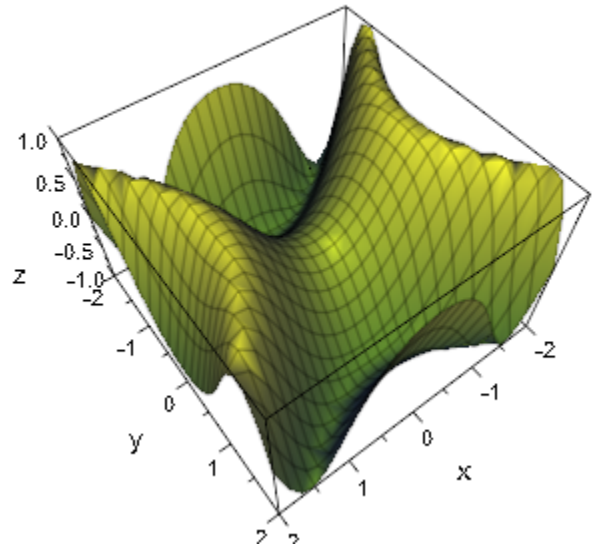
We animate the camera position but keep the light position fixed:

```
camera::Position := [3*sqrt(2)*cos(a + PI/4),
                     4*sqrt(2)*sin(a + PI/4),
                     5*(0.7 + 0.3*cos(2*a))]:
camera::ParameterName := a:
camera::ParameterRange := 0..2*PI:
plot(f, light, camera)
```



Using the same objects, we fix the camera and animate the light position:

```
camera::Position := [3, 4, 5]:
camera::Frames := 1:
light::Position := [3*sqrt(2)*cos(a + PI/4),
                    4*sqrt(2)*sin(a + PI/4),
                    5]:
light::ParameterName := a:
light::ParameterRange := 0..2*PI:
plot(f, light, camera)
```




```
delete f, light, camera:
```

See Also

MuPAD Functions

`CameraCoordinates` | `CameraDirection` | `FocalPoint` | `SpotAngle` | `Target` | `ViewingAngle`

Radius

Radius of circles, spheres etc.

Value Summary

Mandatory	MuPAD expression

Graphics Primitives

Objects	Radius Default Values
`plot::Circle2d`, `plot::Circle3d`, `plot::Cylinder`, `plot::Dodecahedron`, `plot::Hexahedron`, `plot::Icosahedron`, `plot::MuPADCube`, `plot::Octahedron`, `plot::Piechart2d`, `plot::Piechart3d`, `plot::Prism`, `plot::Sphere`, `plot::Tetrahedron`	1
`plot::Waterman`	

Description

`Radius` defines the radius of circles (`plot::Circle2d` and `plot::Circle3d`, respectively), spheres (`plot::Sphere`), cylinders (`plot::Cylinder`), circumcircles of regular bases of prisms (`plot::Prism`) and pie charts (`plot::Piechart2d` and `plot::Piechart3d`, respectively). Also polyhedra such as `plot::Dodecahedron` use this attribute to set their size.

Examples

Example 1

We generate a sphere around the origin with radius 2:

```
s := plot::Sphere(2, [0, 0, 0])
```

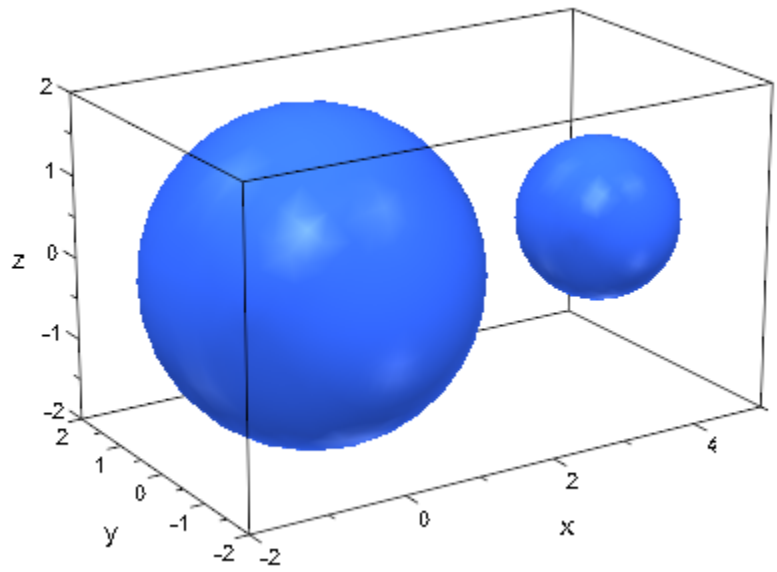
```
plot::Sphere(2, [0, 0, 0])
```

The first argument in `plot::Sphere` is the radius, the second argument is the center. Internally, these values are stored as the attributes `Radius` and `Center`, respectively. We can access the objects' attributes and change them:

```
s::Radius, s::Center
```

```
2, [0, 0, 0]
```

```
s2 := plot::copy(s):
s2::Center := [4, 0, 0]:
s2::Radius := 1:
plot(s, s2):
```

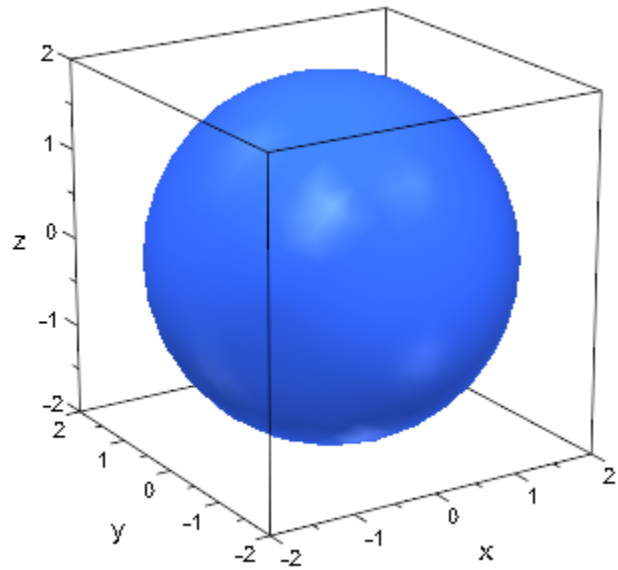



```
delete s, s2:
```

Example 2

The attribute `Radius` can be animated:

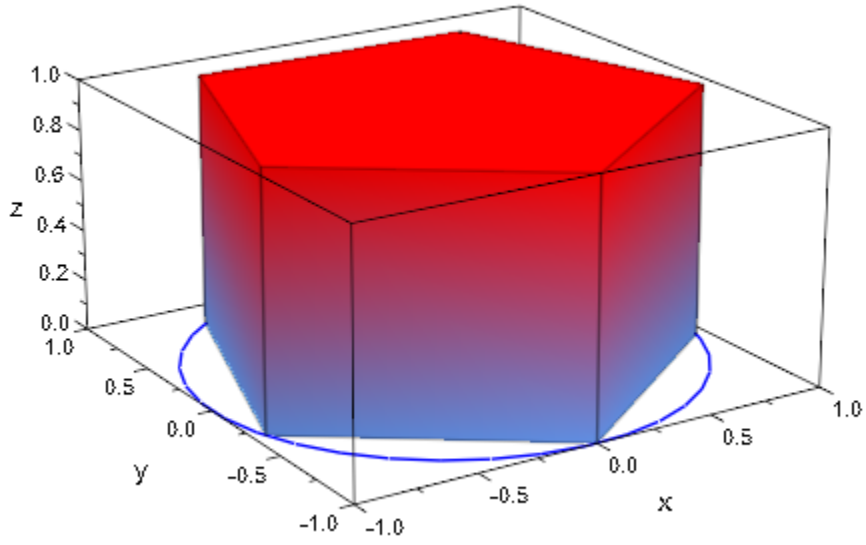
```
plot(plot::Sphere(a, [0, 0, 0], a = 1..2)):
```



Example 3

For a prism, the attribute `Radius` determines the radius of the circumcircle of its regular bases:

```
plot(plot::Prism(1,Edges=5), plot::Circle3d(1)):
```



See Also

MuPAD Functions

`Base` | `Center` | `Normal` | `SemiAxes` | `Top`

RadiusFunction

Radius of a tube plot

Value Summary

Mandatory	Arithmetical expression or function

Graphics Primitives

Objects	RadiusFunction Default Values
`plot::Tube`	`1/10`

Description

`RadiusFunction` is the internal name of the radius function in `plot::Tube`.

With `RadiusFunction = r(t)`, `plot::Tube` will draw (part of) a circle of radius `r(t)` at the point $(x(t), y(t), z(t))$ around the central curve.

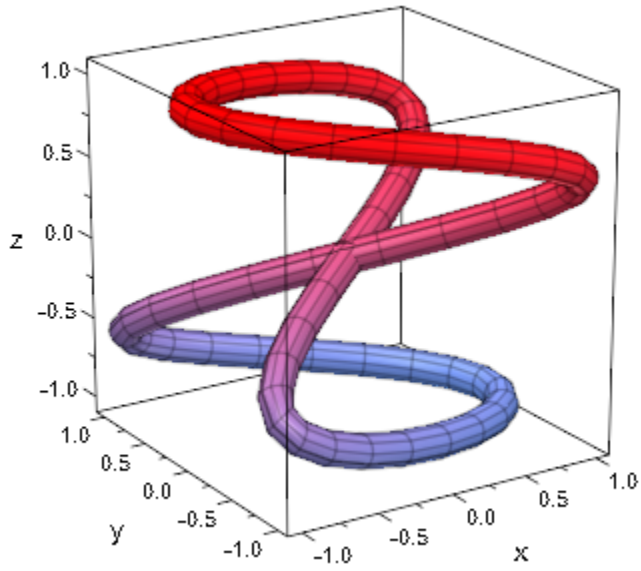
Usually, a user will have no need to access `RadiusFunction` directly, since it is set by `plot::Tube` directly.

Examples

Example 1

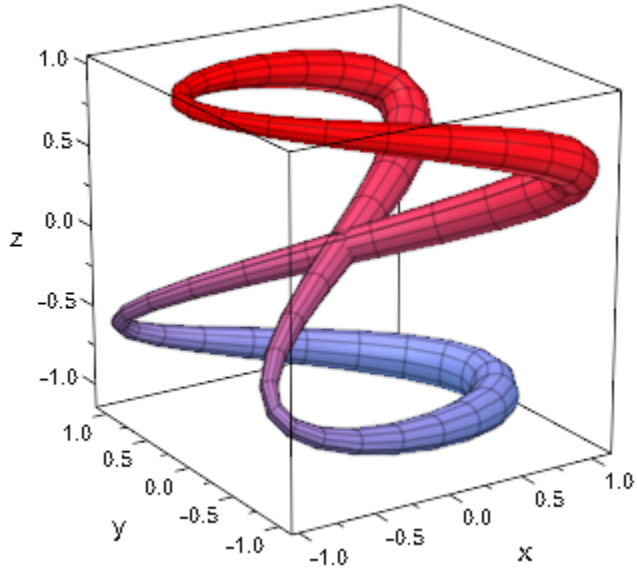
By default, `plot::Tube` uses a constant radius of $\frac{1}{10}$:

```
t := plot::Tube([sin(3*x), sin(2*x), sin(x)], x = 0..2*PI):
plot(t)
```



To change this default, either set some other radius when creating the tube plot (see the documentation of `plot::Tube` for this) or set `RadiusFunction`:

```
t::RadiusFunction := (1+sin(3*x)/2)/10:
plot(t)
```


1.0
0.5
0.0
-0.5
-1.0
z
1.0
0.5
0.0
-0.5
-1.0
y
-1.0
-0.5
0.0
0.5
1.0
x

RationalExpression

Rational expression in a rootlocus plot

Value Summary

Mandatory	MuPAD expression

Graphics Primitives

Objects	RationalExpression Default Values
`plot::Rootlocus`	

Description

`RationalExpression` is the internal name of the expression whose roots are depicted by `plot::Rootlocus`.

A rootlocus plot depicts the roots of a rational function `p(z, u)` in the complex plane, depending on a parameter `u`. The expression `p(z, u)` is stored as the attribute `RationalExpression` in the rootlocus object.

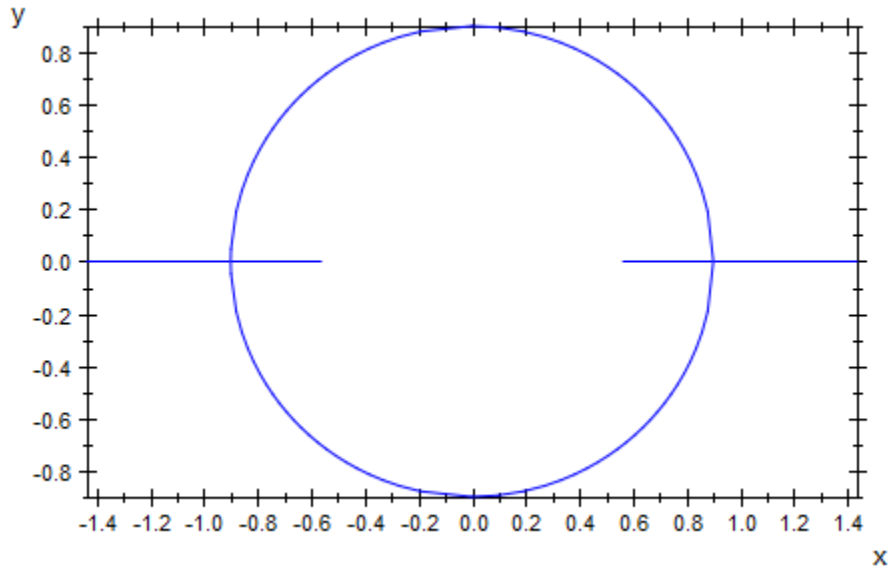
Usually, a user will have no need to access the attribute `RationalExpression`, since it is set by `plot::Rootlocus` directly.

Examples

Example 1

We define a rootlocus plot:

```
r:= plot::Rootlocus(z^2 - 2*u*z + 0.81, u = -1..1):
plot(r)
```



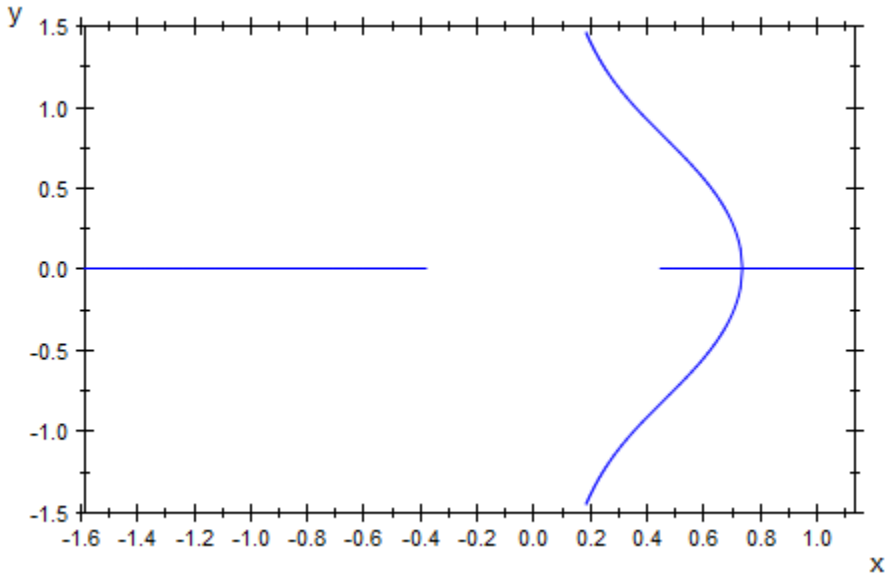
The function z^2 - 2 u z + 0.81 is stored as the attribute `r::RationalExpression` in the object `r`:

```
r::RationalExpression
```

$$z^2 - 2\,u\,z + 0.81$$

We can redefine this attribute:

```
r::RationalExpression:= z^3 - 2*u*z + 0.81:
plot(r)
```


y
1.5
1.0
0.5
0.0
-0.5
-1.0
-1.5
-1.6 -1.4 -1.2 -1.0 -0.8 -0.6 -0.4 -0.2 0.0 0.2 0.4 0.6 0.8 1.0
x

Scale, ScaleX, ScaleY, ScaleZ

Scaling factors

Value Summary

`Scale`	Library wrapper for "[`ScaleX`, `ScaleY`]" (2D), "[`ScaleX`, `ScaleY`, `ScaleZ`]" (3D)	List of 2 or 3 expressions, depending on the dimension
`ScaleX`, `ScaleY`, `ScaleZ`	Optional	MuPAD expression

Graphics Primitives

Objects	Default Values
`plot::Scale2d`	`Scale`: [1, 1] `ScaleX`, `ScaleY`: 1
`plot::Scale3d`	`Scale`: [1, 1, 1] `ScaleX`, `ScaleY`, `ScaleZ`: 1

Description

`Scale` defines the scaling factors used by `plot::Scale2d` and `plot::Scale3d`.

`ScaleX` etc. correspond to the factors in the single coordinate directions.

The scaling objects `plot::Scale2d` and `plot::Scale3d` apply the scaling transformation $x \to A\,x$ with the matrix $A = diag(s_x, s_y)$ in 2D and $A = diag(s_x, s_y, s_z)$ in 3D, repectively.

`Scale` is the list `[`s_x`, `s_y`]` resp. `[`s_x`, `s_y`, `s_z`]` of the scaling factors. The attributes `ScaleX` etc. correspond to s_x etc.

These attributes can be animated.

Examples

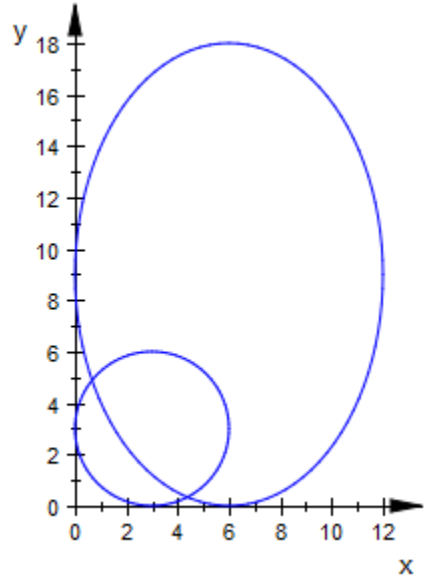
Example 1

We start with a 2D circle:

```
c := plot::Circle2d(3, [3, 3]):
```

We apply a scaling transformation:

```
S := plot::Scale2d([2, 3], c):
plot(c, S):
```



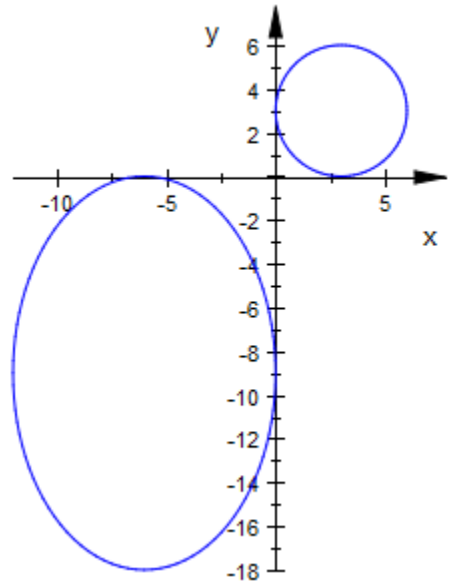
The scaling factors are stored as the `Scale` attribute in the scaling object `S`:

```
S::Scale, S::ScaleX, S::ScaleY
```

$$[2, 3],\ 2,\ 3$$

We change the scaling factors:

```
S::Scale := [-2, -3]:
plot(c, S):
```




```
delete c, S:
```

See Also

MuPAD Functions

`Matrix2d` | `Matrix3d` | `Shift`

SemiAxes, SemiAxisX, SemiAxisY, SemiAxisZ

Semi axes of ellipses and ellipsods

Value Summary

`SemiAxes`	Library wrapper for "[`SemiAxisX`, `SemiAxisY`]" (2D), "[`SemiAxisX`, `SemiAxisY`, `SemiAxisZ`]" (3D)	List of two or three real-valued expressions
`SemiAxisX`, `SemiAxisY`, `SemiAxisZ`	Mandatory	MuPAD expression

Graphics Primitives

Objects	Default Values
`plot::Ellipsoid`	`SemiAxes`: [1, 2, 3] `SemiAxisX`: 1 `SemiAxisY`: 2 `SemiAxisZ`: 3
`plot::Ellipse2d`	`SemiAxes`: [2, 1] `SemiAxisX`: 2 `SemiAxisY`: 1
`plot::Ellipse3d`	`SemiAxisX`: 2 `SemiAxisY`: 1
`plot::Arc2d`	`SemiAxes`: [1, 1] `SemiAxisX`, `SemiAxisY`: 1

Objects	Default Values
`plot::Arc3d`	`SemiAxisX`, `SemiAxisY`: 1

Description

`SemiAxes` determines the lengths of the semi axes of ellipses in 2D and ellipsoids in 3D.

`SemiAxes = [r_x, r_y]` sets the lengths r_x, r_y of the semi axes of an ellipse in 2D.

`SemiAxes = [r_x, r_y, r_z]` sets the lengths r_x, r_y, r_z of the semi axes of an ellipsoid in 3D.

`SemiAxisX = r_x`, `SemiAxisY = r_y`, `SemiAxisZ = r_z` refer to the semi axis in the x, y, and z direction, respectively.

The values of these attributes can be animated.

Examples

Example 1

We create an ellipse around the origin with semi axes 1 and 2:

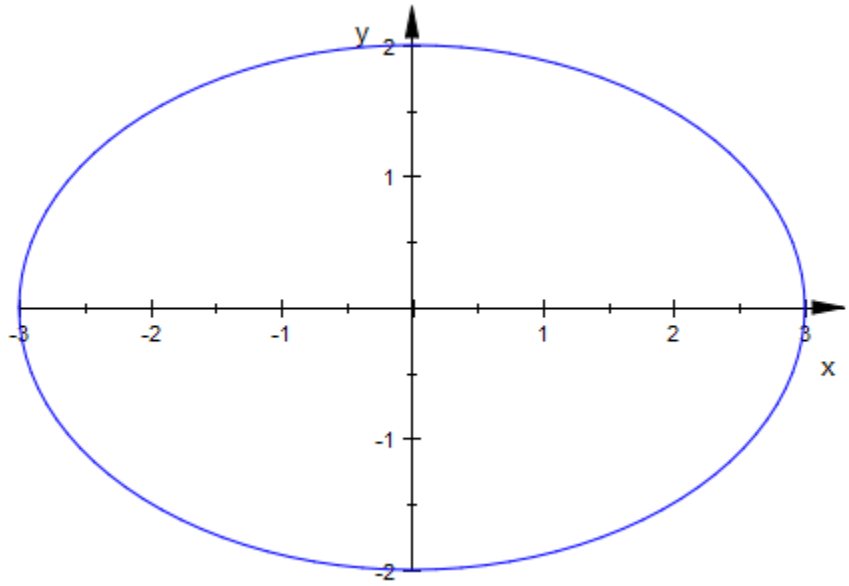
```
e := plot::Ellipse2d(1, 2, [0, 0]):
```

The first two arguments in `plot::Ellipse2d` are the semi axes. Internally, they are stored as the attributes `SemiAxisX` and `SemiAxisY` and can be changed by assigning new values:

```
e::SemiAxisX, e::SemiAxisY, e::SemiAxes
```

```
1, 2, [1, 2]
```

```
e::SemiAxes := [3, 2]:
plot(e):
```

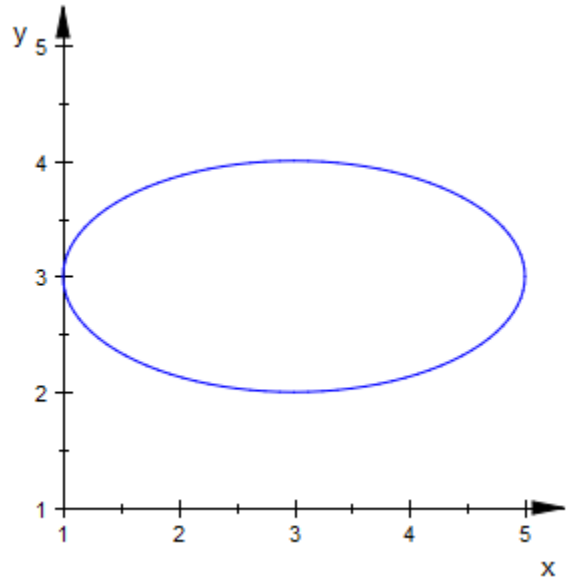



```
delete e:
```

Example 2

`SemiAxes` can be animated:

```
plot(plot::Ellipse2d(a, 3 - a, [3, 3], a = 1..2)):
```



See Also

MuPAD Functions
`Radius`

Shift, ShiftX, ShiftY, ShiftZ

Shift vector

Value Summary

`Shift`	Library wrapper for "[`ShiftX`, `ShiftY`]" (2D), "[`ShiftX`, `ShiftY`, `ShiftZ`]" (3D)	List of 2 or 3 expressions, depending on the dimension
`ShiftX`, `ShiftY`, `ShiftZ`	Optional	MuPAD expression

Description

`Shift` is the shift vector in transformation objects. `ShiftX` etc. refer to the single components of this vector.

The general transformation objects `plot::Transform2d` and `plot::Transform3d` allow to apply the affine-linear transformation $x \to A\,x + b$ to 2D and 3D objects, respectively. The shift vector b can be accessed and changed via the attribute `Shift`.

Special transformation objects such as `plot::Translate2d`, `plot::Translate3d` correspond to special matrices A and shifts b. They also allow to access and change the shift vector by the attribute `Shift`.

When setting the `Shift` attribute, matrices, arrays, and lists with 2 or 3 elements are accepted. Internally, however, the shift data are always stored as the list `[b1, b2]` in 2D or `[b1, b2, b3]` in 3D, respectively. When reading the vector by a slot access, this list is returned.

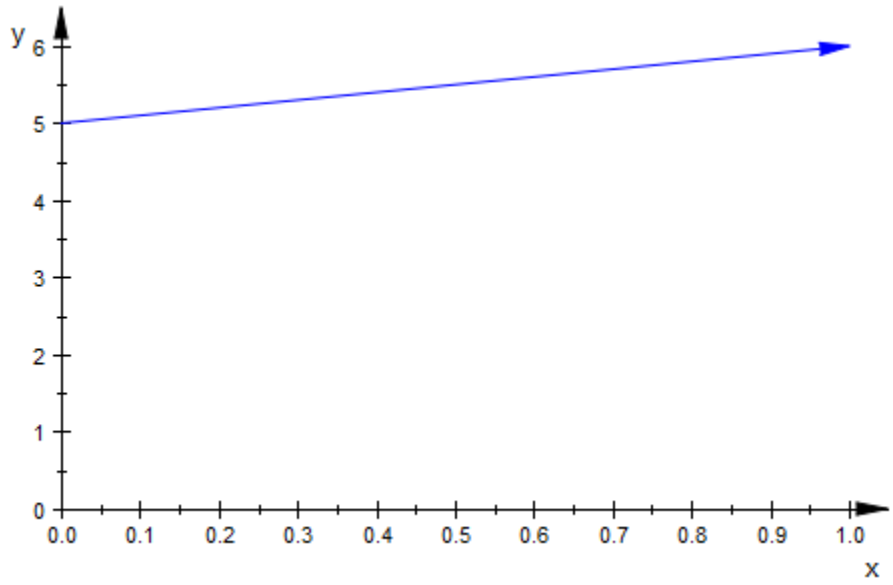
The entries of `Shift` can be animated.

Examples

Example 1

We move an arrow along the y-axis:

```
T := plot::Translate2d([0, a], a = 0..5,
                       plot::Arrow2d([0, 0], [1, 1])):
plot(T)
```



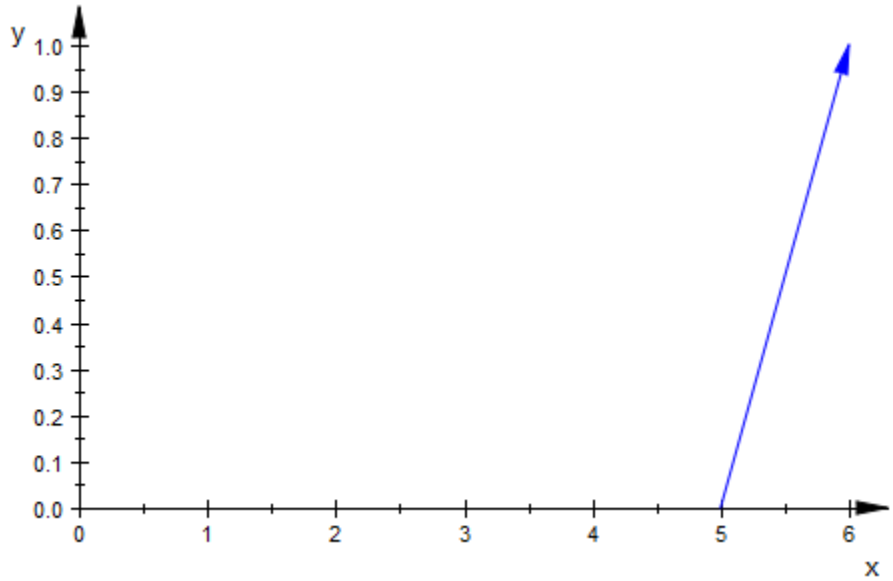
The `Shift` vector is the first argument in the call above. It is stored in the corresponding slot of the translation object `T`:

```
T::Shift
```

$$[0, a]$$

We change the shift vector:

```
T::Shift := [a, 0]:
plot(T)
```




```
delete T:
```

See Also

MuPAD Functions
`Matrix2d` | `Matrix3d` | `Scale`

Size

Size of a point list

Value Summary

Optional	MuPAD expression

Graphics Primitives

Objects	Size Default Values
`plot::QQplot`	

Description

`Size` represents the number of plot points in a `plot::QQplot`.

A `plot::QQplot` accepts two data lists, displaying a set of plot points with coordinate values given by quantile values of the data. By default, the number of plot points is chosen as the minimum of the sizes of the two data lists. In principle, however, the number of plot points can be chosen independently of the data sizes. With `Size` = n, the number of plot points of the QQ plot can be set to any positive integer value n.

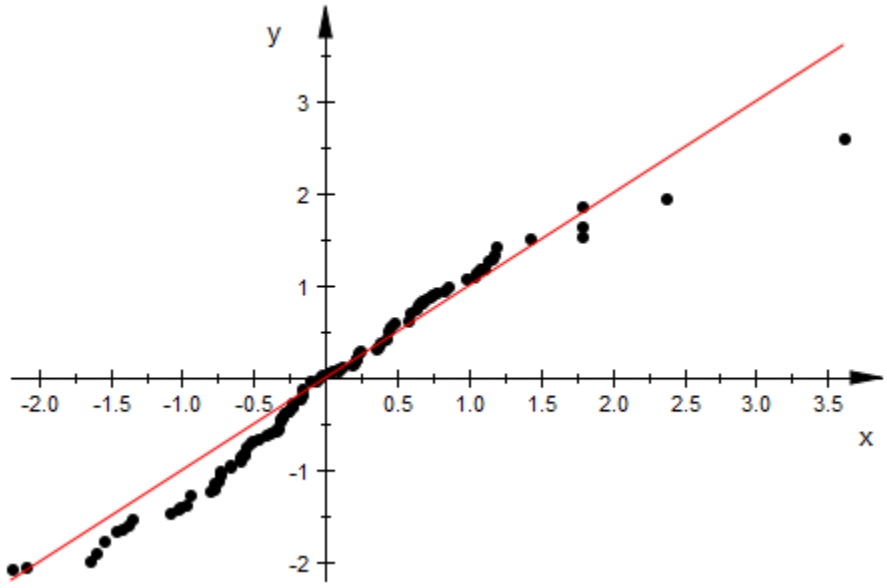
The value of `Size` can be animated.

Examples

Example 1

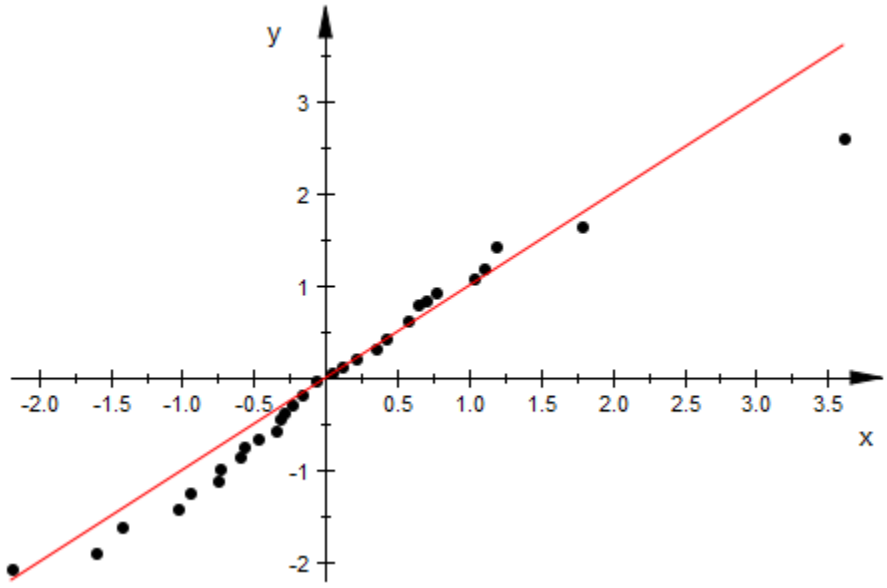
We create a QQ plot:

```
data1 := [stats::normalRandom(0, 1)() $ k = 1..100]:
data2 := [stats::normalRandom(0, 1)() $ k = 1..200]:
qq := plot::QQplot(data1, data2):
plot(qq)
```



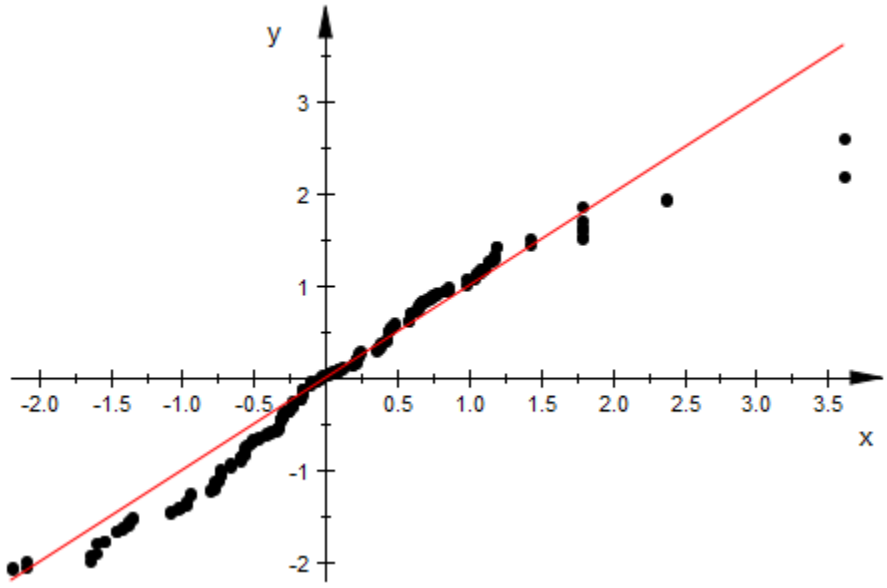
By default, the minimum of the data sizes is chosen as the number of plot points in the plot (i.e., `Size` = 100 in this case). We reduce the number of plot points by setting the value of `Size` explicitly:

```
qq::Size := 30:
plot(qq)
```



The number of plot points can also be specified directly by passing the attribute `Size` = *n*. In the following graphics, this value is animated:

```
plot(plot::QQplot(data1, data2, Size = n, n = 10..200));
```




```
delete data1, data2, qq:
```

Tangent1, Tangent1X, Tangent1Y, Tangent1Z, Tangent2, Tangent2X, Tangent2Y, Tangent2Z

First vector spanning parallelograms

Value Summary

`Tangent1`	Library wrapper for "[`Tangent1X`, `Tangent1Y`]" (2D), "[`Tangent1X`, `Tangent1Y`, `Tangent1Z`]" (3D)	List of 2 or 3 expressions, depending on the dimension
`Tangent1X`, `Tangent1Y`, `Tangent1Z`, `Tangent2X`, `Tangent2Y`, `Tangent2Z`	Mandatory	MuPAD expression
`Tangent2`	Library wrapper for "[`Tangent2X`, `Tangent2Y`]" (2D), "[`Tangent2X`, `Tangent2Y`, `Tangent2Z`]" (3D)	List of 2 or 3 expressions, depending on the dimension

Graphics Primitives

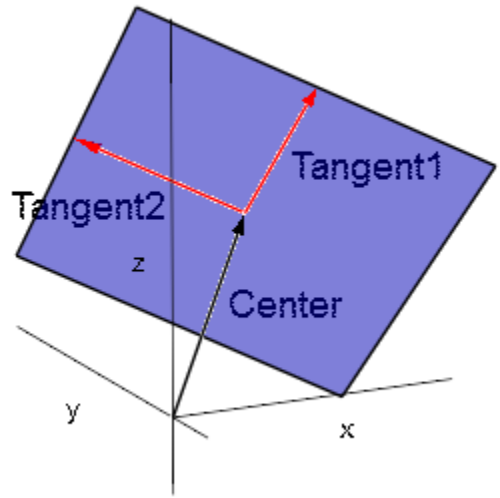
Objects	Default Values
`plot::Parallelogram3d`	`Tangent1`: [0, 1, 0] `Tangent1X`, `Tangent1Z`, `Tangent2Y`, `Tangent2Z`: 0 `Tangent1Y`, `Tangent2X`: 1 `Tangent2`: [1, 0, 0]
`plot::Parallelogram2d`	`Tangent1`: [0, 1] `Tangent1X`, `Tangent2Y`: 0

Objects	Default Values
	`Tangent1Y`, `Tangent2X`: 1 `Tangent2`: [1, 0]

Description

`Tangent1` and `Tangent2` determine the vectors spanning the parallelograms created by `plot::Parallelogram2d` and `plot::Parallelogram3d`.

Parallelograms created by `plot::Parallelogram2d` and `plot::Parallelogram3d` are specified by a vector defining the `Center` and two vectors `Tangent1` and `Tangent2` which span the plane of the parallelogram. The lengths of the "tangent" vectors are half the side lengths of the parallelogram:



Depending on the dimension, the vectors `Tangent1`, `Tangent2` are given by lists or vectors of two or three components.

The attributes `Tangent1X` etc. represent the x, y, z coordinates of these vectors.

The values of these attributes can be animated.

Examples

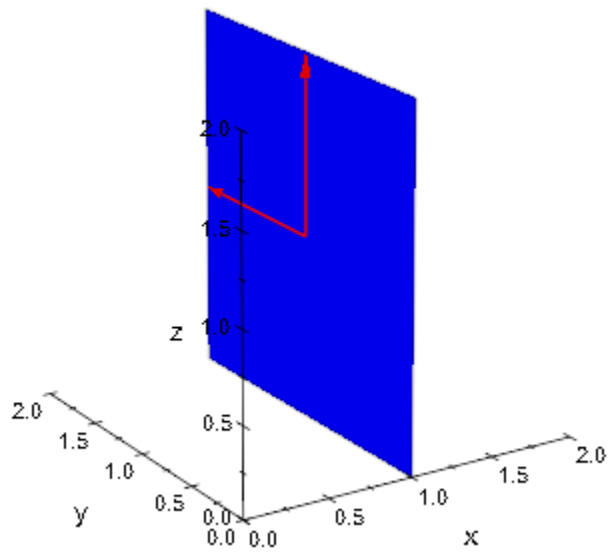
Example 1

The “tangent vectors” of a parallelogram are accessible via the slots `Tangent1` and `Tangent2`:

```
p := plot::Parallelogram3d([1, 1, 1], [0, 1, 0], [0, 0, 1],
                           Color = RGB::Blue):
p::Tangent1, p::Tangent2
```

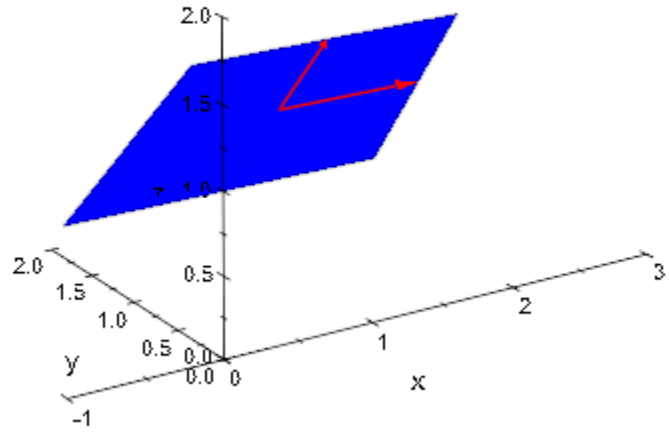
```
[0, 1, 0], [0, 0, 1]
```

```
plot(p,
     plot::Arrow3d([1, 1, 1], [1, 2, 1], Color = RGB::Red),
     plot::Arrow3d([1, 1, 1], [1, 1, 2], Color = RGB::Red),
     Axes = Origin, Scaling = Constrained):
```



We change the “tangent vectors”:

```
p::Tangent1 := [1, 0, 0]:  p::Tangent2 := [1, 1,  0]:
plot(p,
     plot::Arrow3d([1, 1, 1], [2, 1, 1], Color = RGB::Red),
     plot::Arrow3d([1, 1, 1], [2, 2, 1], Color = RGB::Red),
     Axes = Origin, Scaling = Constrained):
```




```
delete p:
```

See Also

MuPAD Functions
`Center`

Text

Text of a text object

Value Summary

Mandatory	String or function

Graphics Primitives

Objects	Text Default Values
`plot::Text2d`, `plot::Text3d`	

Description

The attribute `Text` represents the text of a text object. It may be a text string or a function generating a text string at runtime.

The `Text` attribute represents the text in text objects of type `plot::Text2d` and `plot::Text3d`. When creating a text object such as

```
t := plot::Text2d("hello world", [0, 0]),
```

the text is the first argument. Internally, it is stored as the attribute `Text = "hello world"` and can be accessed and changed via a slot call `t::Text`.

In most cases, the text is given as a string.

Note: Note that this string has to be quoted when changing it in the “property inspector” of the interactive graphics tool (see section Viewer, Browser, and Inspector: Interactive Manipulation of this document). If the string contains white space and the quotes are removed, the recalculation following the change will produce a syntax error!

A text given by a fixed string cannot be animated. Use a procedure to create animated texts.

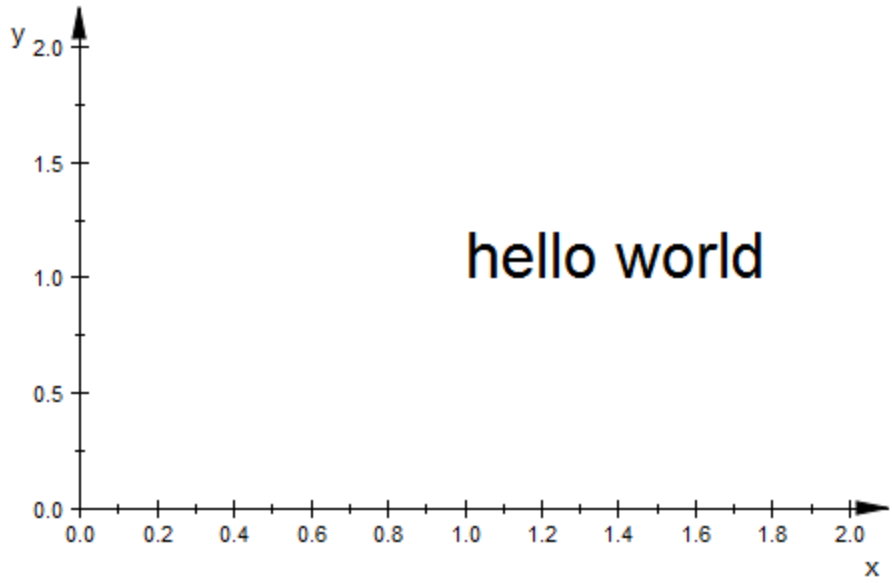
The attribute `Text` can be a procedure that is called at runtime with the animation parameter as the only input parameter. The return value is used as the text of the text object in the corresponding frame of the animation. If the result is not a string, `expr2text` is applied to the return value.

Examples

Example 1

Usually, a text is given by a string:

```
t := plot::Text2d("hello world", [1, 1], TextFont = [24]):
plot(t)
```

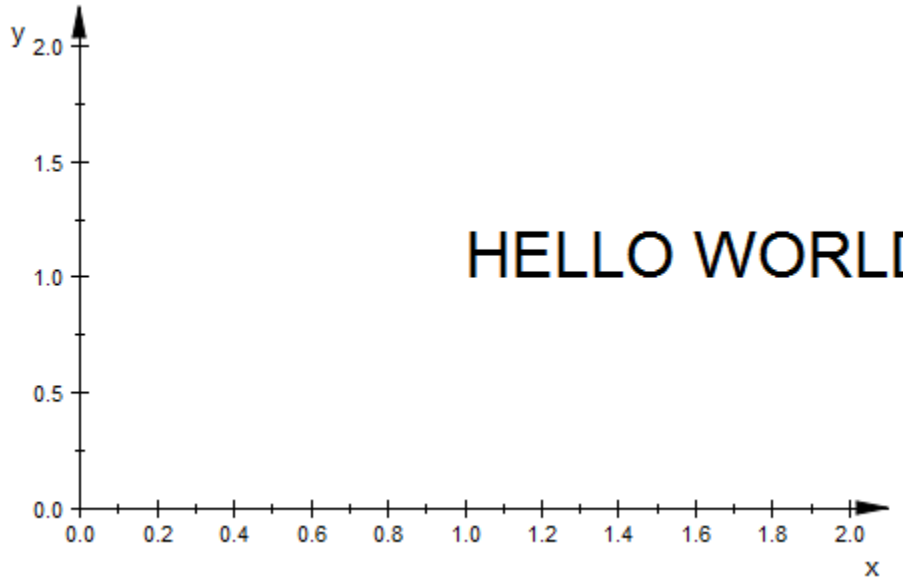


We access and change the text:

```
t::Text
```

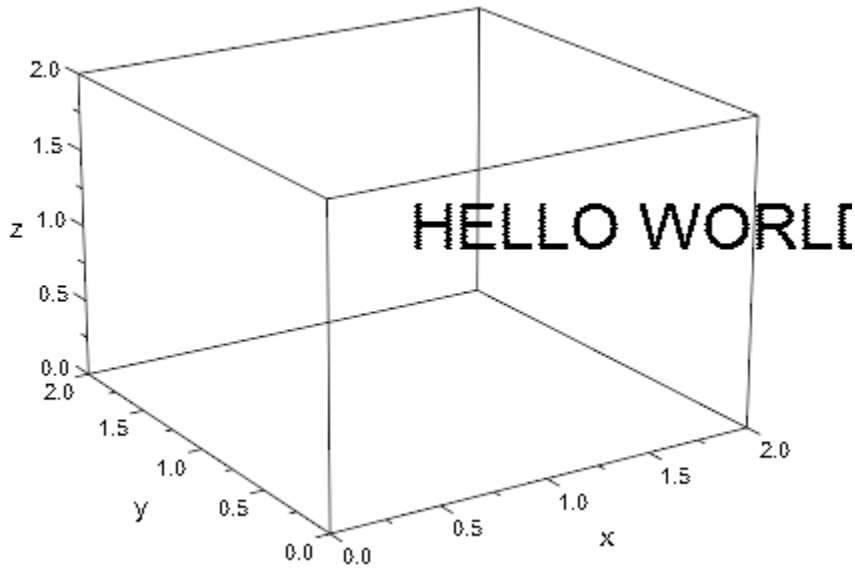
"hello world"

```
t::Text := "HELLO WORLD":
plot(t)
```



The same message in 3D:

```
plot(plot::Text3d("HELLO WORLD", [1, 1, 1],
                  TextFont = [24]))
```

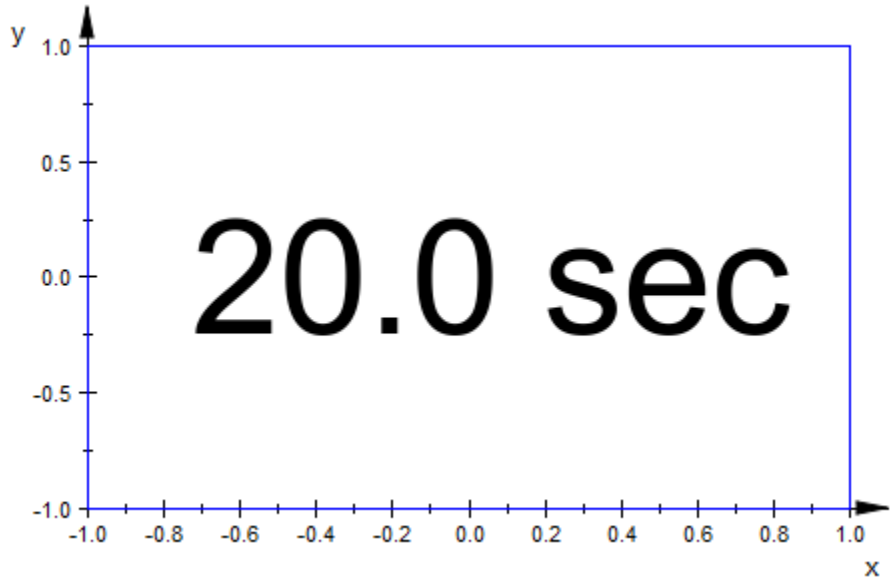



```
delete t:
```

Example 2

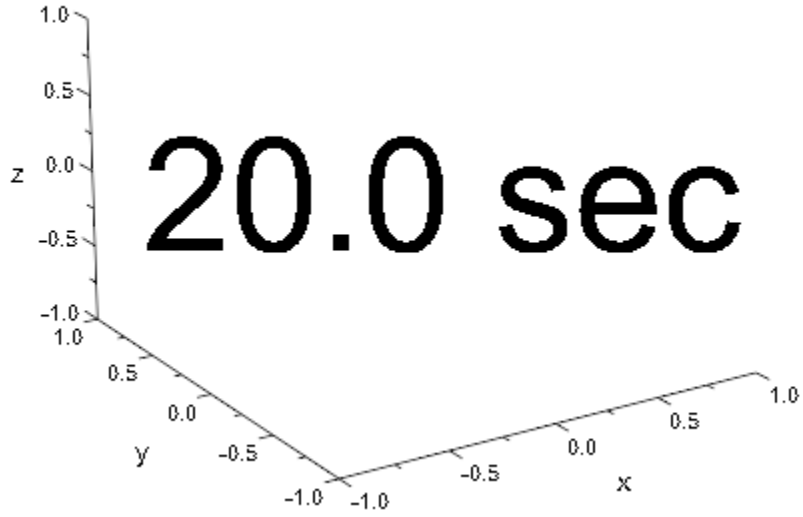
The text of a text object can be animated if the text string is provided by a procedure. We use `stringlib::formatf` to format the animation parameter that is passed to the procedure as a floating-point number for each frame of the animation:

```
plot(plot::Rectangle(-1..1, -1..1),
     plot::Text2d(a -> stringlib::formatf(a, 2, 5)." sec",
                  [0, 0], a = 0..20,
                  TextFont = [60],
                  HorizontalAlignment = Center,
                  VerticalAlignment = Center),
     Axes = Frame, Frames = 201, TimeRange = 0..20)
```



Here is the corresponding example in 3D:

```
plot(plot::Text3d(a -> stringlib::formatf(a, 2, 5)." sec",
                  [0, 0, 0], a = 0..20,
                  TextFont = [60],
                  HorizontalAlignment = Center,
                  VerticalAlignment = Center),
     Axes = Frame, Frames = 201, TimeRange = 0..20)
```



See Also

MuPAD Functions

`Billboarding` | `HorizontalAlignment` | `stringlib::formatf` | `TextFont` | `VerticalAlignment`

TextOrientation

Orientation of a 3D text

Value Summary

Optional	List of six real-valued expressions

Graphics Primitives

Objects	TextOrientation Default Values
`plot::Text3d`	[1, 0, 0, 0, 0, 1]

Description

`TextOrientation` defines the orientation of a text object of type `plot::Text3d`. Its orientation in 3 space is given by 2 directions. There is the “writing direction” from the first character of the text to the last. The direction from the bottom of the characters to their top shall be referred to as the “up direction”.

Together with the anchor point of the text (the attribute `Position` of a `plot::Text3d` object), these two directions define a 2 dimensional plane in 3D. You may regard this plane as the sheet onto which the text is written.

The value of `TextOrientation` has to be a list of 6 numerical values or expressions of the animation parameter. The first 3 components of this list define the “writing direction”, the last 3 components the “up direction”.

The length of these two vectors is irrelevant, only their directions matter. The lengths should not be zero, though.

Further, the “up direction” should be orthogonal to the “writing direction”. If this is not the case, the “up direction” is automatically replaced by the vector orthogonal to the “writing direction” that lies in the plane given by the original directions.

"Writing direction" and "up direction" should not be parallel.

Note: `TextOrientation` only has an effect in conjunction with the attribute `Billboarding = FALSE`.

`TextOrientation` can be animated.

The effect of `TextOrientation` is independent of the `HorizontalAlignment` and `VerticalAlignment` of the text relative to its anchor point (`Position`).

While `TextOrientation` is used for orienting 3D texts, `TextRotation` is used for rotating a 2D text of type `plot::Text2d`.

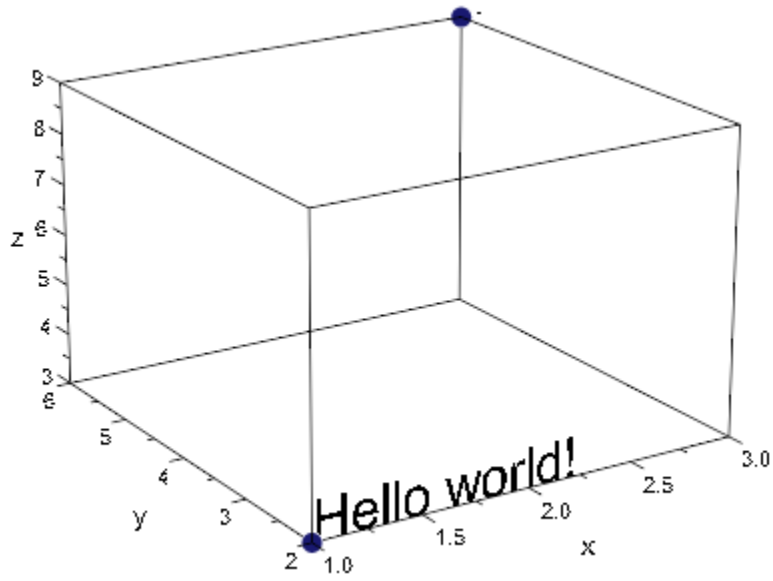
Examples

Example 1

The "writing direction" of the text object `text1` is rotated around an axis parallel to the *z*-axis. The "up direction" of its characters is the *z* direction.

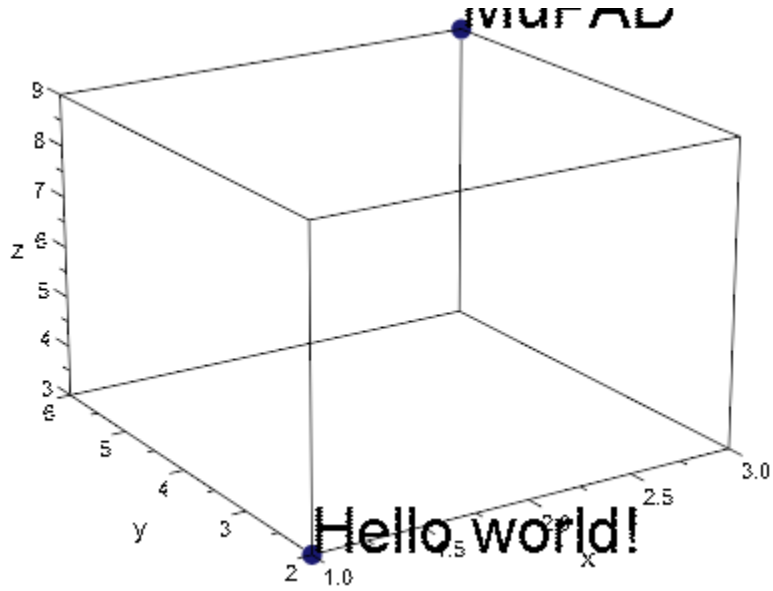
The "writing direction" of the text object `text2` is parallel to the *x* axis. The animated "up direction" is rotated around an axis pointing into the *x* direction:

```
p1 := plot::Point3d([1, 2, 3], PointSize = 3*unit::mm):
text1 := plot::Text3d("Hello world!", [1, 2, 3],
             TextOrientation = [cos(a), sin(a), 0, 0, 0, 1],
             a = 0..2*PI, TextFont = [24],
             Billboarding = FALSE):
p2 := plot::Point3d([3, 6, 9], PointSize = 3*unit::mm):
text2 := plot::Text3d("MuPAD", [3, 6, 9],
             TextOrientation = [1, 0, 0, 0, sin(a), cos(a)],
             a = 0..2*PI, TextFont = [24],
             Billboarding = FALSE):
plot(p1, text1, p2, text2)
```



When `Billboarding` is set to `TRUE`, `TextOrientation` does not have any effect:

```
text1::Billboarding := TRUE:
text2::Billboarding := TRUE:
plot(p1, text1, p2, text2)
```




```
delete p1, text1, p2, text2:
```

See Also

MuPAD Functions
`TextRotation`

TextRotation

Rotation of a 2D text

Value Summary

Optional	Real-valued expression (interpreted in radians)

Graphics Primitives

Objects	TextRotation Default Values
`plot::Integral`, `plot::Text2d`	`0`

Description

`TextRotation` sets the rotation angle of a 2D text object relative to the horizontal axis.

`TextRotation` rotates a text object of type `plot::Text2d`. around its anchor point (the attribute `Position` of a `plot::Text2d` object). Note that a `plot::Text2d` allows different alignments (`HorizontalAlignment`, `VerticalAlignment`) relative to this point.

The rotation angle in `TextRotation` = `angle` has to be entered in radians. If positive, the rotation is counterclockwise.

The rotation of the text refers to 'rotation on the screen'. It is invariant w.r.t. rescaling of the canvas, aspect ratio etc.

While `TextRotation` is used for rotating 2D texts, `TextOrientation` is used for rotating a 3D text of type `plot::Text3d`.

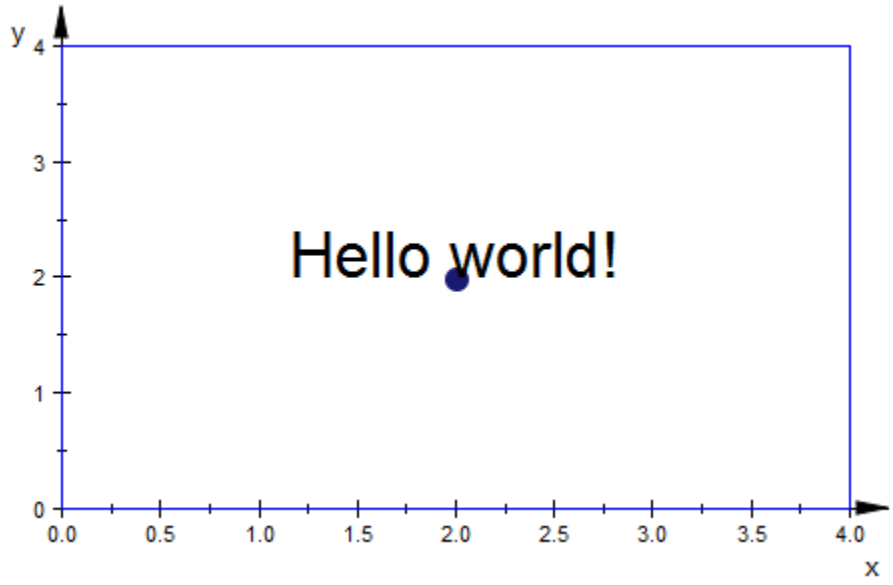
`TextRotation` can be animated.

Examples

Example 1

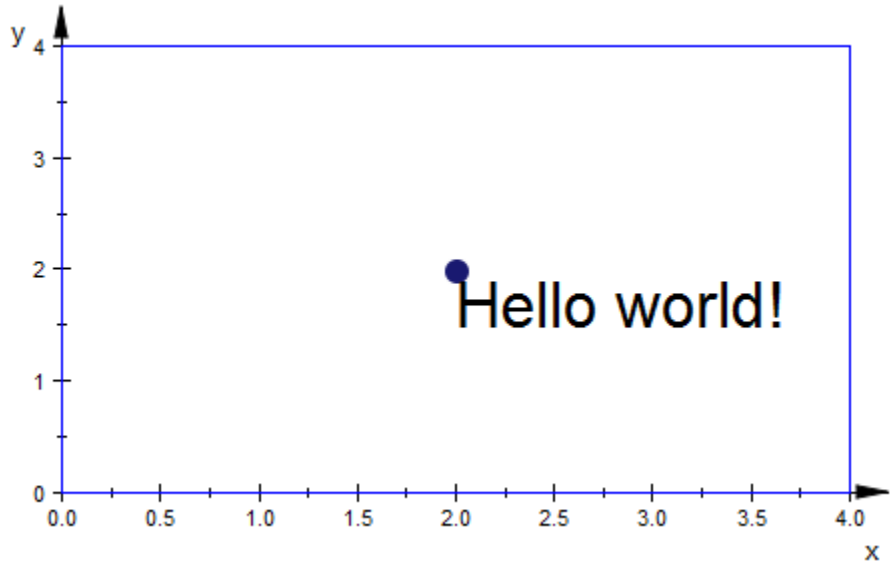
We draw a rectangle and a rotating text inside:

```
r := plot::Rectangle(0..4, 0..4):
p := plot::Point2d([2, 2], PointSize = 3*unit::mm):
text := plot::Text2d("Hello world!", [2, 2],
                     HorizontalAlignment = Center,
                     TextRotation = a, a = 0..2*PI,
                     TextFont = [24]):
plot(r, p, text):
```

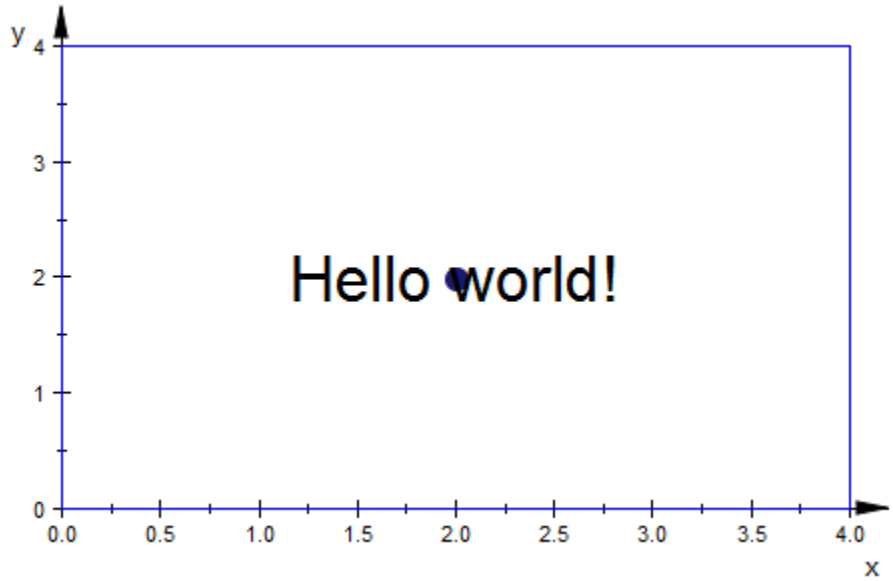


We change the alignment of the text w.r.t. its anchor point [2, 2]:

```
text:: HorizontalAlignment := Left:
text:: VerticalAlignment := Top:
plot(r, p, text):
```




```
text:: HorizontalAlignment := Center:
text:: VerticalAlignment := Center:
plot(r, p, text):
```




```
delete r, p, text:
```

See Also

MuPAD Functions
`TextOrientation`

UName, URange, UMin, UMax, VName, VRange, VMin, VMax, XName, XRange, XMin, XMax, YName, YRange, YMin, YMax, ZName, ZRange, ZMin, ZMax

Names and values ranges of parameters

Value Summary

`UMax`, `UMin`, `UName`, `VMax`, `VMin`, `VName`, `XMax`, `XMin`, `XName`, `YMax`, `YMin`, `YName`, `ZMax`, `ZMin`, `ZName`	Mandatory	MuPAD expression
`URange`	[`UMin` .. `UMax`]	Range of arithmetical expressions
`VRange`	[`VMin` .. `VMax`]	Range of arithmetical expressions
`XRange`	[`XMin` .. `XMax`]	Range of arithmetical expressions
`YRange`	[`YMin` .. `YMax`]	Range of arithmetical expressions
`ZRange`	[`ZMin` .. `ZMax`]	Range of arithmetical expressions

Graphics Primitives

Objects	Default Values

`plot::Curve2d`, `plot::Curve3d`, `plot::Function2d`, `plot::XRotate`	`URange`, `XRange`: `-5 .. 5`
	
	`UMin`, `XMin`: `-5`
	
	`UMax`, `XMax`: `5`

Objects	Default Values
plot::Bars3d, plot::ClippingBox, plot::Conformal, plot::Cylindrical, plot::Density, plot::Implicit2d, plot::Implicit3d, plot::Inequality, plot::Iteration, plot::Listplot, plot::Matrixplot, plot::Polar, plot::Raster, plot::Rootlocus, plot::Sequence, plot::SparseMatrixplot, plot::Spherical, plot::Streamlines2d, plot::Sum, plot::Surface, plot::Sweep, plot::Tube, plot::VectorField2d, plot::VectorField3d	
plot::ZRotate	XRange: 0 .. 5 XMin: 0 XMax: 5
plot::Function3d	XRange, YRange: -5 .. 5 XMin, YMin: -5 XMax, YMax: 5
plot::Box	XRange, YRange, ZRange: -1 .. 1 XMin, YMin, ZMin: -1 XMax, YMax, ZMax: 1
plot::Rectangle	XRange, YRange: -1 .. 1 XMin, YMin: -1 XMax, YMax: 1

Objects	Default Values
plot::Hatch	XRange: -infinity .. infinity XMin: -infinity XMax: infinity

Description

UName, VName, XName, YName, ZName specify the names of parameters defining parametrized objects such as functions, curves and surfaces.

UMin, UMax, VMin, VMax, XMin, XMax, YMin, YMax, ZMin, ZMax specify the minimal and maximal values of the range of the parameters.

URange, VRange, XRange, YRange, ZRange serve as shortcuts for setting UMin, UMax etc.

In most cases, the user has no need for using these attributes explicitly, because parameter ranges are set implicitly during creation of plot objects. For example, the definition

```
f := plot::Function2d(sin(x), x = 0 .. 2*PI)
```

sets the attribute values XName = x, XMin = 0, XMax = 2*PI automatically for the function object f. In fact, you can define f by the equivalent call

```
f := plot::Function2d(sin(x), XName = x, XMin = 0, XMax = 2*PI).
```

In the interactive object browser of the MuPAD graphics tool (see section Viewer, Browser, and Inspector: Interactive Manipulation of this document), you will not see a specification such as x = 0 .. 2*PI, but you find separate entries for XName, XMin, XMax.

The use of 'X', 'Y', 'Z' as opposed to 'U', 'V' depends on the type of the object.

Functions of type plot::Function2d refer to the independent variable (“the parameter”) as XName.

Functions of type plot::Function3d refer to the two independent variables as XName, YName.

Parametrized curve of type `plot::Curve2d` or `plot::Curve3d` refer to the curve parameter as `UName`.

Parametrized surfaces of type `plot::Surface`, `plot::XRotate` etc. refer to the two surface parameters as `UName`, `VName`.

Various other plot objects of type `plot::Implicit2d`, `plot::VectorField2d` etc. also use the attributes `XName` etc. Here, the ranges from `XMin` to `XMax` etc. denote the coordinate range in which the objects are placed.

After a definition such as `f := plot::Function2d(sin(x), x = 0 .. 2*PI)`, the parameter and its range can be accessed as the slots `f::XName`, `f::XMin`, `f::XMax`.

The slot `f::XRange` yields the range `0 .. 2*PI` consisting of the values of `XMin` and `XMax`. Setting the attribute `XRange` is a short cut for setting `XMin` and `XMax` simultaneously. For example, setting `f::XRange := -PI .. PI` is equivalent to setting `f::XMin := -PI` and `f::XMax := PI`.

Of course, the analogous statements hold for `YRange`, `ZRange`, `URange`, `VRange`, too.

Examples

Example 1

We define a function object:

```
f1 := plot::Function2d(sin(x), x = 0.. 2*PI)
```

plot::Function2d(sin(x), x = 0..2 π)

This is equivalent to:

```
f2 := plot::Function2d(sin(x), XName = x, XMin = 0, XMax = 2*PI)
```

plot::Function2d(sin(x), x = 0..2 π)

The objects `f1` and `f2` have the same entries for the parameter `x`:

```
f1::XName = f1::XMin .. f1::XMax, f2::XName = f2::XRange
```

$$x = 0..2\,\pi,\ x = 0..2\,\pi$$

Changing the x range via `XRange` is equivalent to changing `XMin` and `XMax` separately:

```
f1::XRange := -PI..PI:
f2::XMin := -PI:
f2::XMax := PI:
f1, f2
```

$$\text{plot::Function2d}(\sin(x),\ x = -\pi..\pi),\ \text{plot::Function2d}(\sin(x),\ x = -\pi..\pi)$$

```
delete f1, f2:
```

See Also

MuPAD Functions

`AngleBegin` | `AngleEnd` | `AngleRange` | `ParameterBegin` | `ParameterEnd` | `ParameterName` | `ParameterRange` | `TimeBegin` | `TimeEnd` | `TimeRange`

ViewingBox, ViewingBoxXMin, ViewingBoxXMax, ViewingBoxXRange, ViewingBoxYMin, ViewingBoxYMax, ViewingBoxYRange, ViewingBoxZMin, ViewingBoxZMax, ViewingBoxZRange

Visible coordinate range

Value Summary

`ViewingBox`	Library wrapper for "[`ViewingBoxXMin` .. `ViewingBoxXMax`, `ViewingBoxYMin` .. `ViewingBoxYMax`]" (2D), "[`ViewingBoxXMin` .. `ViewingBoxXMax`, `ViewingBoxYMin` .. `ViewingBoxYMax`, `ViewingBoxZMin` .. `ViewingBoxZMax`]" (3D)	See below
`ViewingBoxXMax`, `ViewingBoxXMin`, `ViewingBoxYMax`, `ViewingBoxYMin`, `ViewingBoxZMax`, `ViewingBoxZMin`	Optional	MuPAD expression
`ViewingBoxXRange`	[`ViewingBoxXMin` .. `ViewingBoxXMax`]	See below
`ViewingBoxYRange`	[`ViewingBoxYMin` .. `ViewingBoxYMax`]	See below
`ViewingBoxZRange`	[`ViewingBoxZMin` .. `ViewingBoxZMax`]	See below

Graphics Primitives

Objects	Default Values

`plot::CoordinateSystem2d`	`ViewingBox`: [`Automatic` .. `Automatic`, `Automatic` .. `Automatic`]
	
	`ViewingBoxXMin`, `ViewingBoxXMax`, `ViewingBoxYMin`, `ViewingBoxYMax`: `Automatic`
	
	`ViewingBoxXRange`, `ViewingBoxYRange`: `Automatic` .. `Automatic`

`plot::CoordinateSystem3d`	`ViewingBox`: [`Automatic` .. `Automatic`, `Automatic` .. `Automatic`, `Automatic` .. `Automatic`]
	
	`ViewingBoxXMin`, `ViewingBoxXMax`, `ViewingBoxYMin`, `ViewingBoxYMax`, `ViewingBoxZMin`, `ViewingBoxZMax`: `Automatic`
	
	`ViewingBoxXRange`, `ViewingBoxYRange`, `ViewingBoxZRange`: `Automatic` .. `Automatic`

Description

The `ViewingBox` attributes set the coordinate range that is visible in a plot.

`ViewingBoxXMin` = x_{min}, `ViewingBoxXMax` = x_{max}, equivalent to `ViewingBoxXRange` = x_{min} `..` x_{max}, restricts the visibility to x values between x_{min} and x_{max}. `ViewingBoxYMin` etc. work analogously.

Setting `ViewingBox` = [x_{min} `..` x_{max}, y_{min} `..` y_{max}] in 2D and `ViewingBox` = [x_{min} `..` x_{max}, y_{min} `..` y_{max}, z_{min} `..` z_{max}] in 3D

respectively, serves as a short cut for setting the single entries `ViewingBoxXMin` etc.

The ViewingBox of a plot is computed automatically by default. It is chosen as the smallest box containing all graphical objects in the coordinate system.

The values x_{min} etc. of the ViewingBox attributes must be real numerical expressions or the special flag Automatic. With Automatic, the system chooses appropriate values automatically.

When plotting a function or a curve with singularities, a heuristics is used to set a "reasonable" restricted viewing box for the plot. This heuristics sometimes fails to produce a pleasing picture. We recommended to request an explicit ViewingBox in such a case.

When using plot::Rotate2d or plot::Rotate3d, the ViewingBox may be larger than necessary. Its size is computed by rotating the common viewing box of all objects in the rotation object. See "Example 4" on page 24-1522.

The ViewingBox of an animation is automatically chosen as the union of all viewing boxes of the frames of the animation.

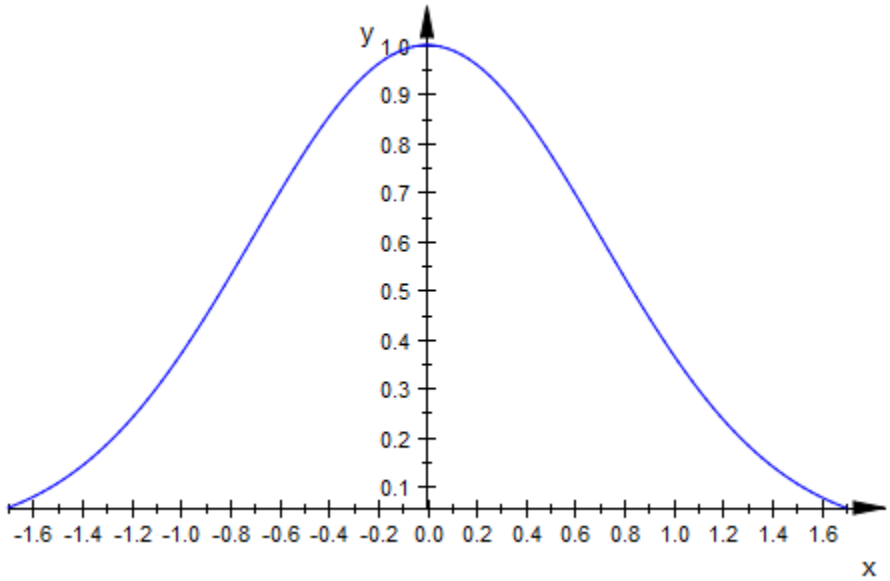
The ViewingBox itself cannot be animated. However, the object plot::ClippingBox may be used to implement animated visibility regions.

Examples

Example 1

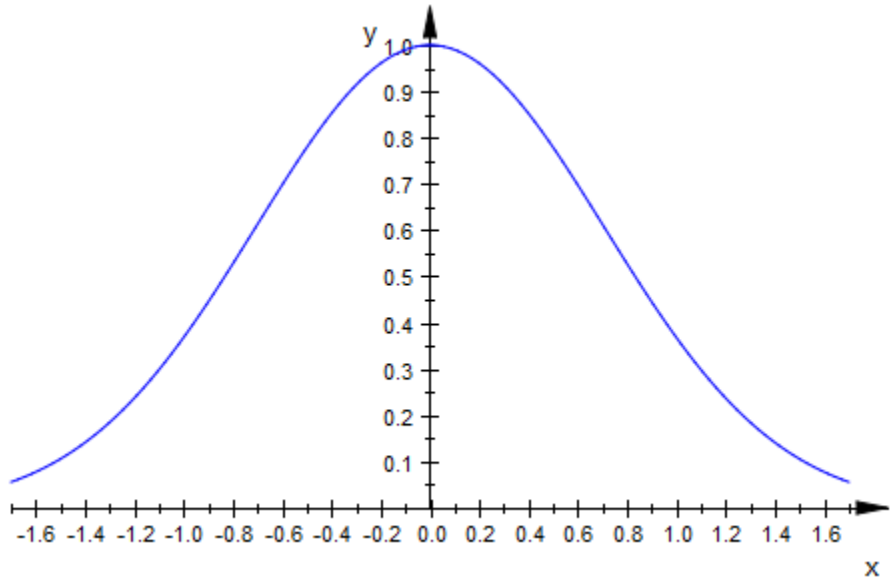
In the following plot, the horizontal axis is placed at the minmal *y*-value produced by the function:

```
f := plot::Function2d(exp(-x^2), x = -1.7 .. 1.7):
plot(f)
```



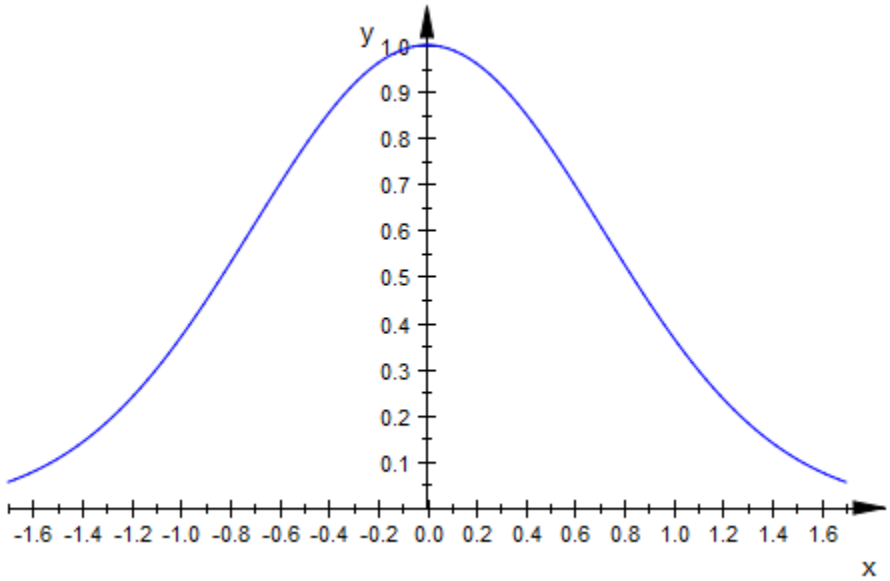
We wish to make the x-axis appear at $y = 0$. To this end, we request the y range to start with $y = 0$ and use `Automatic` to let MuPAD find the maximal y-value automatically:

```
plot(f, ViewingBoxYRange = 0..Automatic)
```



The previous command is equivalent to:

```
plot(f, ViewingBoxYMin = 0)
```

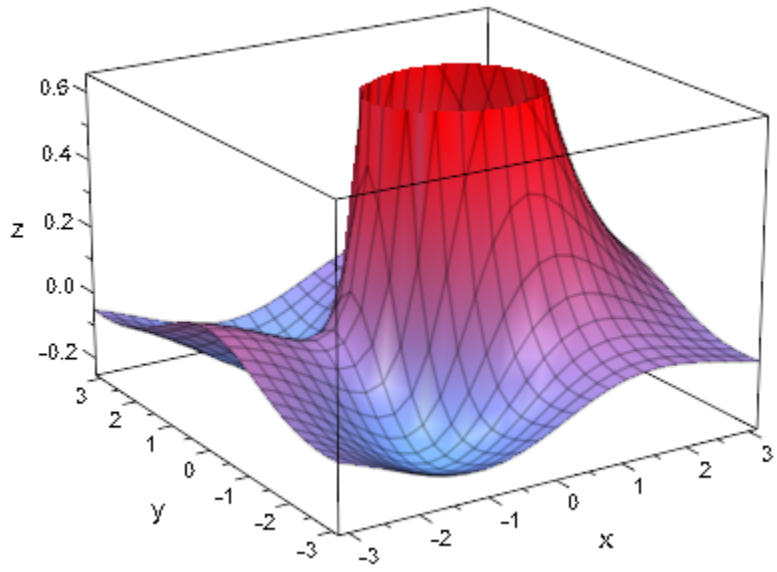



```
delete f:
```

Example 2

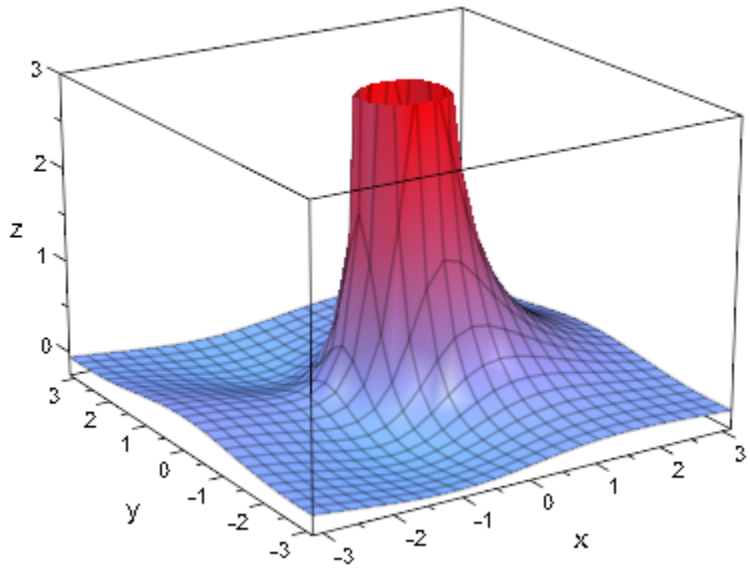
Here is a 3D plot of a singular function:

```
f := plot::Function3d((sin(x) + cos(y))/(x^2 + y^2),
                      x = -PI..PI, y = -PI..PI):
plot(f)
```



We specify the upper z value of the visible volume:

```
plot(f, ViewingBoxZRange = Automatic..3)
```

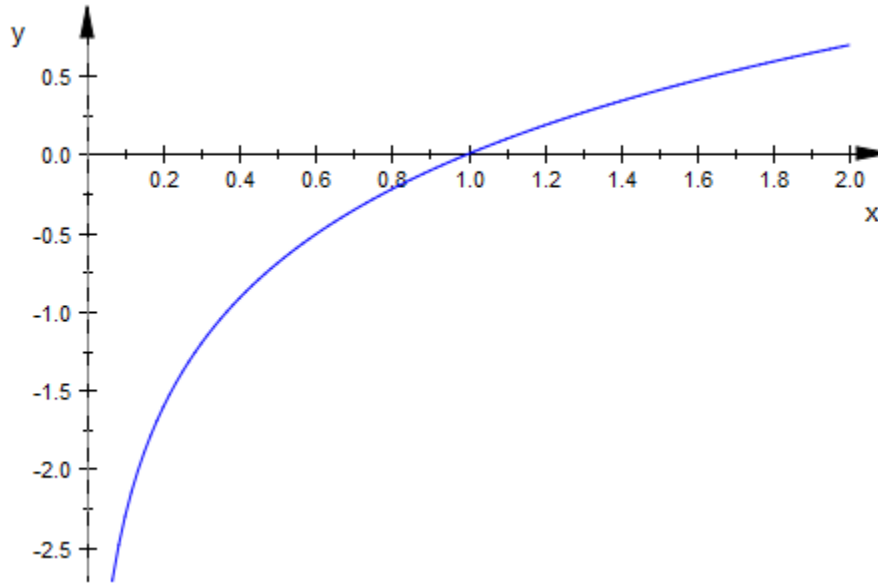



```
delete f:
```

Example 3

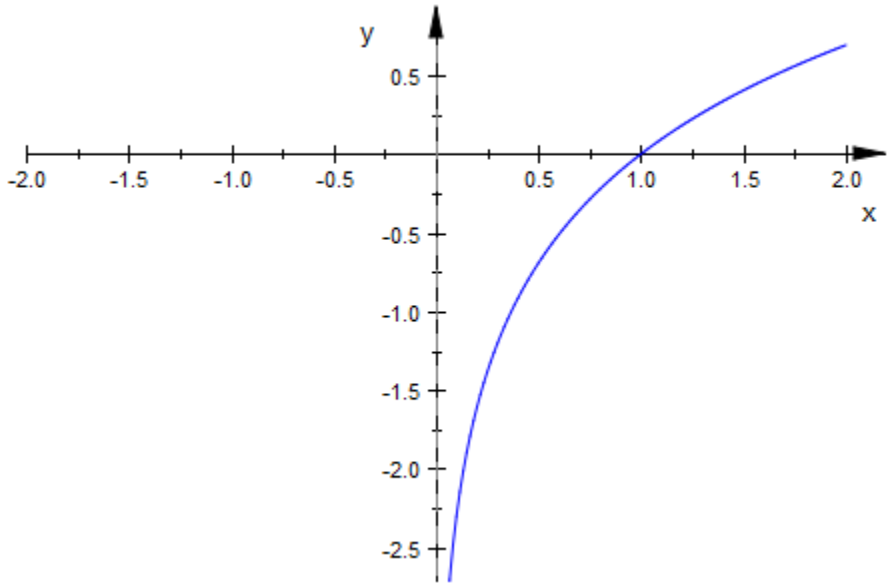
Usually, a plot uses the whole drawing aera:

```
f := plot::Function2d(ln(x), x = 0..2):
plot(f)
```



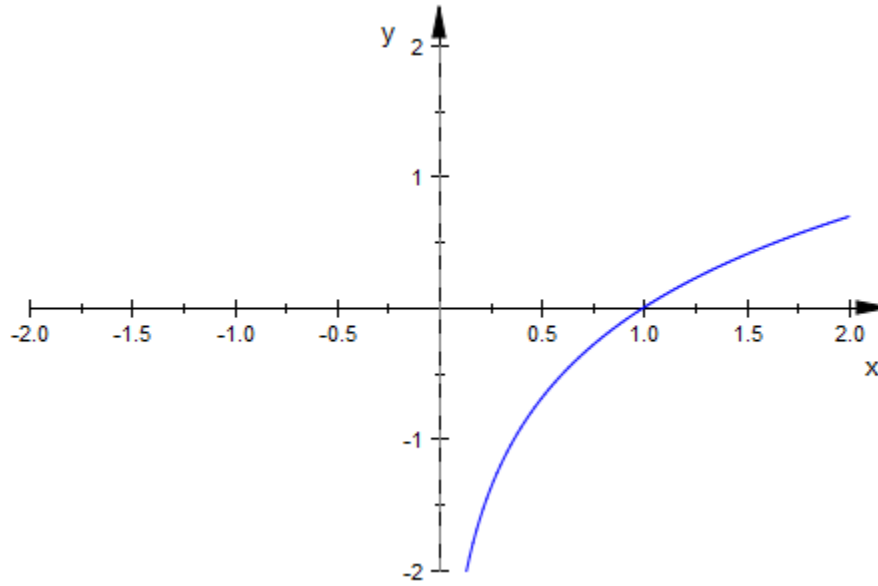
We extend the viewing box in x direction to make it symmetric w.r.t. x:

```
plot(f, ViewingBoxXRange = -2..2)
```



We specify the viewing box both in x and y direction:

```
plot(f, ViewingBox = [-2..2, -2..2])
```

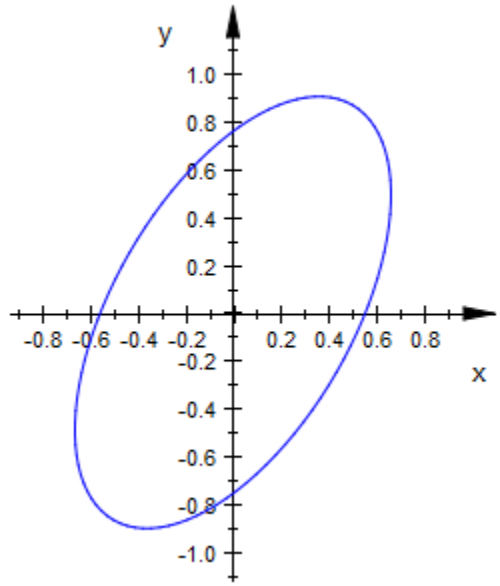



```
delete f:
```

Example 4

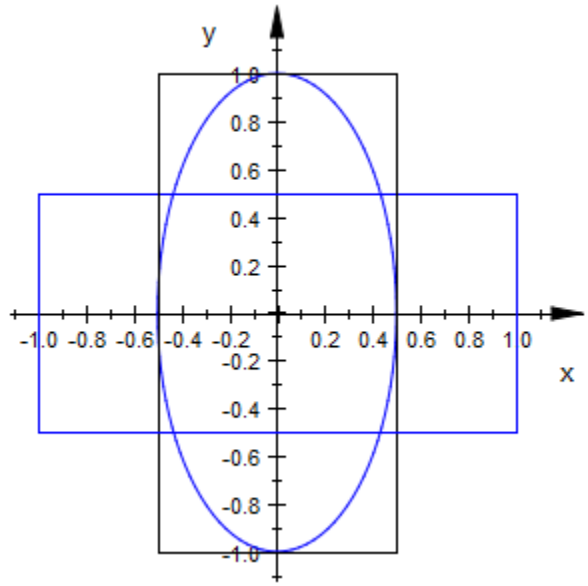
The following viewing box is larger than expected:

```
c := plot::Ellipse2d(1, 0.5, [0, 0]):
r := plot::Rotate2d(c, PI/3):
plot(r)
```



The reason is how the viewing box of the rotation is computed. The blue rectangle is the viewing box of the ellipse. The rotated viewing box is the black rectangle. The viewing box of the rotation is the smallest rectangle containing the rotated viewing box of the ellipse (the dashed black rectangle):

```
rect1 := plot::Rectangle(-1..1, -0.5..0.5, Color = RGB::Black):
rect2 := plot::modify(rect1, Color = RGB::Blue):
r := plot::Rotate2d(c, rect1, a, a = 0..PI/2):
X := cos(a) + 0.5*sin(a):
Y := 0.5*cos(a) + sin(a):
rect3 := plot::Rectangle(-X..X, -Y..Y, a = 0..PI/2,
                         Color = RGB::Black,
                         LineStyle = Dashed):
plot(r, rect2, rect3)
```

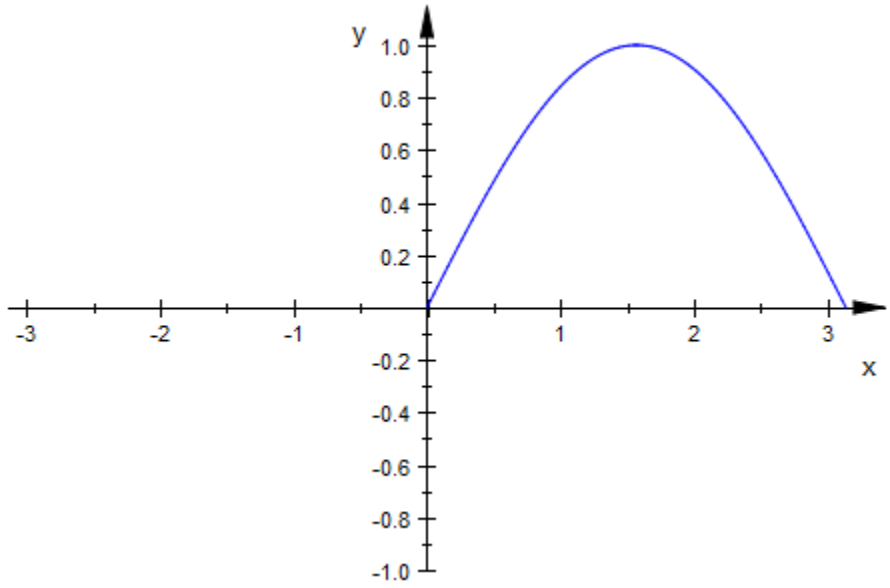



```
delete c, r, rect1, rect2, rect3, X, Y:
```

Example 5

The *x*-range from - π to π is generated by *all* frames of the following animation and does not change from frame to frame:

```
plot(plot::Function2d(sin(x), x = -PI + a .. a, a = 0 .. PI)):
```



See Also

MuPAD Functions
`AffectViewingBox`

MuPAD Graphical Primitives
`plot::ClippingBox`

Visible

Visibility

Value Summary

Optional	FALSE, or TRUE

Graphics Primitives

Objects	Visible Default Values
plot::AmbientLight, plot::Arc2d, plot::Arc3d, plot::Arrow2d, plot::Arrow3d, plot::Bars2d, plot::Bars3d, plot::Box, plot::Boxplot, plot::Camera, plot::Circle2d, plot::Circle3d, plot::ClippingBox, plot::Cone, plot::Conformal, plot::Curve2d, plot::Curve3d, plot::Cylinder, plot::Cylindrical, plot::Density, plot::DistantLight, plot::Dodecahedron, plot::Ellipse2d, plot::Ellipse3d, plot::Ellipsoid, plot::Function2d, plot::Function3d, plot::Group2d, plot::Group3d, plot::Hatch, plot::Hexahedron, plot::Histogram2d, plot::Icosahedron, plot::Implicit2d, plot::Implicit3d, plot::Inequality, plot::Integral, plot::Iteration, plot::Line2d, plot::Line3d, plot::Listplot, plot::Lsys, plot::Matrixplot, plot::MuPADCube, plot::Octahedron,	TRUE

Objects	Visible Default Values
plot::Ode2d, plot::Ode3d, plot::Parallelogram2d, plot::Parallelogram3d, plot::Piechart2d, plot::Piechart3d, plot::Plane, plot::Point2d, plot::Point3d, plot::PointLight, plot::PointList2d, plot::PointList3d, plot::Polar, plot::Polygon2d, plot::Polygon3d, plot::Prism, plot::Pyramid, plot::QQplot, plot::Raster, plot::Rectangle, plot::Rootlocus, plot::Scatterplot, plot::Sequence, plot::SparseMatrixplot, plot::Sphere, plot::Spherical, plot::SpotLight, plot::Streamlines2d, plot::Sum, plot::Surface, plot::SurfaceSet, plot::SurfaceSTL, plot::Sweep, plot::Tetrahedron, plot::Text2d, plot::Text3d, plot::Tube, plot::Turtle, plot::VectorField2d, plot::VectorField3d, plot::Waterman, plot::XRotate, plot::ZRotate	

Description

`Visible = FALSE` makes an object invisible.

All graphical objects react to the attribute `Visible`. With `Visible = FALSE`, an object is made invisible. This attribute can be set in the property inspector of the interactive viewer (see section Viewer, Browser, and Inspector: Interactive Manipulation in this document) to make a selected object disappear without needing to change and re-execute the plot call.

Invisible objects do influence the viewing box of their coordinate systems.

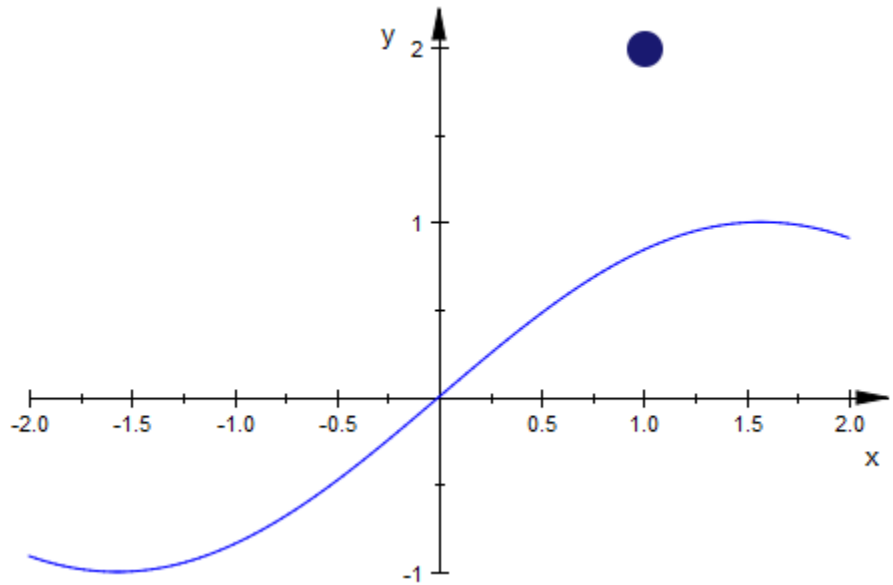
`Visible` cannot be animated. However, the attributes `VisibleBefore`, `VisibleBeforeBegin`, `VisibleAfter`, and `VisibleAfterEnd` serve for some form of animated visibility. See section Frame by Frame Animations in this document for further details and examples.

Examples

Example 1

Consider the following scene:

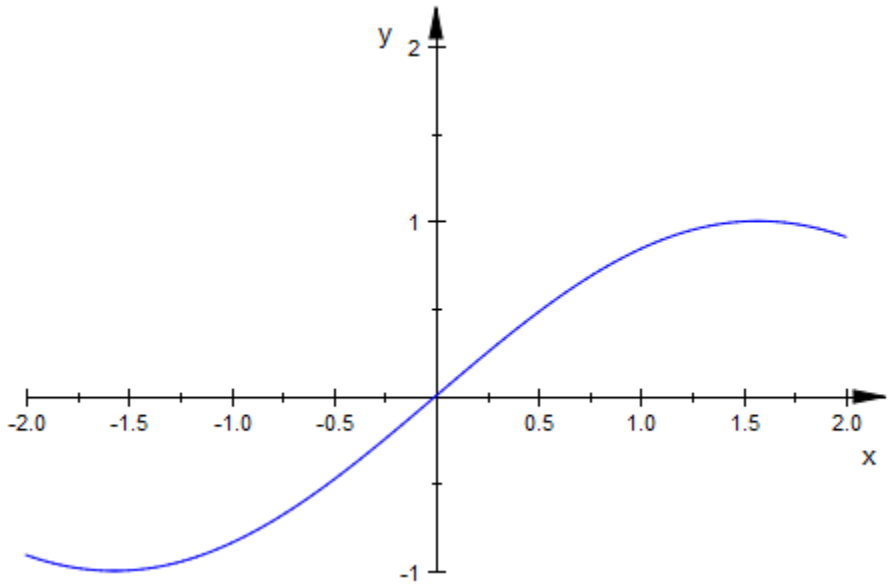
```
plot(plot::Function2d(sin(x), x = -2..2),
     plot::Point2d([1, 2], PointSize = 5*unit::mm))
```



Obviously, the point influences the visible region of the coordinates. This region is not affected by making the point invisible:

```
plot(plot::Function2d(sin(x), x=-2..2),
     plot::Point2d([1, 2], PointSize = 5*unit::mm,
```

```
Visible = FALSE))
```



See Also

MuPAD Functions

`VisibleAfter` | `VisibleAfterEnd` | `VisibleBefore` | `VisibleBeforeBegin`

XFunction1, YFunction1, ZFunction1, XFunction2, YFunction2, ZFunction2

Parametrization of the curves in sweep surfaces

Value Summary

`XFunction1`, `XFunction2`, `YFunction1`, `YFunction2`, `ZFunction1`, `ZFunction2`	Mandatory	Arithmetical expression or function

Graphics Primitives

Objects	Default Values
`plot::Sweep`	

Description

`XFunction1` etc. are the parametrization functions of the curves delimiting a surface of type `plot::Sweep`.

In most cases, the user passes parametrizations $[x_1(u), y_1(u), z_1(u)]$ and $[x_2(u), y_2(u), z_2(u)]$ as expressions of a curve parameter u directly to `plot::Sweep`. Internally, these expressions are stored as the attributes `XFunction1` $= x_1(u)$, ..., `ZFunction2` $= z_2(u)$ in the sweep object. They can be accessed and changed via the corresponding slots `"XFunction1"` etc. of the sweep object.

The attributes `XFunction1` etc. can also be defined by procedures instead of symbolic expressions.

Examples

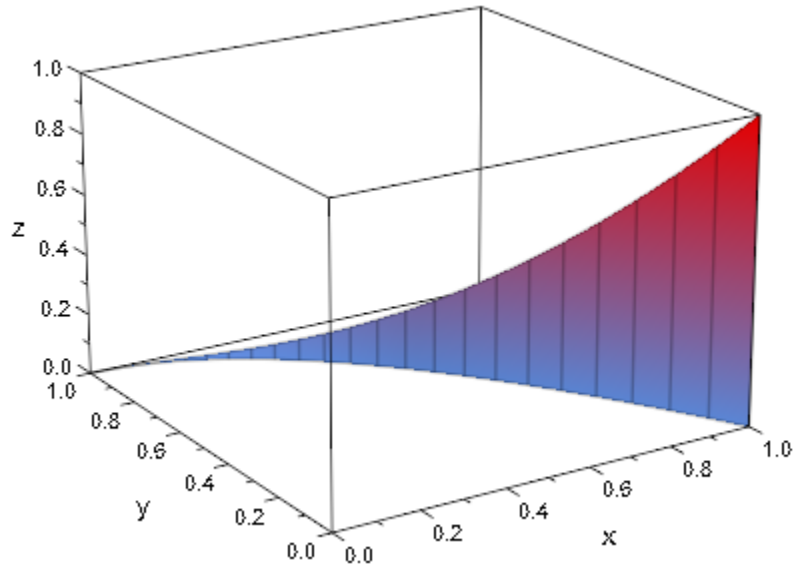
Example 1

Typically, the user sets the parametrization of the bounding curves directly by passing lists of corresponding expressions to `plot::Sweep`. Here, `XFunction1 = u`, `YFunction1 = 1 - u^2`, `ZFunction1 = u^3`, `XFunction2 = u`, `YFunction2 = 1 - u^2`, `ZFunction2 = 0`:

```
s := plot::Sweep([u, 1 - u^2, u^3], [u, 1 - u^2, 0], u = 0..1)
```

$$\text{plot::Sweep}([u,\ 1-u^2,\ u^3],\ [u,\ 1-u^2,\ 0],\ u=0..1)$$

```
plot(s):
```




```
s::XFunction1, s::YFunction1, s::ZFunction1
```

$$u,\ 1-u^2,\ u^3$$

```
s::XFunction2, s::YFunction2, s::ZFunction2
```

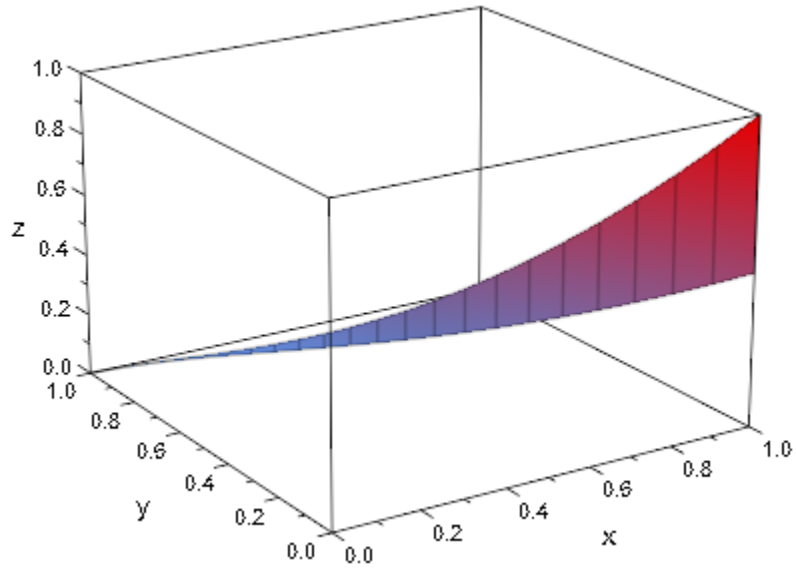
$$u,\ 1-u^2,\ 0$$

We change the z-component of the "target curve":

```
s::ZFunction2 := s::ZFunction1 / 2:
s
```

$$\text{plot::Sweep}\left([u,\ 1-u^2,\ u^3],\ \left[u,\ 1-u^2,\ \frac{u^3}{2}\right],\ u=0..1\right)$$

```
plot(s)
```

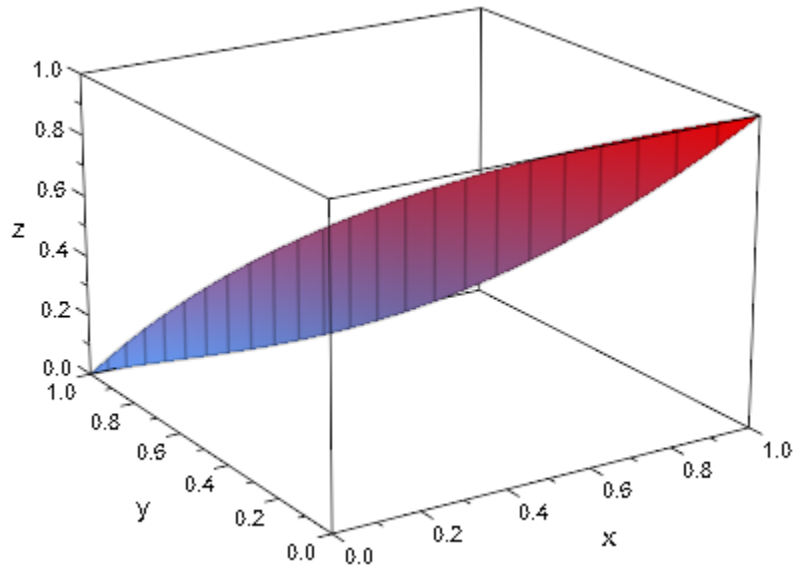


Instead of expressions, the attributes `XFunction1` etc. can be defined by procedures:

```
s::ZFunction2 :=  u -> u:
s
```

$$\text{plot::Sweep}([u,\ 1-u^2,\ u^3],\ [u,\ 1-u^2,\ u \to u],\ u=0..1)$$

```
plot(s)
```




```
delete s:
```

Axes

Type of the coordinate axes

Value Summary

Inherited	`Automatic`, `Boxed`, `Frame`, `None`, or `Origin`

Graphics Primitives

Objects	Axes Default Values
`plot::CoordinateSystem2d`	`Automatic`
`plot::CoordinateSystem3d`	`Boxed`

Description

`Axes` determines the type of the coordinate axes.

The following types of coordinate axes are available:

- `Automatic`: The axes are displayed as a coordinate cross. The cross point is not taken from the attribute `AxesOrigin`, but is chosen automatically.
- `Origin`: The axes are displayed as a coordinate cross. The cross point is set by the attribute `AxesOrigin`. If `AxesOrigin` is not set, the origin of the coordinate system is used as the default cross point. If the `AxesOrigin` is not inside the "viewing box" of the scene, parts of the axes may not be visible (cf. "Example 4" on page 24-1540).
- `Boxed`: The axes are displayed as a box around the graphical scene. It corresponds to the "viewing box" of the scene and may be set explicitly by the attribute `ViewingBox`.
- `Frame`: As with `Axes = Boxed`, the edges of the "viewing box" are used. However, only the labeled edges are displayed.
- `None`: No coordinate axes are displayed.

As an alternative to `Axes = None`, you may also "switch the axes off" by setting `AxesVisible` = `FALSE` in the `plot` command or via the interactive object inspector (see Viewer, Browser, and Inspector: Interactive Manipulation in this document).

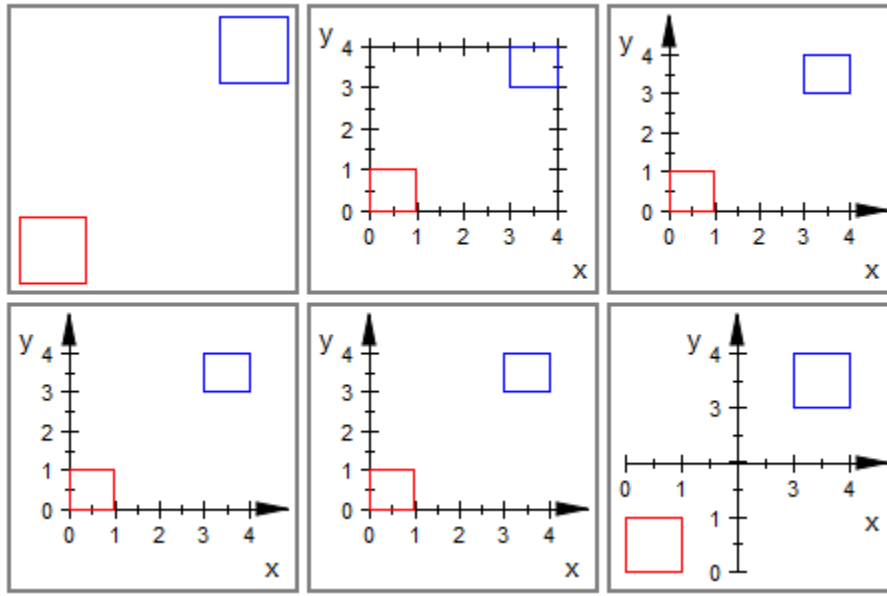
Single coordinate axes can also be "switched off" separately via `XAxisVisible` = `FALSE` etc.

Examples

Example 1

We demonstrate the axes styles in 2D:

```
b1 := plot::Rectangle(0..1, 0..1, Color = RGB::Red):
b2 := plot::Rectangle(3..4, 3..4, Color = RGB::Blue):
plot(plot::Scene2d(b1, b2, Axes = None),
     plot::Scene2d(b1, b2, Axes = Boxed),
     plot::Scene2d(b1, b2, Axes = Frame),
     plot::Scene2d(b1, b2, Axes = Automatic),
     plot::Scene2d(b1, b2, Axes = Origin),
     plot::Scene2d(b1, b2, Axes = Origin,
                   AxesOrigin = [2, 2]),
     plot::Scene2d::BorderWidth = 0.5*unit::mm,
     Rows = 2):
```

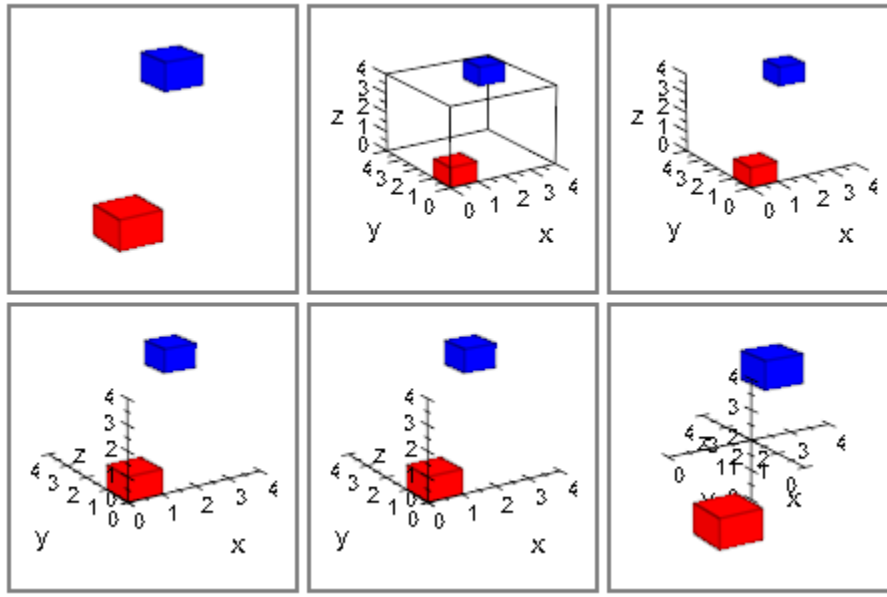



```
delete b1, b2:
```

Example 2

We demonstrate the axes styles in 3D:

```
b1 := plot::Box(0..1, 0..1, 0..1, Color = RGB::Red):
b2 := plot::Box(3..4, 3..4, 3..4, Color = RGB::Blue):
plot(plot::Scene3d(b1, b2, Axes = None),
     plot::Scene3d(b1, b2, Axes = Boxed),
     plot::Scene3d(b1, b2, Axes = Frame),
     plot::Scene3d(b1, b2, Axes = Automatic),
     plot::Scene3d(b1, b2, Axes = Origin),
     plot::Scene3d(b1, b2, Axes = Origin,
                   AxesOrigin = [2, 2, 2]),
     plot::Scene3d::BorderWidth = 0.5*unit::mm,
     Rows = 2):
```

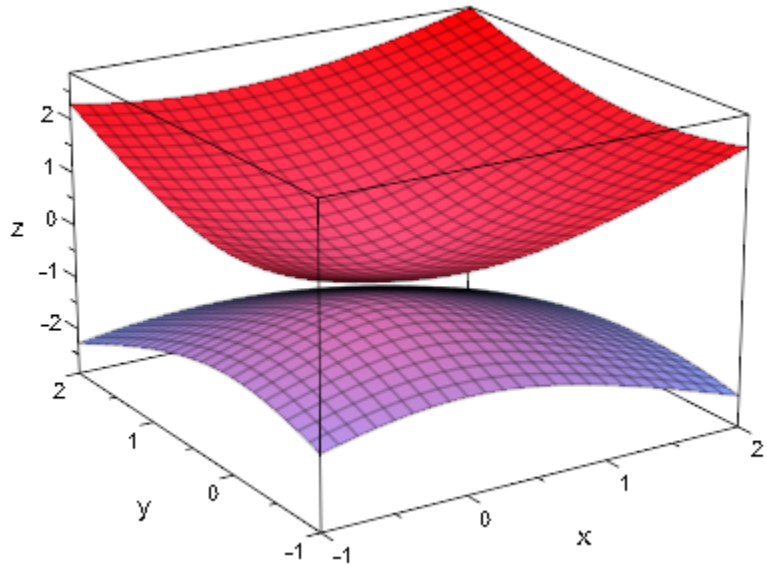


```
delete b1, b2:
```

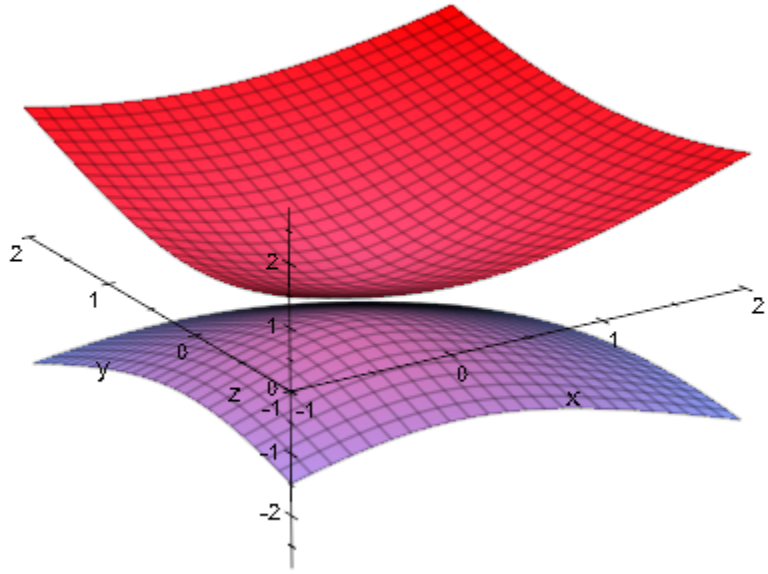
Example 3

Here is a hyperboloid with various axes:

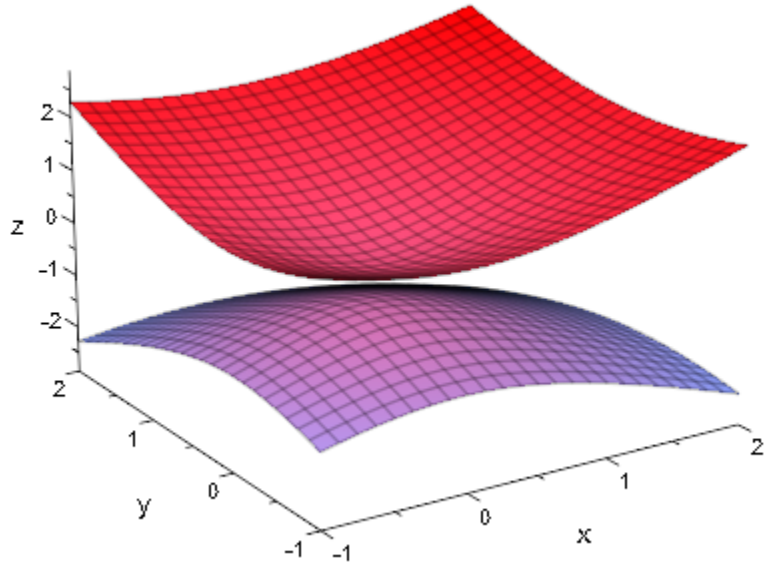
```
f1 := plot::Function3d(sqrt(0.2 + x^2 + y^2),
                       x = -1..2, y = -1..2):
f2 := plot::Function3d(-sqrt(0.2 + x^2 + y^2),
                       x = -1..2, y = -1..2):
plot(f1, f2):
```




```
plot(f1, f2, Axes = Origin,
     AxesOrigin = [-1, -1, 0]):
```




```
plot(f1, f2, Axes = Frame):
```

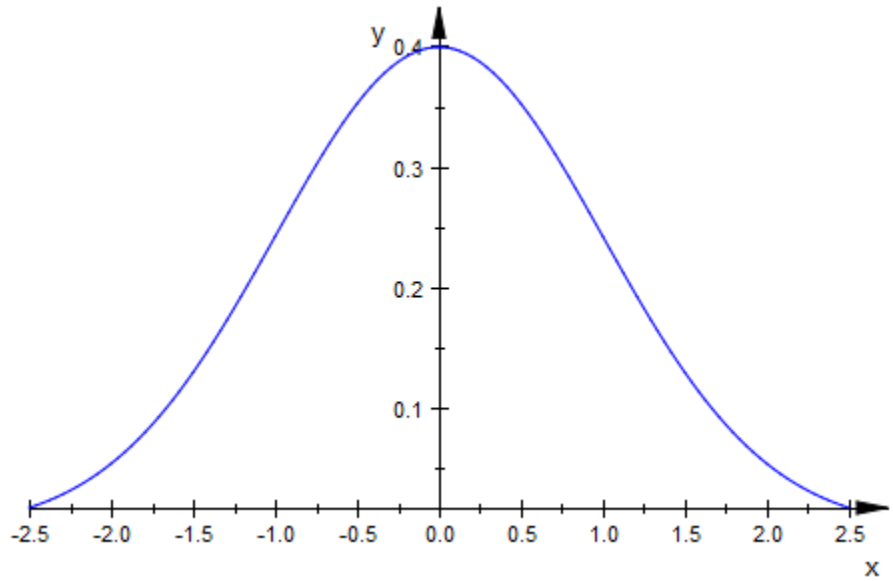



```
delete f1, f2:
```

Example 4

We draw a portion of the normal distribution density:

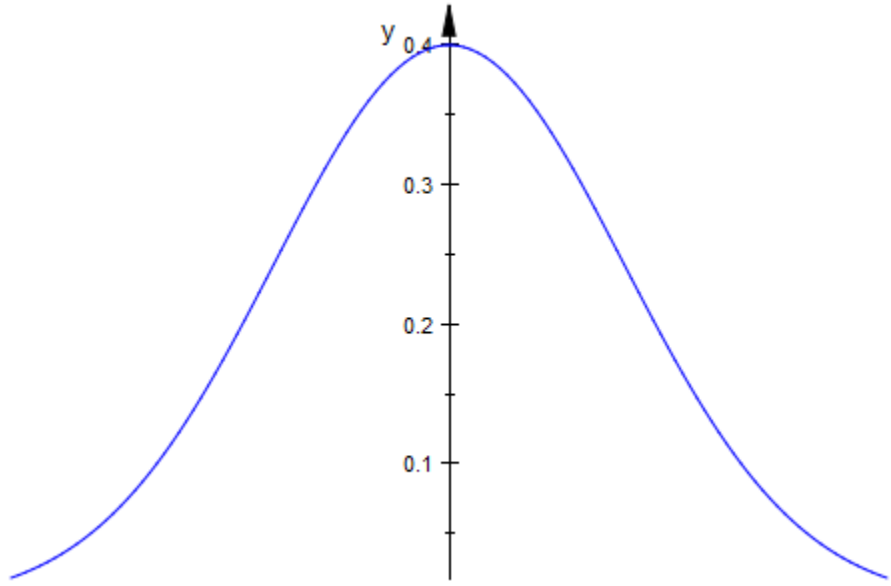
```
F := plot::Function2d(stats::normalPDF(0, 1)(x),
                      x = -2.5 .. 2.5):
plot(F)
```



Note that with the default setting `Axes = Automatic`, the *x*-axis does not pass through the origin but is shifted along the *y*-axis to fit into the viewing box of the scene.

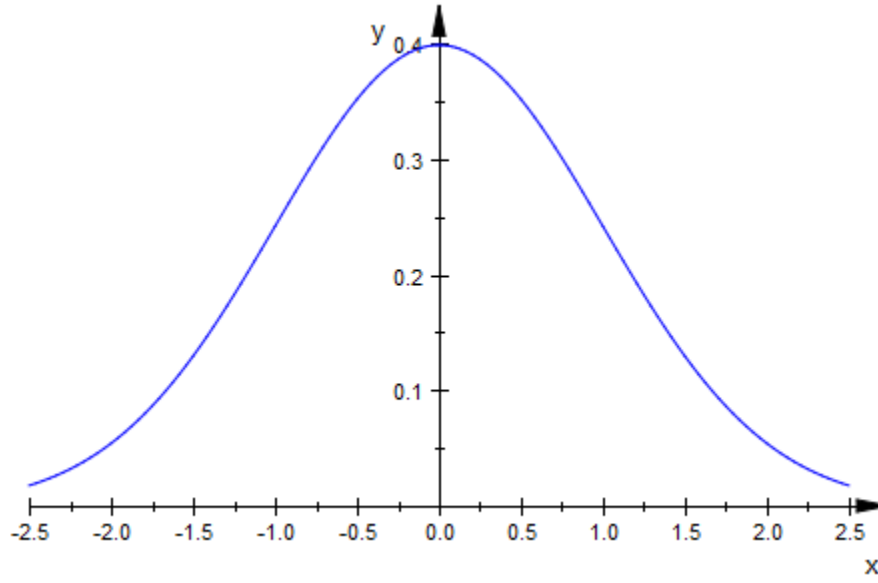
With `Axes = Origin`, the *x*-axis passes through the origin, but is outside the viewing box:

```
plot(F, Axes = Origin)
```



We extend the viewing box in the y direction:

```
plot(F, Axes = Origin, ViewingBoxYRange = 0 .. Automatic):
```




```
delete F:
```

See Also

MuPAD Functions

AxesInFront | AxesLineColor | AxesLineWidth | AxesOrigin | AxesTips | AxesTitleAlignment | AxesTitleFont | AxesTitles | AxesVisible | YAxisTitleOrientation

AxesInFront

Coordinate axes in front of or behind graphical objects?

Value Summary

Inherited	FALSE, or TRUE

Graphics Primitives

Objects	AxesInFront Default Values
`plot::CoordinateSystem2d`	FALSE

Description

`AxesInFront` = `TRUE` versus `AxesInFront` = `FALSE` places 2D axes in front of or behind the graphical objects in the scene.

By default, the coordinate axes are plotted behind the graphical objects in a scene. Consequently, the objects may cover the axes. If only line objects and points are present in a 2D scene, this is desirable in most cases.

However, if there are filled areas such as filled polygons in the scene, the view to the axes, tick marks, and tick labels may be totally blocked. In such a situation, you may want to draw the axes in front of the objects to guarantee visibility of the axes.

Although the default setting is `AxesInFront` = `FALSE`, some objects which create filled areas send `AxesInFront` = `TRUE` as a "hint" (see the section Primitives Requesting Special Scene Attributes: "Hints" of this documentation).

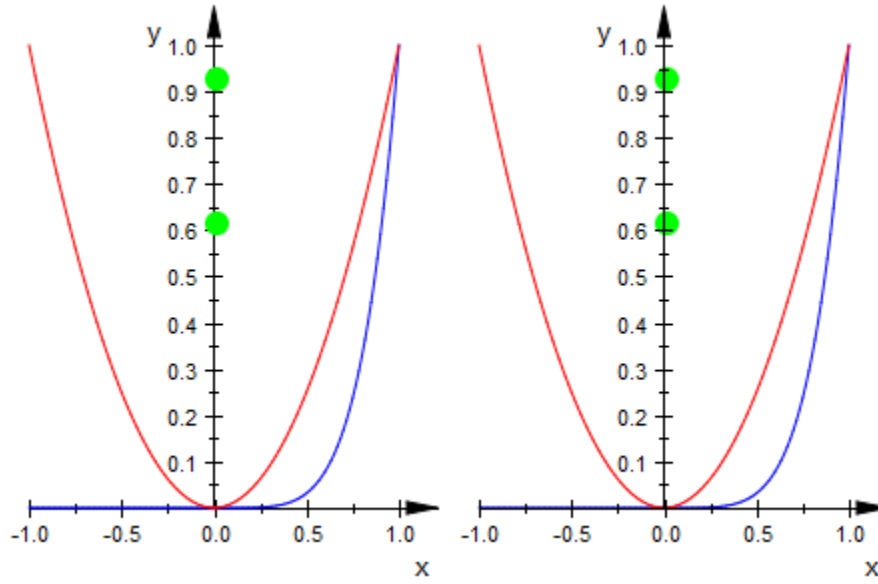
This attribute is available only in 2D.

Examples

Example 1

It is usually desirable to let line objects and points cover the axes:

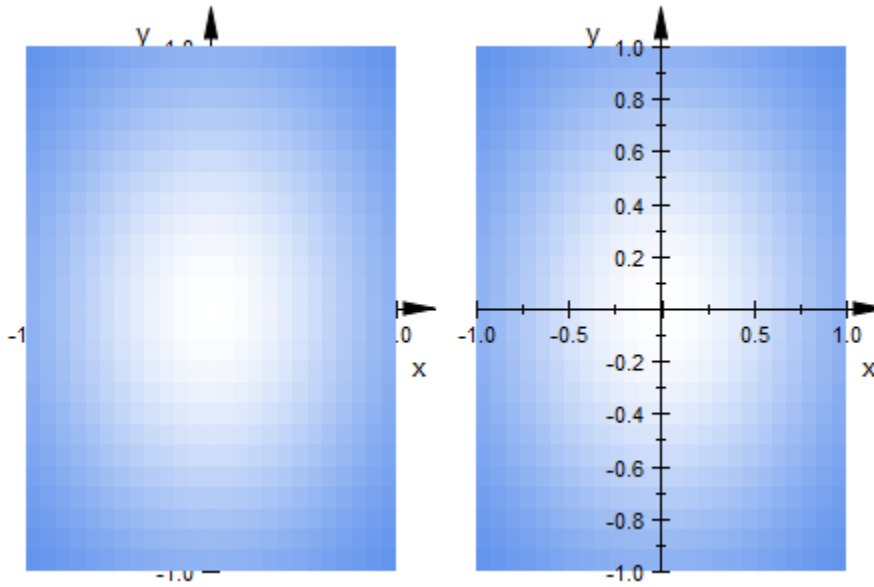
```
p1 := plot::Point2d(0, 0.62, PointSize = 3*unit::mm,
                    Color = RGB::Green):
p2 := plot::Point2d(0, 0.93, PointSize = 3*unit::mm,
                    Color = RGB::Green):
f1 := plot::Function2d(x^5*heaviside(x), x = -1 .. 1,
                       Color = RGB::Blue):
f2 := plot::Function2d(x^2, x = -1 .. 1, Color = RGB::Red):
plot(plot::Scene2d(p1, p2, f1, f2, AxesInFront = FALSE),
     plot::Scene2d(p1, p2, f1, f2, AxesInFront = TRUE)):
```



However, you probably want to have the axes visible in front of the following density plot:

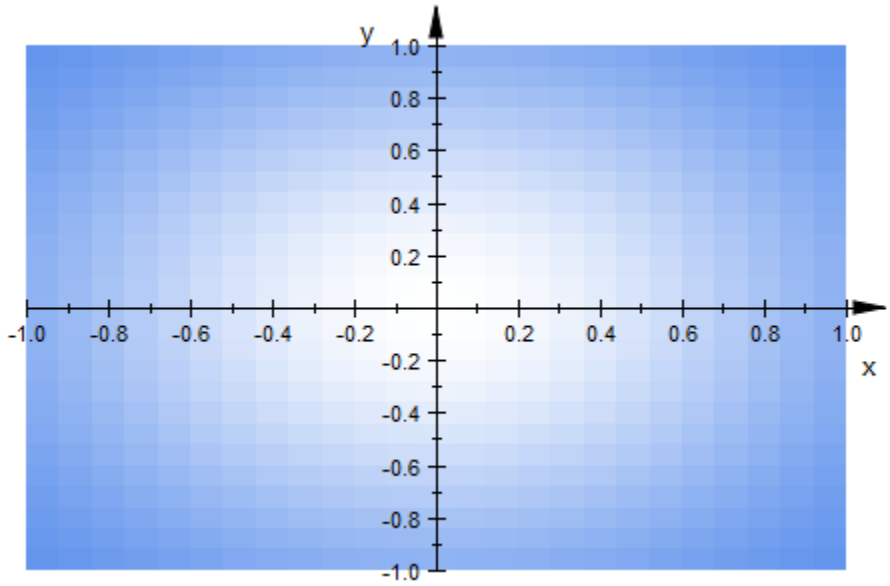
```
d := plot::Density(exp(-x^2 - y^2), x = -1..1, y = -1 ..1,
                   FillColor = RGB::White):
```

```
plot(plot::Scene2d(d, AxesInFront = FALSE),
     plot::Scene2d(d, AxesInFront = TRUE),
     Layout = Horizontal):
```



Note that density objects of type `plot::Density` automatically send the "hint" `AxesInFront = TRUE`, so there is no need to set this attribute explicitly:

```
plot(d):
```




```
delete p1, p2, f1, f2, d:
```

See Also

MuPAD Functions

Axes | AxesLineColor | AxesLineWidth | AxesOrigin | AxesTips | AxesTitleAlignment | AxesTitleFont | AxesTitles | AxesVisible | YAxisTitleOrientation

AxesLineColor

Color of the coordinate axes

Value Summary

Inherited	Color

Graphics Primitives

Objects	AxesLineColor Default Values
`plot::CoordinateSystem2d`, `plot::CoordinateSystem3d`	`RGB::Black`

Description

`AxesLineColor` sets the RGB color for the coordinate axes and the tick marks.

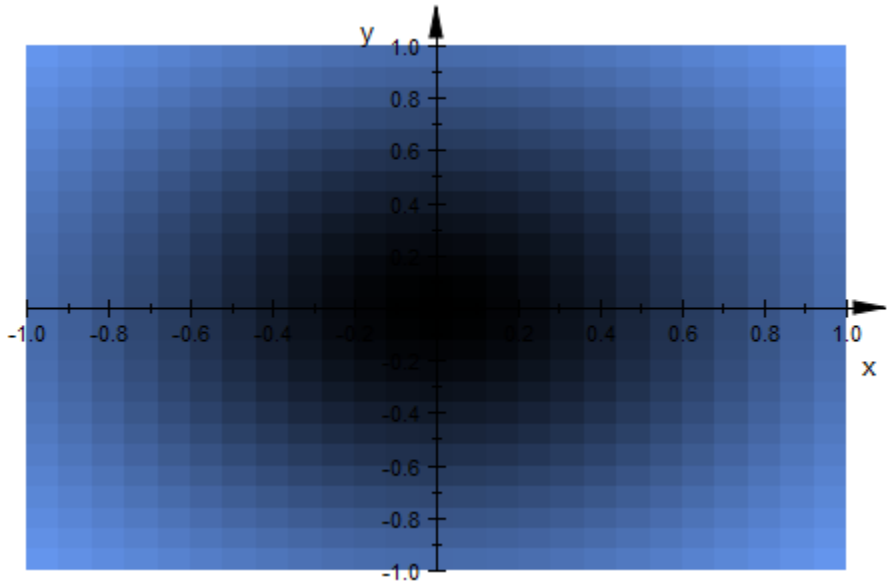
The color of the axes titles and the tick labels are *not* set by `AxesLineColor`. Choose an appropriate color for the corresponding fonts via the attributes `AxesTitleFont` and `TicksLabelFont`.

Examples

Example 1

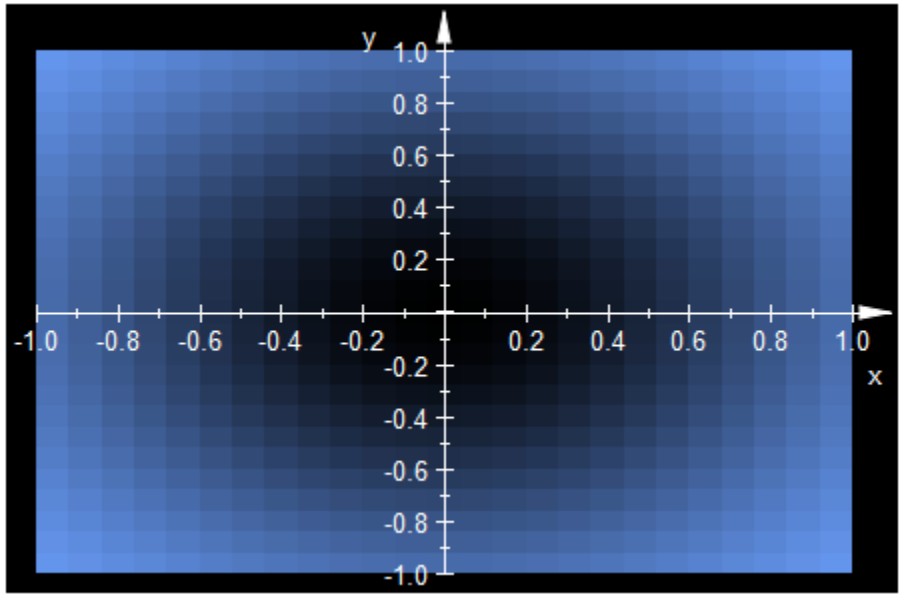
The black axes are not appropriate for the following density graphics:

```
d := plot::Density(exp(-x^2 - y^2), x = -1..1, y = -1 ..1,
                   FillColor = RGB::Black):
plot(d)
```



We change the axes color to `white' via `AxesLineColor`. The titles and the tick labels along the axes do not turn white, automatically, so we choose white font colors as well:

```
plot(d,
     AxesLineColor = RGB::White,
     AxesTitleFont = [RGB::White],
     TicksLabelFont = [RGB::White],
     plot::Scene2d::BackgroundColor = RGB::Black)
```

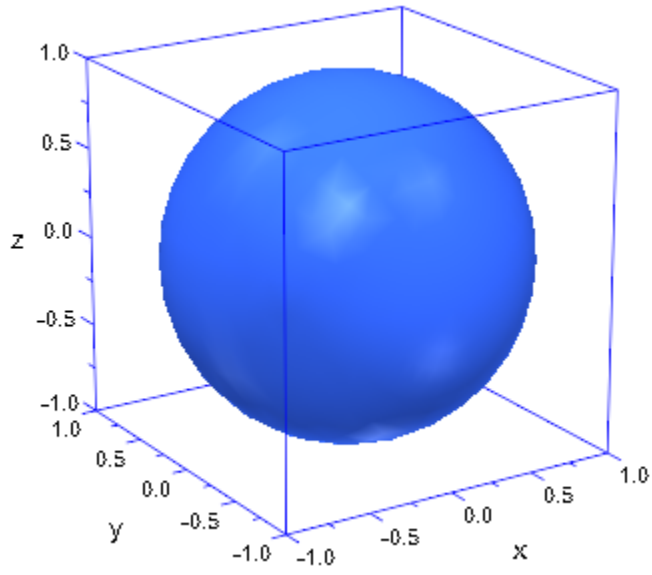



```
delete d:
```

Example 2

We display the axes as a blue box:

```
plot(plot::Sphere(1, [0, 0, 0]), AxesLineColor = RGB::Blue):
```



See Also

MuPAD Functions

`Axes` | `AxesInFront` | `AxesLineWidth` | `AxesOrigin` | `AxesTips` | `AxesTitleAlignment` | `AxesTitleFont` | `AxesTitles` | `AxesVisible` | `YAxisTitleOrientation`

AxesLineWidth

Width of the coordinate axes

Value Summary

Inherited	Positive output size

Graphics Primitives

Objects	AxesLineWidth Default Values
`plot::CoordinateSystem2d`, `plot::CoordinateSystem3d`	`0.18`

Description

`AxesLineWidth` sets the width for the coordinate axes, the ticks, and the `AxesTips`. The value should be specified as an absolute physical length including a length unit such as `AxesLineWidth = 0.5*unit::mm`. Numbers without a physical unit give the size in mm.

The length of the ticks is not affected by `AxesLineWidth` and can be set separately via `TicksLength`.

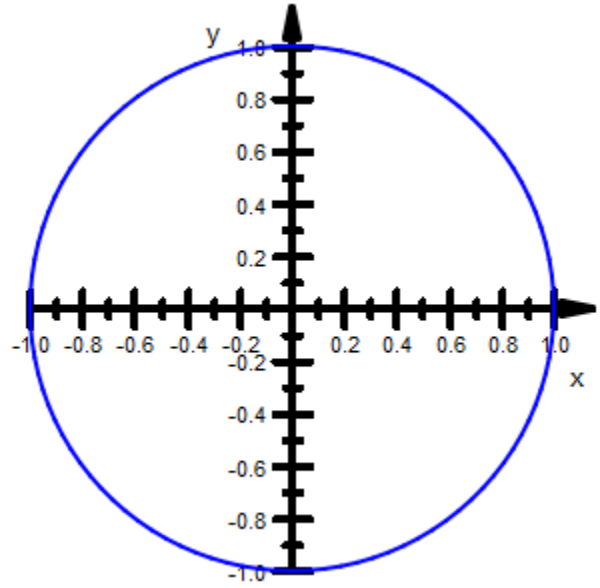
Note that the graphics cannot always react to small changes of the line width because of the discretization into pixels.

Examples

Example 1

We create a graticule with “thick” wiring. Note that `LineWidth` refers to the circles, whereas `AxesLineWidth` relates to the coordinate axes:

```
plot(plot::Circle2d(1, [0, 0]),
     TicksDistance = 0.2, TicksLength = 5*unit::mm,
     LineWidth = 0.5*unit::mm, AxesLineWidth = 1*unit::mm):
```



See Also

MuPAD Functions

Axes | AxesInFront | AxesLineColor | AxesOrigin | AxesTips | AxesTitleAlignment | AxesTitleFont | AxesTitles | AxesVisible | TicksLength | YAxisTitleOrientation

AxesOrigin, AxesOriginX, AxesOriginY, AxesOriginZ

Crosspoint of the coordinate axes

Value Summary

`AxesOrigin`	Library wrapper for "[`AxesOriginX`, `AxesOriginY`]" (2D), "[`AxesOriginX`, `AxesOriginY`, `AxesOriginZ`]" (3D)	See below
`AxesOriginX`, `AxesOriginY`, `AxesOriginZ`	Optional	MuPAD expression

Graphics Primitives

Objects	Default Values
`plot::CoordinateSystem2d`	`AxesOrigin`: [0, 0] `AxesOriginX`, `AxesOriginY`: 0
`plot::CoordinateSystem3d`	`AxesOrigin`: [0, 0, 0] `AxesOriginX`, `AxesOriginY`, `AxesOriginZ`: 0

Description

`AxesOrigin` determines the crosspoint of the coordinate axes.

These attributes only have an effect with `Axes` = `Origin`. The coordinate axes are displayed as a cross.

The vector `AxesOrigin` determines the point where the coordinate axes cross. Depending on the dimension of the scene, it is given by a list of 2 or 3 components.

`AxesOriginX` etc. refer to the x, y, z components of this point.

Note: If the crosspoint of the axes is not inside the "viewing box" of the scene, parts of the axes may not be visible.

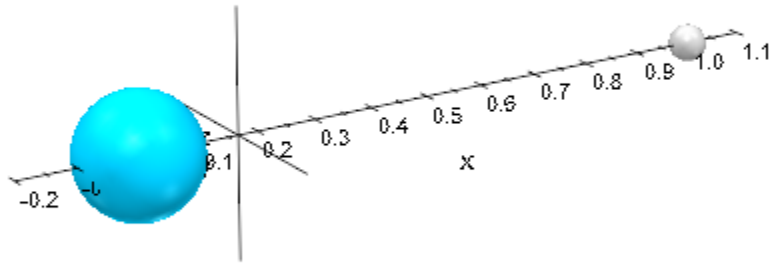
The viewing box may be set explicitly via the attribute `ViewingBox`. With `Axes` = `Automatic`, the point given by `AxesOrigin` is ignored; the crosspoint of the axes is chosen automatically inside the viewing box.

Examples

Example 1

We plot two spheres representing a planet with a moon. The coordinate axes cross at their common center of gravity:

```
m1 := 1: x1 := 0:
x2 := 1: m2 := 0.2:
earth := plot::Sphere(0.1, [x1, 0, 0],
                      FillColor = RGB::SkyBlue):
moon := plot::Sphere(0.03, [x2, 0, 0],
                     FillColor = RGB::Grey):
plot(earth, moon, Axes = Origin,
     YTicksNumber = None, ZTicksNumber = None,
     AxesOrigin = [(m1*x1 + m2*x2)/(m1 + m2), 0, 0],
     ViewingBox = [-0.2 .. 1.1, -0.2..0.2, -0.2..0.2]):
```




```
delete m1, m2, x1, x2, earth, moon:
```

See Also

MuPAD Functions

Axes | AxesInFront | AxesLineColor | AxesLineWidth | AxesTips | AxesTitleAlignment | AxesTitleFont | AxesTitles | AxesVisible | YAxisTitleOrientation

AxesTips

Arrow tips at the coordinate axes?

Value Summary

Inherited	FALSE, or TRUE

Graphics Primitives

Objects	AxesTips Default Values
plot::CoordinateSystem2d	TRUE
plot::CoordinateSystem3d	FALSE

Description

With AxesTips = TRUE, the coordinate axes are drawn with arrow tips.

This attribute only has an effect with Axes = Automatic or Axes = Origin. In both cases the coordinate axes are displayed as a cross.

With AxesTips = TRUE, little arrows are drawn on the end of the coordinate axes pointing into the positive direction.

The size of the arrow tips that are displayed as lines is controlled by AxesLineWidth.

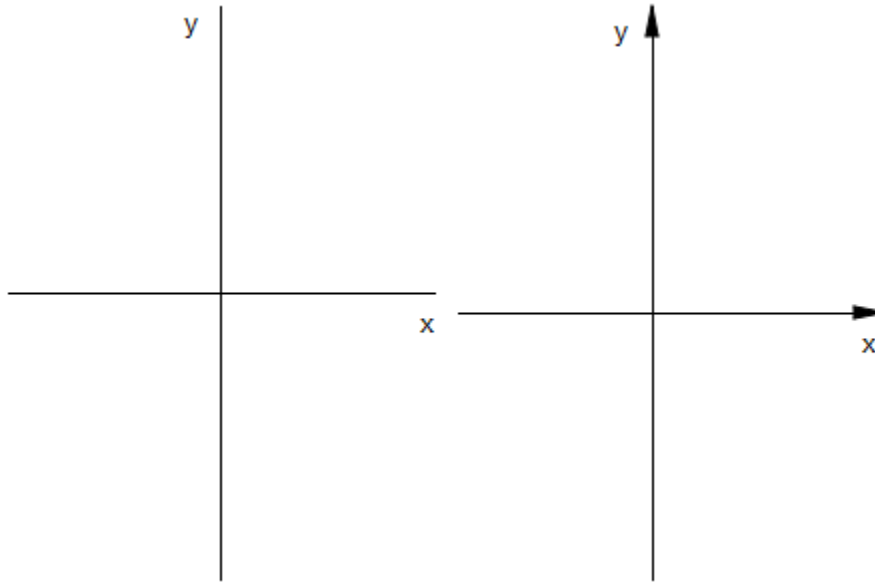
AxesTips = FALSE suppresses any coordinate axes tips.

Examples

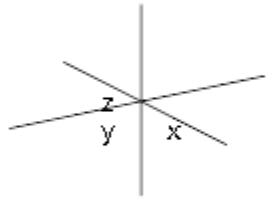
Example 1

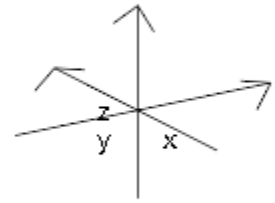
In order to emphasize on AxesTips, we plot empty scenes. The tick marks are "switched off" via TicksNumber = None:

```
S1 := plot::Scene2d(AxesTips = FALSE):
S2 := plot::Scene2d(AxesTips = TRUE):
plot(S1, S2, TicksNumber = None, Layout = Horizontal):
```




```
S1 := plot::Scene3d(AxesTips = FALSE):
S2 := plot::Scene3d(AxesTips = TRUE):
plot(S1, S2, Axes = Origin, TicksNumber = None,
     Layout = Horizontal):
```





See Also

MuPAD Functions

`Axes` | `AxesInFront` | `AxesLineColor` | `AxesLineWidth` | `AxesOrigin` | `AxesTitleAlignment` | `AxesTitleFont` | `AxesTitles` | `AxesVisible` | `TipAngle` | `TipLength` | `TubeDiameter` | `YAxisTitleOrientation`

AxesTitleAlignment, XAxisTitleAlignment, YAxisTitleAlignment, ZAxisTitleAlignment

Alignment of axes titles

Value Summary

`AxesTitleAlignment`	Library wrapper for "{`XAxisTitleAlignment`, `YAxisTitleAlignment`}" (2D), "{`XAxisTitleAlignment`, `YAxisTitleAlignment`, `ZAxisTitleAlignment`}" (3D)	See below
`XAxisTitleAlignment`, `YAxisTitleAlignment`, `ZAxisTitleAlignment`	Inherited	`Begin`, `Center`, or `End`

Graphics Primitives

Objects	Default Values
`plot::CoordinateSystem2d`	`AxesTitleAlignment`, `XAxisTitleAlignment`, `YAxisTitleAlignment`: `End`
`plot::CoordinateSystem3d`	`AxesTitleAlignment`, `XAxisTitleAlignment`, `YAxisTitleAlignment`, `ZAxisTitleAlignment`: `Center`

Description

`AxesTitleAlginment` governs the alignment of axes titles along the coordinate axes.

With `AxesTitleAlignment = End`, titles for all coordinate axes are displayed at that end of the axes with higher coordinate values.

With `AxesTitleAlignment = Begin`, titles are displayed at that end of the axes with lower coordinate values.

With `AxesTitleAlignment = Center`, titles are centered along the axes.

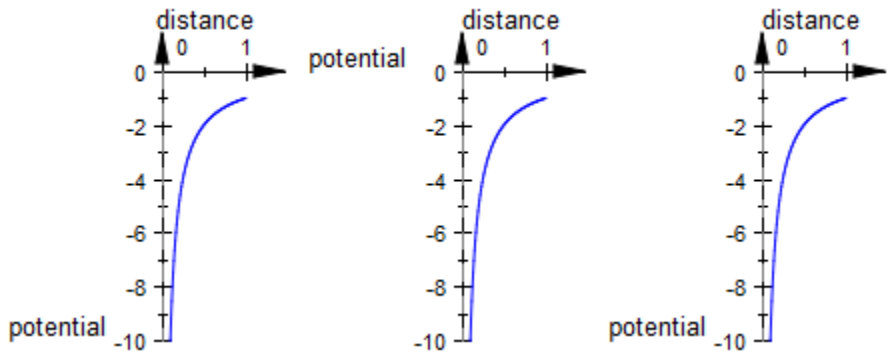
`XAxisTitleAlignment` etc. allow to set the title alignments separately for each single axis.

Examples

Example 1

We plot the Coulomb potential of a charged particle:

```
F := plot::Function2d(-1/r, r = 0..1,
                      ViewingBoxYRange = -10..0):
S1 := plot::Scene2d(F, AxesTitles = ["distance", "potential"],
                    XAxisTitleAlignment = Center,
                    YAxisTitleAlignment = Begin):
S2 := plot::Scene2d(F, AxesTitles = ["distance", "potential"],
                    XAxisTitleAlignment = Begin,
                    YAxisTitleAlignment = End):
S3 := plot::Scene2d(F, AxesTitles = ["distance", "potential"],
                    XAxisTitleAlignment = Begin,
                    YAxisTitleAlignment = Begin):
plot(S1, S2, S3, Layout = Horizontal,
     Width = 120*unit::mm, Height = 50*unit::mm):
```

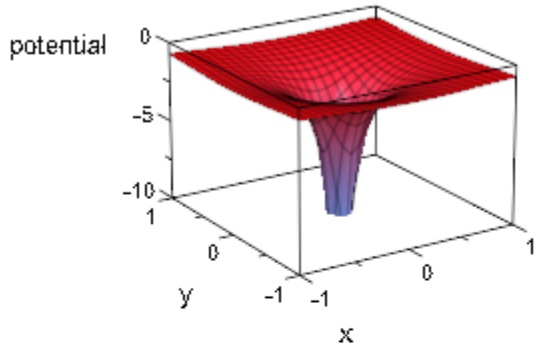


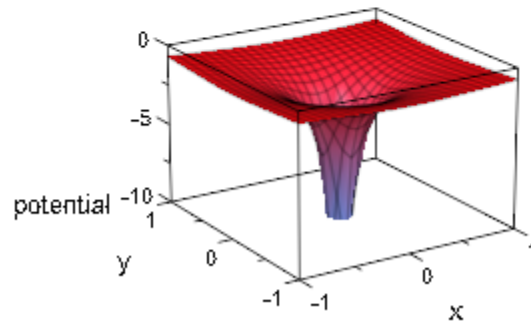
```
delete F, S1, S2, S3:
```

Example 2

We use the 3D analogue of the previous example to demonstrate the alignment of axes titles in 3D: :

```
F := plot::Function3d(-1/sqrt(x^2 + y^2), x = -1..1, y = -1..1,
                      ViewingBoxZRange = -10 .. 0):
S1 := plot::Scene3d(F, AxesTitles = ["x", "y", "potential"],
                    XAxisTitleAlignment = Begin,
                    YAxisTitleAlignment = Center,
                    ZAxisTitleAlignment = End):
S2 := plot::Scene3d(F, AxesTitles = ["x", "y", "potential"],
                    XAxisTitleAlignment = Center,
                    YAxisTitleAlignment = End,
                    ZAxisTitleAlignment = Begin):
plot(S1, S2, Layout = Vertical,
     Width = 80*unit::mm, Height = 120*unit::mm):
```






```
delete F, S1, S2:
```

See Also

MuPAD Functions

Axes | AxesInFront | AxesLineColor | AxesLineWidth | AxesOrigin | AxesTips | AxesTitleFont | AxesTitles | AxesVisible | YAxisTitleOrientation

AxesTitles, XAxisTitle, YAxisTitle, ZAxisTitle

Titles for the coordinate axes

Value Summary

`AxesTitles`	Library wrapper for "[`XAxisTitle`, `YAxisTitle`]" (2D), "[`XAxisTitle`, `YAxisTitle`, `ZAxisTitle`]" (3D)	See below
`XAxisTitle`, `YAxisTitle`, `ZAxisTitle`	Optional	Text string

Graphics Primitives

Objects	Default Values

`plot::CoordinateSystem2d`	`AxesTitles`: `[" x "," y "]`
	`XAxisTitle`: `" x "`
	`YAxisTitle`: `" y "`

`plot::CoordinateSystem3d`	`AxesTitles`: `[" x "," y "," z "]`
	`XAxisTitle`: `" x "`
	`YAxisTitle`: `" y "`
	`ZAxisTitle`: `" z "`

Description

`AxesTitles` sets the titles attached to the coordinate axes.

Depending on the dimension of the coordinate system, the value of the attribute `AxisTitles` must be a list with two or three strings.

Per default, the coordinate axes titles are `["x", "y"]` in 2D and `["x", "y", "z"]` in 3D regardless of the names of involved parameters. Cf. “Example 1” on page 24-1565.

Using `AxesTitles`, axes titles can be edited as desired.

With `XAxisTitle` etc., the titles can be edited separately for the different coordinate directions.

Set empty strings `AxesTitles = ["",""]` in 2D or `AxesTitles = ["","",""]` in 3D, respectively, if no axes titles shall be displayed.

Some objects in the MuPAD `plot` library override the default setting via the “hint mechanism” (see the section Primitives Requesting Special Scene Attributes: “Hints” in this document). Whenever such an object is plotted in a scene, the axes titles chosen by the object are used. A complete list of these objects is given further up on this help page.

You can still override these titles via `AxesTitles` etc.

The attribute `AxesTitleAlignment` can be used to change the default alignment of the titles along the axes.

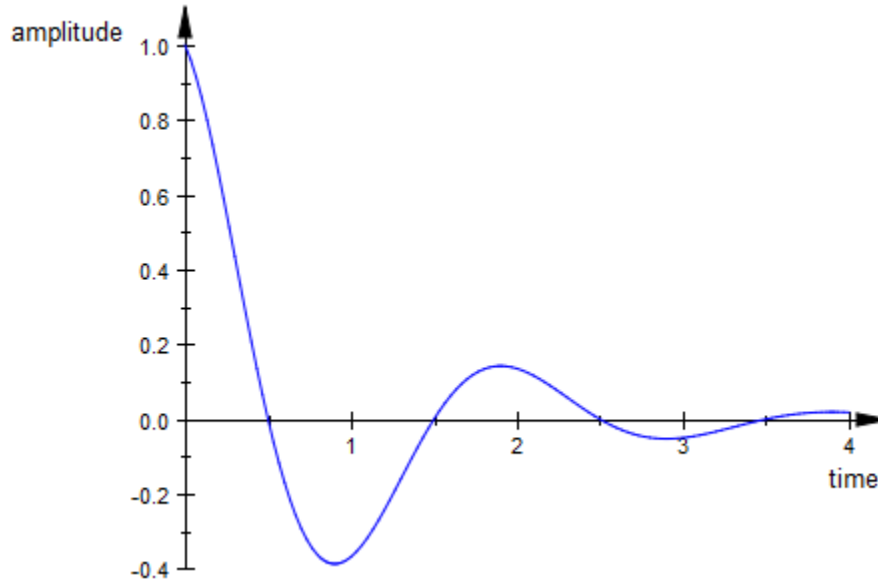
The attribute `YAxisTitleOrientation` can be used in 2D to rotate the title of the vertical axis.

Examples

Example 1

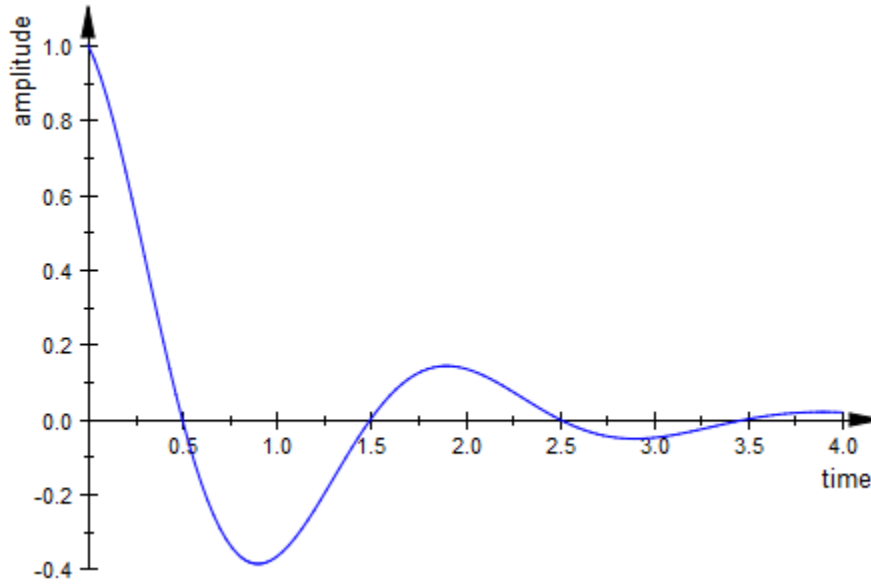
We set appropriate axes titles for a damped vibration given by a plot of the “amplitude over time”:

```
F := plot::Function2d(exp(-t)*cos(PI*t), t = 0 .. 4):
plot(F, AxesTitles = ["time", "amplitude"]):
```



It might be desirable to use the attribute `YAxisTitleOrientation` to twist the title for the vertical axis:

```
plot(F, AxesTitles = ["time", "amplitude"],
     YAxisTitleOrientation = Vertical):
```



See Also

MuPAD Functions

`Axes` | `AxesInFront` | `AxesLineColor` | `AxesLineWidth` | `AxesOrigin` | `AxesTips` | `AxesTitleAlignment` | `AxesTitleFont` | `AxesVisible` | `YAxisTitleOrientation`

AxesVisible, XAxisVisible, YAxisVisible, ZAxisVisible

Display coordinate axes?

Value Summary

`AxesVisible`	Library wrapper for "{`XAxisVisible`, `YAxisVisible`}" (2D), "{`XAxisVisible`, `YAxisVisible`, `ZAxisVisible`}" (3D)	`TRUE`, `FALSE`, or list of 2 or 3 of these, depending on the dimension
`XAxisVisible`, `YAxisVisible`, `ZAxisVisible`	Inherited	`FALSE`, or `TRUE`

Graphics Primitives

Objects	Default Values
`plot::CoordinateSystem2d`	`AxesVisible`, `XAxisVisible`, `YAxisVisible`: `TRUE`
`plot::CoordinateSystem3d`	`AxesVisible`, `XAxisVisible`, `YAxisVisible`, `ZAxisVisible`: `TRUE`

Description

With `AxesVisible` = `TRUE` versus `AxesVisible` = `FALSE` all coordinate axes are "switched on" or "off".

With `XAxisVisible` etc., the coordinate axes in the different coordinate directions can be switched on and off, separately.

With `Axes = Box`, the coordinate axes are displayed as a box about the scene. With `XAxisVisible = FALSE` etc., the four edges of this box parallel to the respective axis are suppressed.

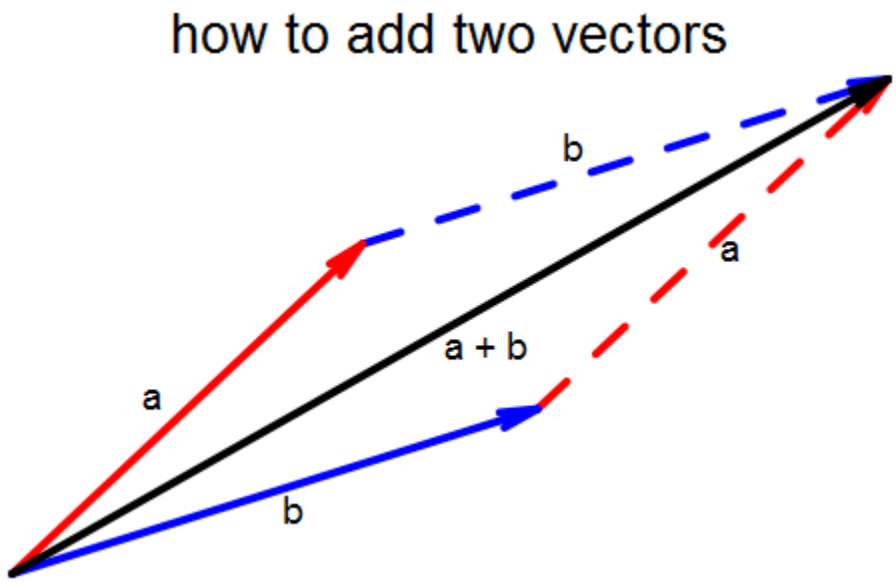
Alternatively to `AxesVisible = FALSE`, you may switch the axes off by setting `Axes = None`, too.

Examples

Example 1

In the following illustration, you probably do not want any axes:

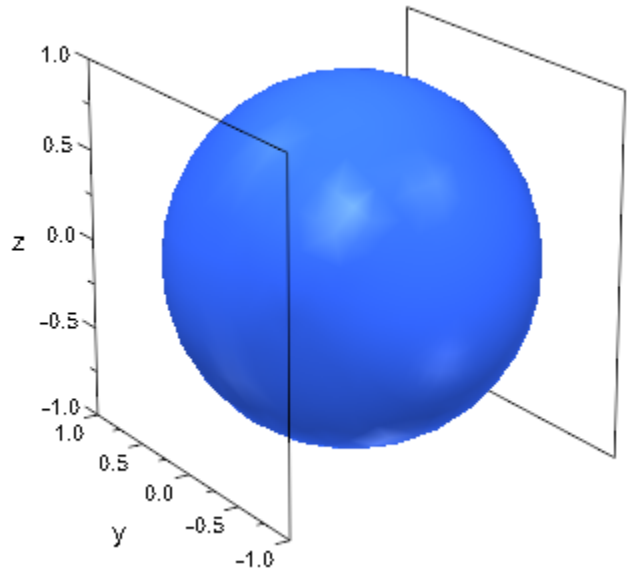
```
plot(
  plot::Arrow2d([1.5, 1], [2.5, 3],
                Title = "a", TitlePosition = [2.05, 1.9],
                LineStyle = Dashed, Color = RGB::Red),
  plot::Arrow2d([1, 2], [2.5, 3],
                Title = "b", TitlePosition = [1.6, 2.5],
                LineStyle = Dashed, Color = RGB::Blue),
  plot::Arrow2d([0, 0], [1, 2], Color = RGB::Red,
                 Title = "a", TitlePosition = [0.4, 1.0]),
  plot::Arrow2d([0, 0], [1.5, 1], Color = RGB::Blue,
                Title = "b", TitlePosition = [0.8, 0.3]),
  plot::Arrow2d([0, 0], [2.5, 3], Color = RGB::Black,
                Title = "a + b", TitlePosition = [1.35, 1.3]),
  AxesVisible = FALSE, TitleFont = [14],
  TipLength = 5.0*unit::mm, LineWidth = 1.0*unit::mm,
  HeaderFont = [20], Header = "how to add two vectors"
)
```



Example 2

Using the default axes style `Axes = Box` in 3D, we suppress all parts of the axes box in the x direction:

```
plot(plot::Sphere(1, [0, 0, 0]), XAxisVisible = FALSE):
```



See Also

MuPAD Functions

`Axes` | `AxesInFront` | `AxesLineColor` | `AxesLineWidth` | `AxesOrigin` | `AxesTips` | `AxesTitleAlignment` | `AxesTitleFont` | `AxesTitles` | `YAxisTitleOrientation`

YAxisTitleOrientation

Orientation of the vertical axis title in 2D

Value Summary

Inherited	`Horizontal`, or `Vertical`

Graphics Primitives

Objects	YAxisTitleOrientation Default Values
`plot::CoordinateSystem2d`	`Horizontal`

Description

`YAxisTitleOrientation` determines whether the title of the vertical axis in 2D is plotted horizontally or vertically.

If the title of the vertical axis in 2D is long, it uses up a lot of horizontal space when rendered from left to right with `YAxisTitleOrientation` = Horizontal. This space may be taken away from the drawing region for the graphical objects. In such a case it might be desirable to use `YAxisTitleOrientation = Vertical` to let the title be rendered from bottom to top instead, parallel to the vertical axis.

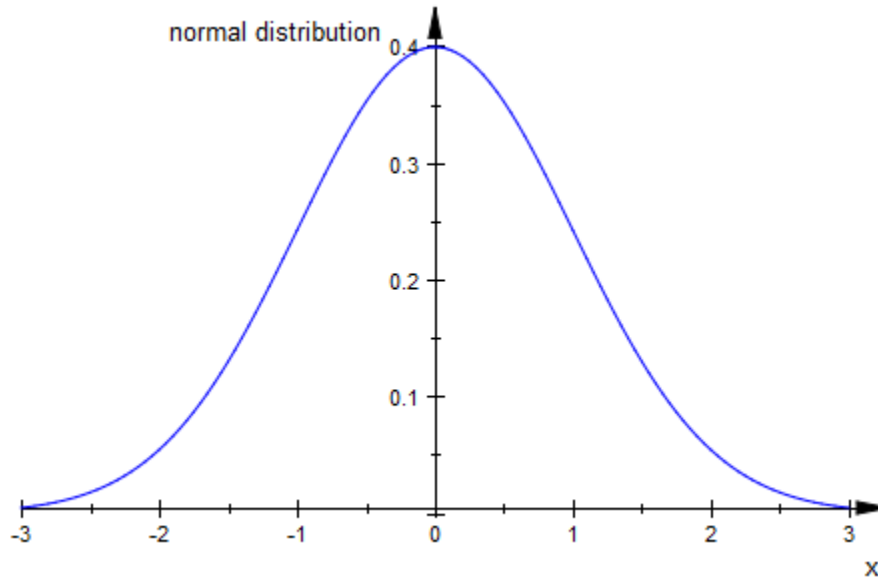
This attribute is ignored in 3D.

Examples

Example 1

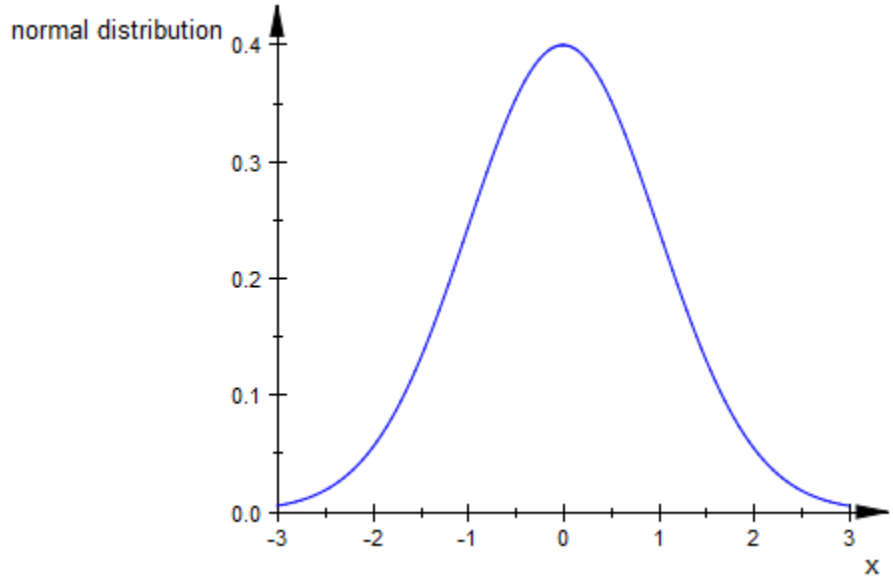
We plot the density of the normal distribution function:

```
f := plot::Function2d(stats::normalPDF(0, 1)(x), x = -3 .. 3):
plot(f, Axes = Automatic,
     AxesTitles = ["x", "normal distribution"],
     YAxisTitleOrientation = Horizontal):
```



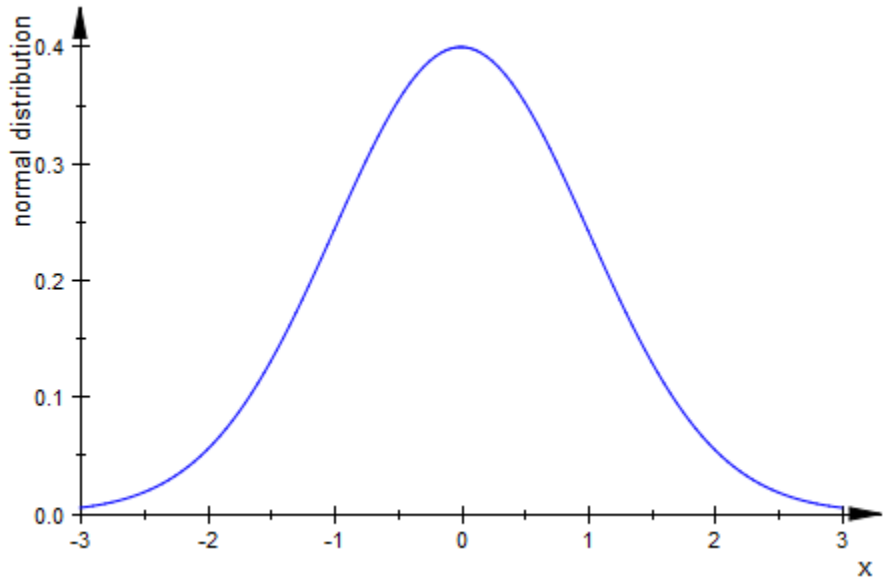
There is plenty of room to draw the long title `"normal distribution"`, because the vertical axis is placed in the middle of the plot. In the next plot, however, the vertical axis is flushed left and a lot of space is “wasted” for the axis title:

```
plot(f, Axes = Frame,
     AxesTitles = ["x", "normal distribution"],
     YAxisTitleOrientation = Horizontal):
```



You make better use of the drawing area by plotting the title of the vertical axis parallel to this axis:

```
plot(f, Axes = Frame,
     AxesTitles = ["x", "normal distribution"],
     YAxisTitleOrientation = Vertical):
```




```
delete f:
```

See Also

MuPAD Functions

Axes | AxesInFront | AxesLineColor | AxesLineWidth | AxesOrigin | AxesTips | AxesTitleAlignment | AxesTitles | AxesVisible | YAxisTitleOrientation

TicksAnchor, XTicksAnchor, YTicksAnchor, ZTicksAnchor

User defined start of axes tick marks

Value Summary

`TicksAnchor`	Library wrapper for "{`XTicksAnchor`, `YTicksAnchor`}" (2D), "{`XTicksAnchor`, `YTicksAnchor`, `ZTicksAnchor`}" (3D)	MuPAD expression
`XTicksAnchor`, `YTicksAnchor`, `ZTicksAnchor`	Optional	MuPAD expression

Graphics Primitives

Objects	Default Values
`plot::CoordinateSystem2d`	`TicksAnchor`, `XTicksAnchor`, `YTicksAnchor`: `0`
`plot::CoordinateSystem3d`	`TicksAnchor`, `XTicksAnchor`, `YTicksAnchor`, `ZTicksAnchor`: `0`

Description

With `TicksAnchor` = t_0, `TicksDistance` = `d`, the automatic ticks along the coordinate axes are switched off and replaced by equidistant ticks with distance `d` at the positions $t_j = t_0 + j\,d, j \in \mathbb{Z}$.

With `TicksAnchor` = t_0, `TicksDistance` = `d`, these ticks are used for all coordinate axes.

With `XTicksAnchor` = t_0, `XTicksDistance` = `d` etc., these ticks may be defined separately for each single coordinate axis.

When executing a plot command, per default a "reasonable" placing for tick marks on coordinate axes is automatically computed. Through this process tick marks may not come to lie on desired positions. The attributes `TicksAnchor` and `TicksDistance` allow to generate an alternative mesh of equidistant tick marks.

Note: The attributes `TicksAnchor`, `XTicksAnchor` etc. only have an effect when a positive distance $d > 0$ between major ticks marks is set explicitly via `TicksDistance` = `d`, `XTicksDistance` = `d` etc.

The ticks set by `TicksAnchor` and `TicksDistance` are "major" tick marks bearing labels. Depending on the value of `TicksBetween`, there may be additional "minor" ticks without labels between each pair of major tick marks.

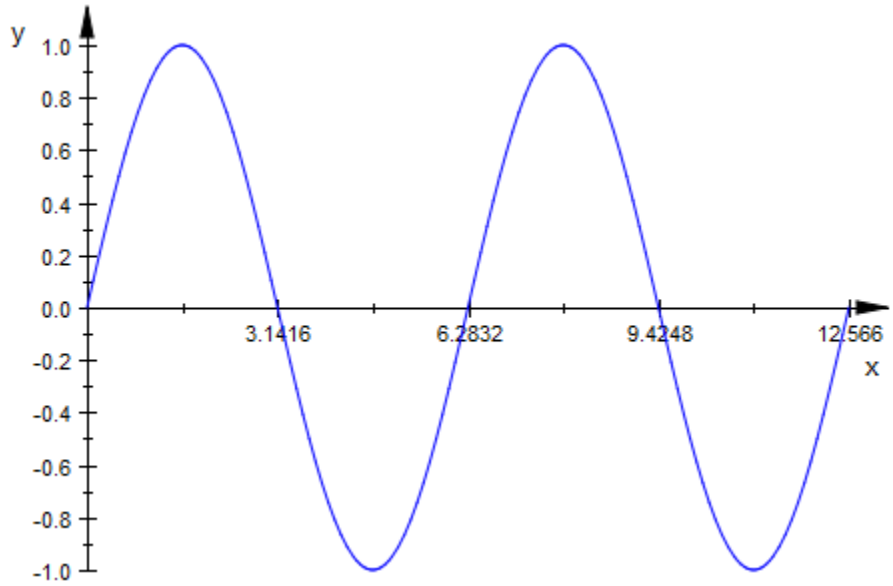
Additional tick marks at specific positions can be inserted with `TicksAt`.

Examples

Example 1

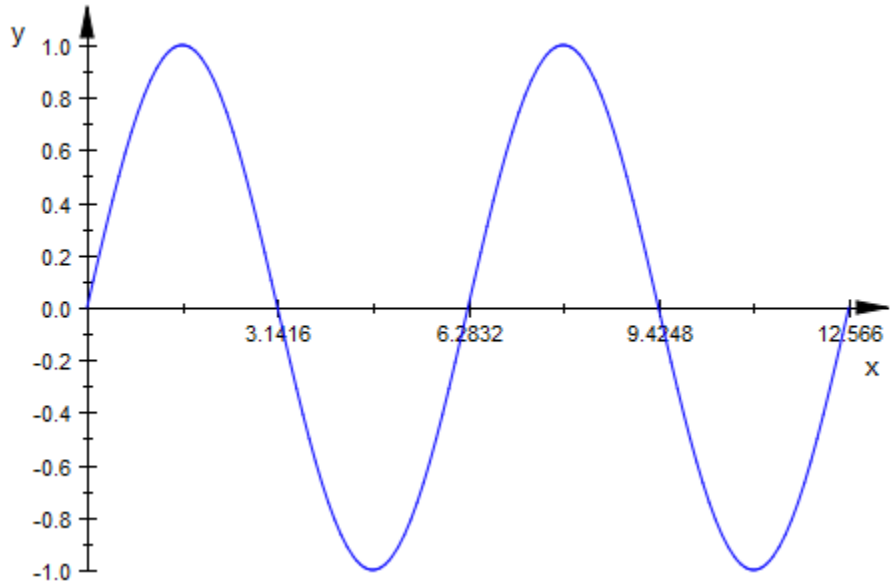
For the following plot of the sine function, the tick marks along the x-axis are chosen to match the period:

```
plot(plot::Function2d(sin(x), x = 0..4*PI),
     XTicksAnchor = 0, XTicksDistance = PI):
```



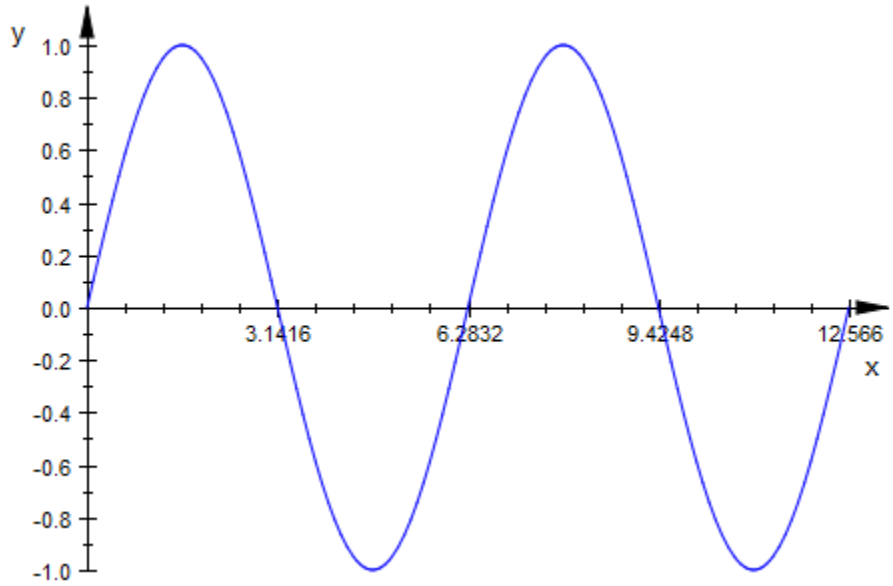
The ticks along the *y*-axis are re-defined with a distance of 0.2:

```
plot(plot::Function2d(sin(x), x = 0..4*PI),
     XTicksAnchor = 0, XTicksDistance = PI,
     YTicksAnchor = 0, YTicksDistance = 0.2):
```



We increase the number of "minor" ticks along the *x*-axis:

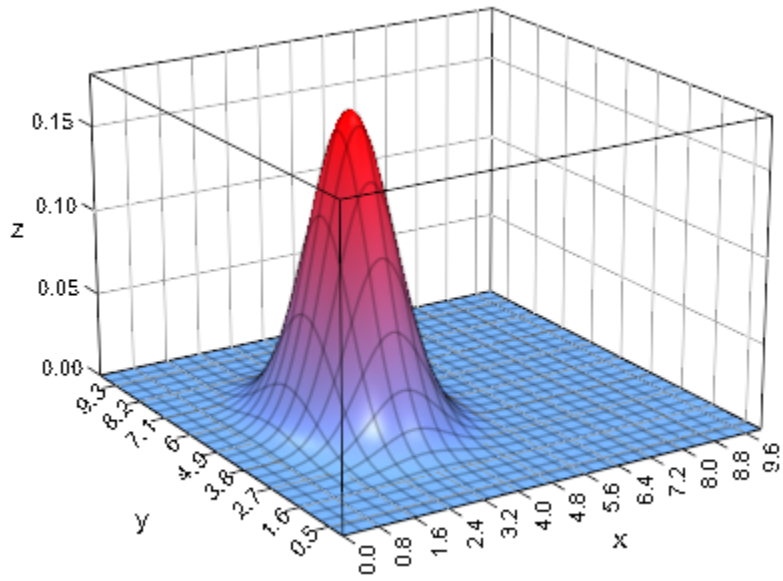
```
plot(plot::Function2d(sin(x), x = 0..4*PI),
     XTicksAnchor = 0, XTicksDistance = PI,
     XTicksBetween = 4,
     YTicksAnchor = 0, YTicksDistance = 0.2):
```



Example 2

We plot the two-dimensional normal distribution centered around the mean (m_1, m_2) = (3.2, 4.9). This point is used as the anchor for the tick marks along the x-axis and the y-axis, respectively. Ticks are positioned at distances that are integer multiples of the standard deviations (s_1, s_2) = (0.8, 1.2):

```
m1:= 3.2: s1 := 0.8:
m2:= 4.9: s2 := 1.1:
plot(plot::Function3d( stats::normalPDF(m1, s1^2)(x)
                      *stats::normalPDF(m2, s2^2)(y),
                      x = 0 .. 10, y = 0 .. 10,
                      Submesh = [3, 3]),
     TicksBetween = 0,
     XTicksAnchor = m1, XTicksDistance = s1,
     YTicksAnchor = m2, YTicksDistance = s2,
     XTicksLabelStyle = Vertical,
     YTicksLabelStyle = Diagonal,
     GridVisible = TRUE):
```




```
delete m1, s1, m2, s2:
```

See Also

MuPAD Functions

TicksAt | TicksBetween | TicksDistance | TicksLabelFont | TicksLabelStyle | TicksLabelsVisible | TicksLength | TicksNumber | TicksVisible

TicksAt, XTicksAt, YTicksAt, ZTicksAt

Special axes tick marks

Value Summary

`TicksAt`	Library wrapper for "[`XTicksAt`, `YTicksAt`]" (2D), "[`XTicksAt`, `YTicksAt`, `ZTicksAt`]" (3D)	See below
`XTicksAt`, `YTicksAt`, `ZTicksAt`	Optional	See below

Graphics Primitives

Objects	Default Values
`plot::CoordinateSystem2d`, `plot::CoordinateSystem3d`	

Description

`XTicksAt` = `[`x_1`, `x_2`, …]` allows to set additional tick marks on the *x*-axis at the positions x_1, x_2 etc. With `XTicksAt` = `[`x_1 `=` L_1`, `x_2 `=` L_2`, …]`, the special tick marks at the positions x_1, x_2 etc. are labeled with the strings L_1, L_2 etc.

`YTicksAt`, `ZTicksAt` work analogously for the other coordinate directions.

`TicksAt = [[`x_1`, `x_2`, …], [`y_1`, `y_2`, …]]` in 2D and `TicksAt = [[`x_1`, `x_2`, …], [`y_1`, `y_2`, …], [`z_1`, `z_2`, …]]` in 3D serve as shortcuts for setting `XTicksAt`, `YTicksAt` etc.

Per default, equidistant tick marks along the coordinate axes are chosen automatically.

With `XTicksAt = [x1, x2, ...]`, *additional* tick marks are inserted along the *x*-axis at arbitrary positions `x1`, `x2` etc. These values must be numbers or exact numerical expressions such as PI or `sqrt(2)` that can be converted to floating-point numbers via `float`.

The special ticks set by `XTicksAt` are labeled automatically by floating-point numbers approximating `x1`, `x2` etc.

Special labels for these ticks may be requested by replacing the coordinate values `x1`, `x2` etc. by equations `x1 = L1`, `x2 = L2` etc., where `L1`, `L2` etc. are strings to be used as the labels. Note that MuPAD strings have to be enclosed by the string delimiters `"`. For example, `XTicksAt = [3.14 = "pi"]` adds a single tick as the position $x = 3.14$ with the label `pi`. Cf. "Example 1" on page 24-1583.

With `YTicksAt = [y1, y2, ...]` or `YTicksAt = [y1 = L1, y2 = L2, ...]` etc., special ticks can be inserted along the *y*-axis.

In 3D, `ZTicksAt` allows to insert special ticks along the *z*-axis.

If no automatic tick marks are desired, set `TicksNumber = None` or `XTicksNumber = None` etc. to switch them off on all coordinate axes or on single coordinate axes, respectively.

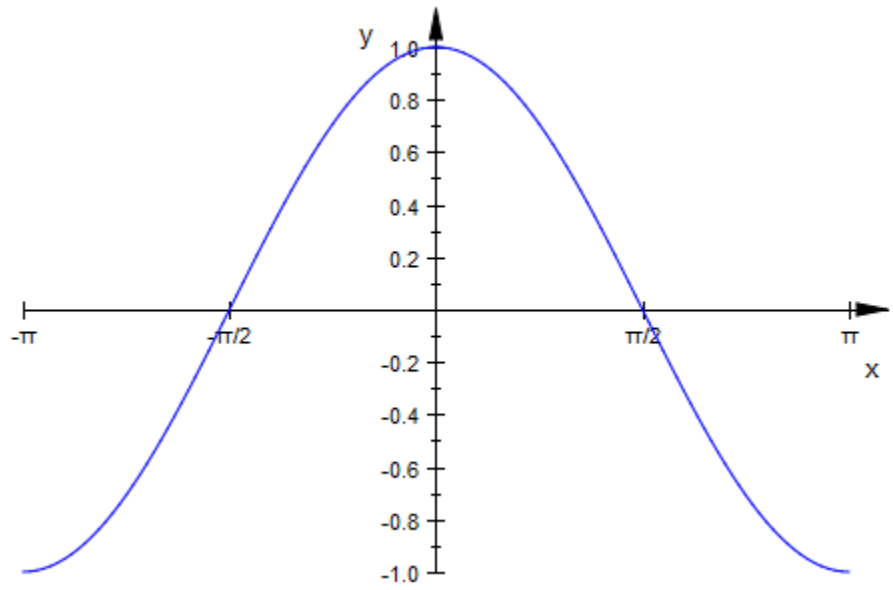
Use `TicksAt`, `XTicksAt` etc. to set alternative tick marks.

Examples

Example 1

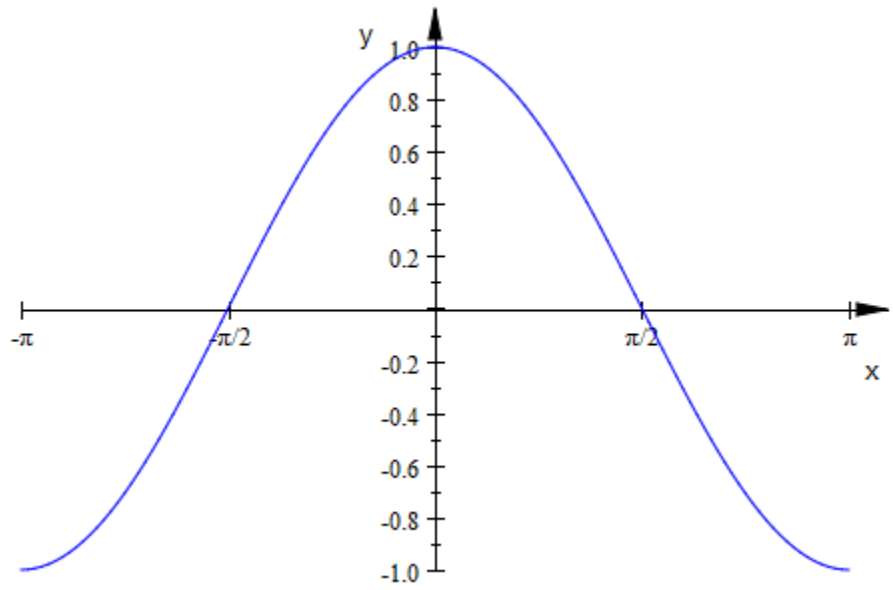
We plot the cosine function. The automatic tick marks along the *x*-axis are suppressed via `XTicksNumber = None`. Points of special interest such as the extrema and the zeroes of the function are set as special tick marks:

```
plot(plot::Function2d(cos(x), x = -PI..PI),
     XTicksNumber = None,
     XTicksAt = [-PI = "-π", -PI/2 = "-π/2",
                 0 = "0", PI/2 = "π/2", PI = "π"])
```



We improve the labeling of the tick marks by specifying the font . This font allows to typeset Greek characters such as π better:

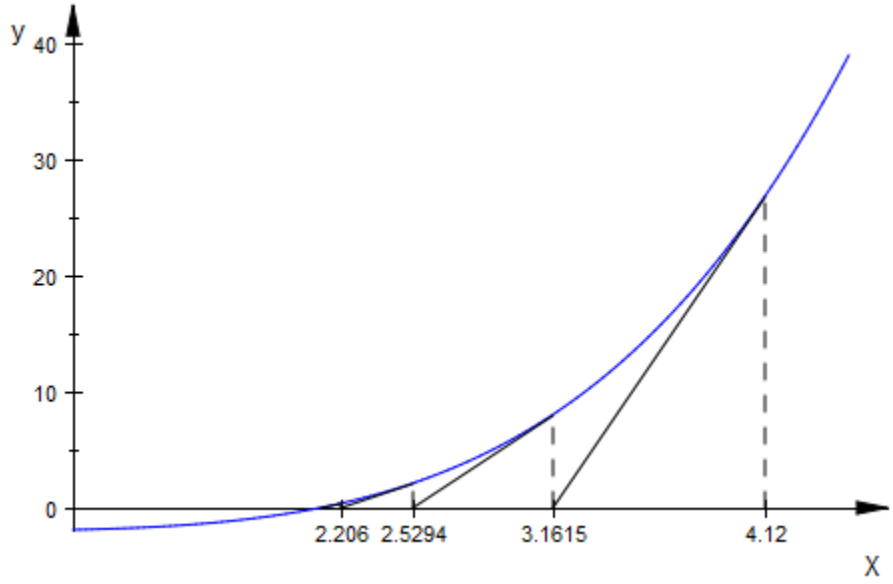
```
plot(plot::Function2d(cos(x), x = -PI..PI),
     XTicksNumber = None, TicksLabelFont = ["Times New Roman"],
     XTicksAt = [-PI = "-π", -PI/2 = "-π/2",
                 0 = "0", PI/2 = "π/2", PI = "π"])
```



Example 2

The Newton iteration $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$ finds successive approximations to a zero of a function $f(x)$. We switch the automatic ticks along the x-axis off via `XTicksNumber = None` and display some elements of the Newton sequence as tick marks:

```
f := x -> x^4/10 - 2:
x[0] := 4.12:
for i from 0 to 3 do
   x[i + 1] := x[i] - f(x[i])/f'(x[i]);
end_for:
plot(plot::Function2d(f(X), X = 1..4.5),
     plot::Line2d([x[i], f(x[i])], [x[i+1], 0],
                  Color = RGB::Black) $ i = 0..3,
     plot::Line2d([x[i], 0], [x[i], f(x[i])],
                  Color = RGB::Black,
                  LineStyle = Dashed) $ i = 0..4,
     XTicksNumber = None, XTicksAt = [x[i] $ i = 0..3])
```




```
delete f, x, i:
```

See Also

MuPAD Functions

`TicksAnchor` | `TicksBetween` | `TicksDistance` | `TicksLabelFont` | `TicksLabelStyle` | `TicksLabelsVisible` | `TicksLength` | `TicksNumber` | `TicksVisible`

TicksBetween, XTicksBetween, YTicksBetween, ZTicksBetween

Number of minor (unlabeled) axes tick marks between major (labeled) axes tick marks

Value Summary

`TicksBetween`	Library wrapper for "{`XTicksBetween`, `YTicksBetween`}" (2D), "{`XTicksBetween`, `YTicksBetween`, `ZTicksBetween`}" (3D)	Non-negative integer
`XTicksBetween`, `YTicksBetween`, `ZTicksBetween`	Inherited	Non-negative integer

Graphics Primitives

Objects	Default Values
`plot::CoordinateSystem2d`	`TicksBetween`, `XTicksBetween`, `YTicksBetween`: 1
`plot::CoordinateSystem3d`	`TicksBetween`, `XTicksBetween`, `YTicksBetween`, `ZTicksBetween`: 1

Description

The tick marks along the coordinate axes consist of "major" tick marks bearing labels and of "minor" tick marks without labels.

`TicksBetween` sets the number of minor ticks between each pair of major ticks for all coordinate axes.

With `XTicksBetween` etc., the number of minor ticks may be set separately for each single coordinate axis.

Per default between every two major tick marks one minor tick mark is rendered. Via `TicksBetween` this number can be increased or set to zero. In contrast to major tick marks, minor tick marks are never labelled.

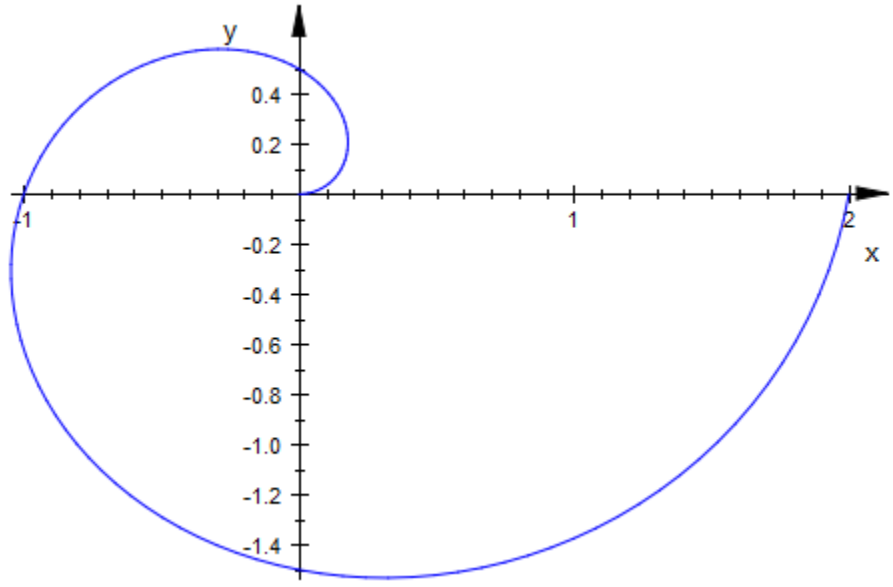
Minor tick marks are rendered always with half the length of the major tick marks. Cf. `TicksLength`.

Examples

Example 1

We request few "major" tick marks in the *x* direction and place 9 "minor" tick marks between each pair. The ticks in *y* direction are chosen automatically:

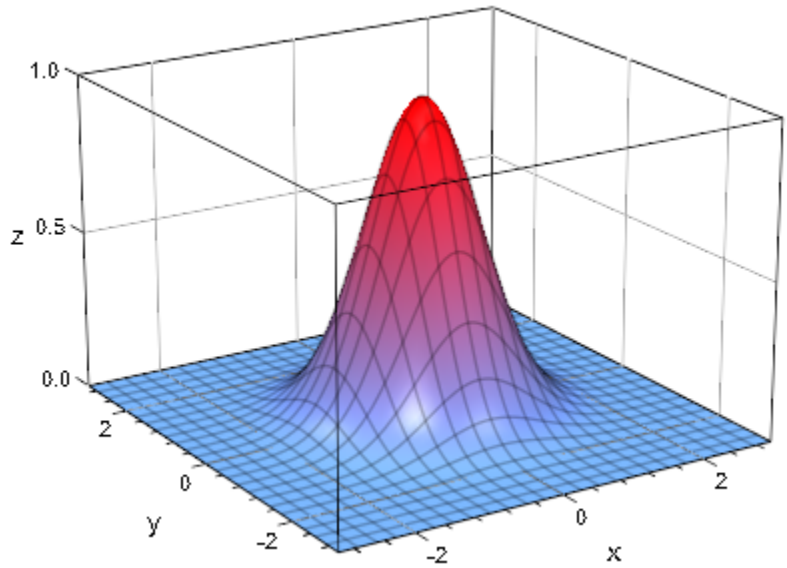
```
plot(plot::Curve2d([u*cos(u*PI), u*sin(u*PI)], u = 0..2),
     XTicksNumber = Low, XTicksBetween = 9)
```



Example 2

We request few “major” tick marks in all directions. In the horizontal directions, we place 4 “minor” tick marks between each pair. The ticks in z direction consist of the labeled ticks only:

```
plot(plot::Function3d(exp(-x^2 - y^2), x = -3..3, y = -3..3,
                      Submesh = [2, 2]),
     TicksNumber = Low, XTicksBetween = 4, YTicksBetween = 4,
     ZTicksBetween = 0, GridVisible = TRUE)
```



See Also

MuPAD Functions

TicksAnchor | TicksAt | TicksDistance | TicksLabelFont | TicksLabelStyle | TicksLabelsVisible | TicksLength | TicksNumber | TicksVisible

TicksDistance, XTicksDistance, YTicksDistance, ZTicksDistance

User defined axes tick mark distance

Value Summary

`TicksDistance`	Library wrapper for "{`XTicksDistance`, `YTicksDistance`}" (2D), "{`XTicksDistance`, `YTicksDistance`, `ZTicksDistance`}" (3D)	MuPAD expression
`XTicksDistance`, `YTicksDistance`, `ZTicksDistance`	Optional	MuPAD expression

Graphics Primitives

Objects	Default Values
`plot::CoordinateSystem2d`	`TicksDistance`, `XTicksDistance`, `YTicksDistance`: `0`
`plot::CoordinateSystem3d`	`TicksDistance`, `XTicksDistance`, `YTicksDistance`, `ZTicksDistance`: `0`

Description

With `TicksAnchor = t0`, `TicksDistance = d`, the automatic ticks along the coordinate axes are switched off and replaced by equidistant ticks with distance `d` at the positions $t_j = t_0 + j\,d$, $j \in \mathbb{Z}$.

With `TicksAnchor = t0`, `TicksDistance = d`, these ticks are used for all coordinate axes.

With `XTicksAnchor = t0`, `XTicksDistance = d` etc., these ticks may be defined separately for each single coordinate axis.

When executing a plot command, per default a "reasonable" placing for tick marks on coordinate axes is automatically computed. Through this process tick marks may not come to lie on desired positions. The attributes `TicksAnchor` and `TicksDistance` allow to generate an alternative mesh of equidistant tick marks.

Note: The attributes `TicksAnchor`, `XTicksAnchor` etc. only have an effect when a positive distance $d > 0$ between major ticks marks is set explicitly via `TicksDistance = d`, `XTicksDistance = d` etc.

The ticks set by `TicksAnchor` and `TicksDistance` are "major" tick marks bearing labels. Depending on the value of `TicksBetween`, there may be additional "minor" ticks without labels between each pair of major tick marks.

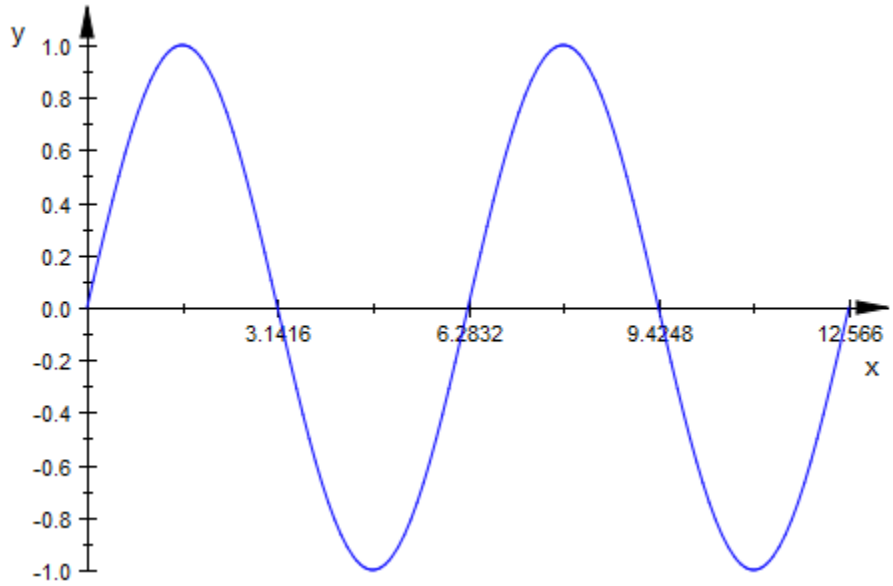
Additional tick marks at specific positions can be inserted with `TicksAt`.

Examples

Example 1

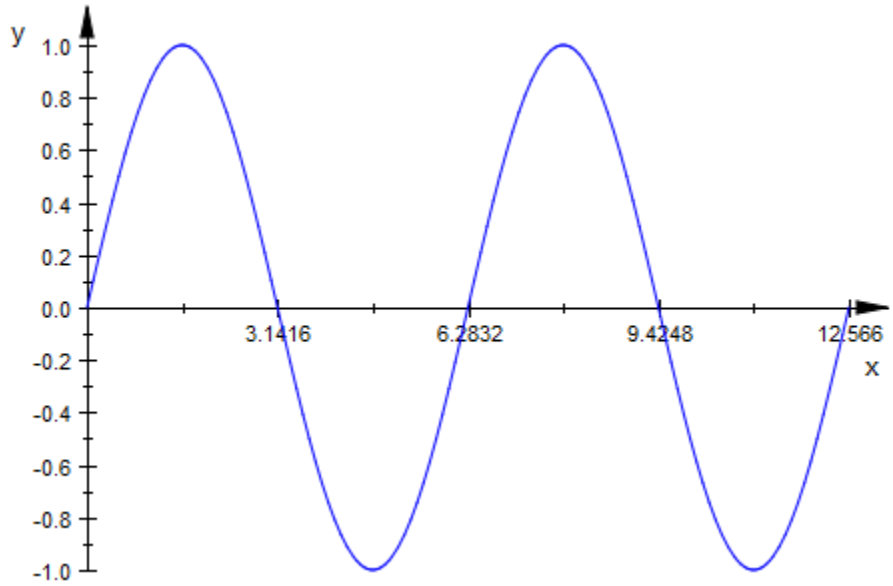
For the following plot of the sine function, the tick marks along the x-axis are chosen to match the period:

```
plot(plot::Function2d(sin(x), x = 0..4*PI),
     XTicksAnchor = 0, XTicksDistance = PI):
```



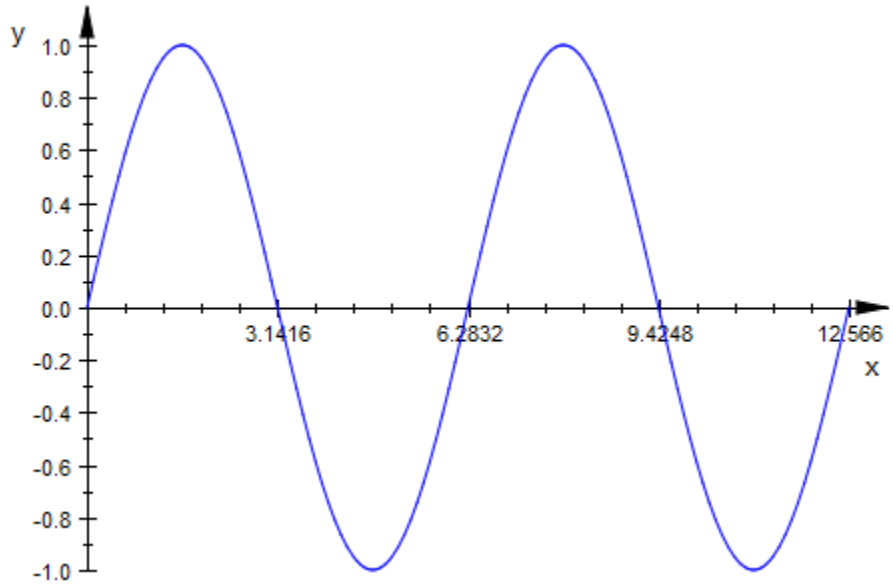
The ticks along the *y*-axis are re-defined with a distance of 0.2:

```
plot(plot::Function2d(sin(x), x = 0..4*PI),
     XTicksAnchor = 0, XTicksDistance = PI,
     YTicksAnchor = 0, YTicksDistance = 0.2):
```



We increase the number of "minor" ticks along the *x*-axis:

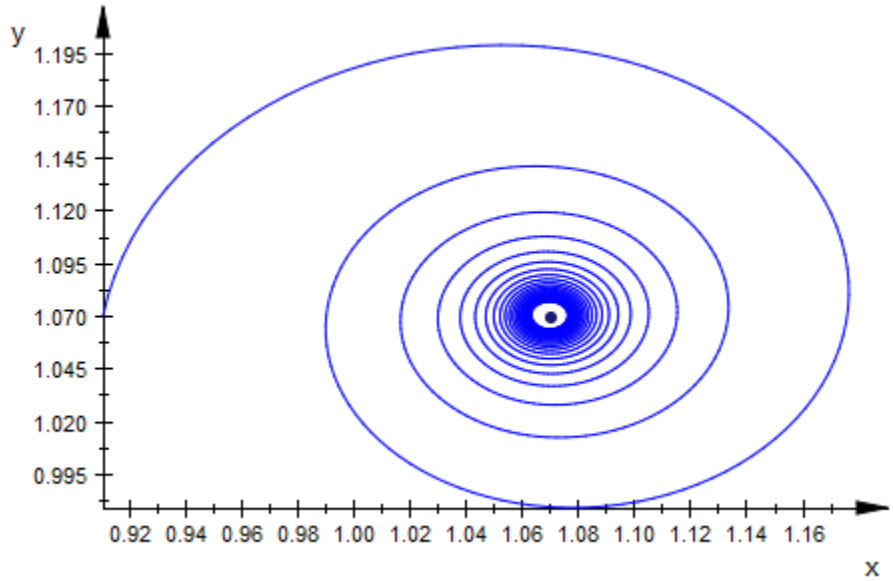
```
plot(plot::Function2d(sin(x), x = 0..4*PI),
     XTicksAnchor = 0, XTicksDistance = PI,
     XTicksBetween = 4,
     YTicksAnchor = 0, YTicksDistance = 0.2):
```



Example 2

We plot a hyperbolic spiral about the point `(1.07, 1.07)` which is not included in the automatic tick marks. We increase the number of ticks along the vertical axis and position the ticks relative to this point. Note that the tick marks along the horizontal axis miss the center of the spiral:

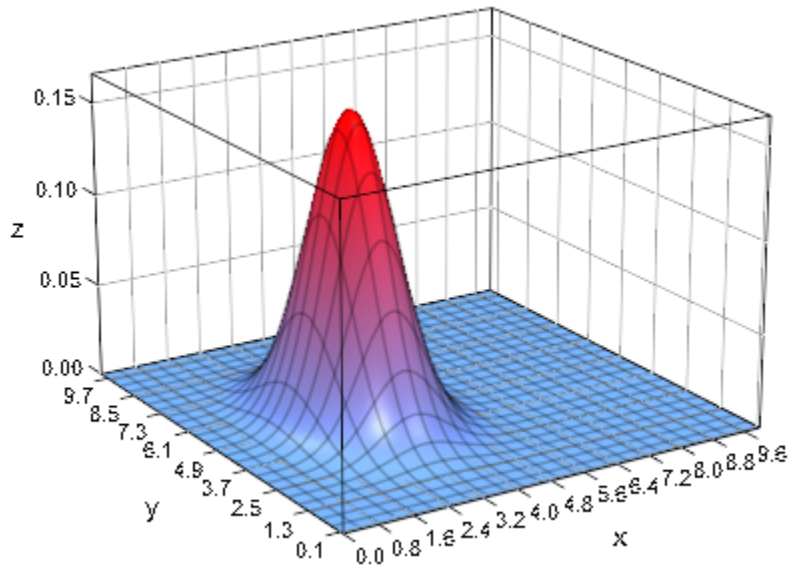
```
plot(plot::Point2d(1.07, 1.07),
     plot::Curve2d([1.07 - cos(t)/t, 1.07 + sin(t)/t],
                   t = 2*PI..50*PI, Submesh = 20),
     YTicksDistance = 0.025, YTicksAnchor = 1.07)
```



Example 3

We plot the two-dimensional normal distribution centered around the mean $(m_1, m_2) = (3.2, 4.9)$. This point is used as the anchor for the tick marks along the x-axis and the y-axis, respectively. Ticks are positioned at distances that are integer multiples of the standard deviations $(s_1, s_2) = (0.8, 1.2)$:

```
m1:= 3.2: s1 := 0.8:
m2:= 4.9: s2 := 1.2:
plot(plot::Function3d( stats::normalPDF(m1, s1^2)(x)
                      *stats::normalPDF(m2, s2^2)(y),
                      x = 0 .. 10, y = 0 .. 10,
                      Submesh = [3, 3]),
     XTicksAnchor = m1, XTicksDistance = s1,
     YTicksAnchor = m2, YTicksDistance = s2,
     TicksBetween = 0, GridVisible = TRUE):
```




```
delete m1, s1, m2, s2:
```

See Also

MuPAD Functions

TicksAnchor | TicksAt | TicksBetween | TicksLabelFont | TicksLabelStyle | TicksLabelsVisible | TicksLength | TicksNumber | TicksVisible

TicksLabelStyle, XTicksLabelStyle, YTicksLabelStyle, ZTicksLabelStyle

Display style of axes tick labels

Value Summary

`TicksLabelStyle`	Library wrapper for "{`XTicksLabelStyle`, `YTicksLabelStyle`}" (2D), "{`XTicksLabelStyle`, `YTicksLabelStyle`, `ZTicksLabelStyle`}" (3D)	See below
`XTicksLabelStyle`, `YTicksLabelStyle`, `ZTicksLabelStyle`	Inherited	`Diagonal`, `Horizontal`, `Shifted`, or `Vertical`

Graphics Primitives

Objects	Default Values
`plot::CoordinateSystem2d`	`TicksLabelStyle`, `XTicksLabelStyle`, `YTicksLabelStyle`: `Horizontal`
`plot::CoordinateSystem3d`	`TicksLabelStyle`, `XTicksLabelStyle`, `YTicksLabelStyle`, `ZTicksLabelStyle`: `Horizontal`

Description

`TicksLabelStyle` allows to modify the display style of the tick labels on all coordinate axes.

`XTicksLabelStyle` etc. allow to set the label styles separately for each single coordinate axis.

It may occur that tick labels overlap if too many tick marks along the coordinate axes are requested. The following styles for the tick labels are available to deal with this problem:

`Horizontal`: The labels are displayed in the usual horizontal reading order from left to right.

`Vertical`: The labels are tilted 90 degrees counter clockwise, i.e., they have to be read from bottom to top.

`Diagonal`: The labels are tilted 45 degrees counter clockwise.

`Shifted`: Each second label is shifted to avoid overlapping.

Note that also in 3D the orientation `Horizontal`, `Diagonal`, `Vertical` refers to the screen output irrespectively of the 3D orientation of the corresponding axis.

`TicksLabelStyle` sets the display style for the ticks labels along *all* coordinate axes.

With `XTicksLabelStyle` etc. the style may be set separately for each single axis.

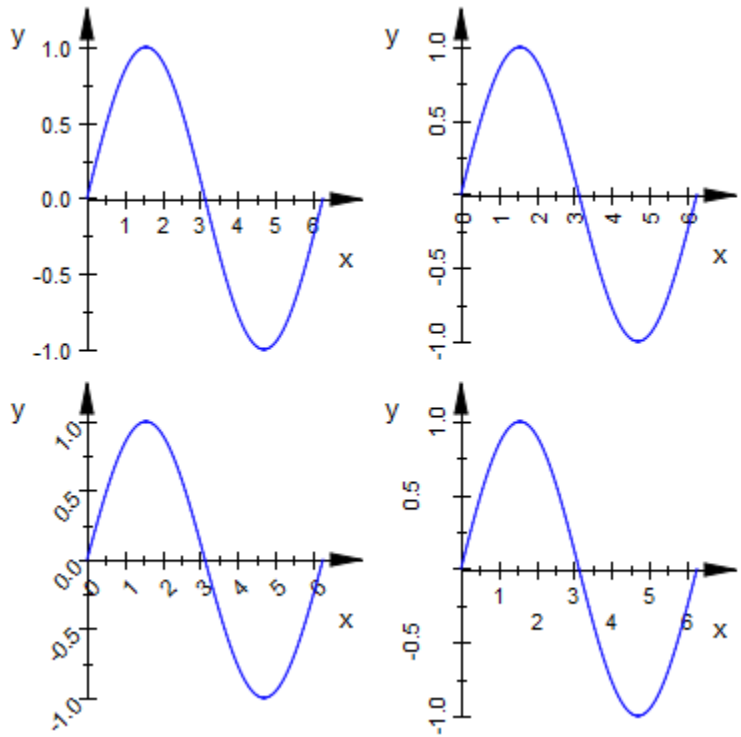
Independently from `TicksLabelStyle`, the titles of the axes are rendered horizontally. In 2D, the attribute `YAxisTitleOrientation` is available to tilt the title of the vertical axis by 90 degrees.

Examples

Example 1

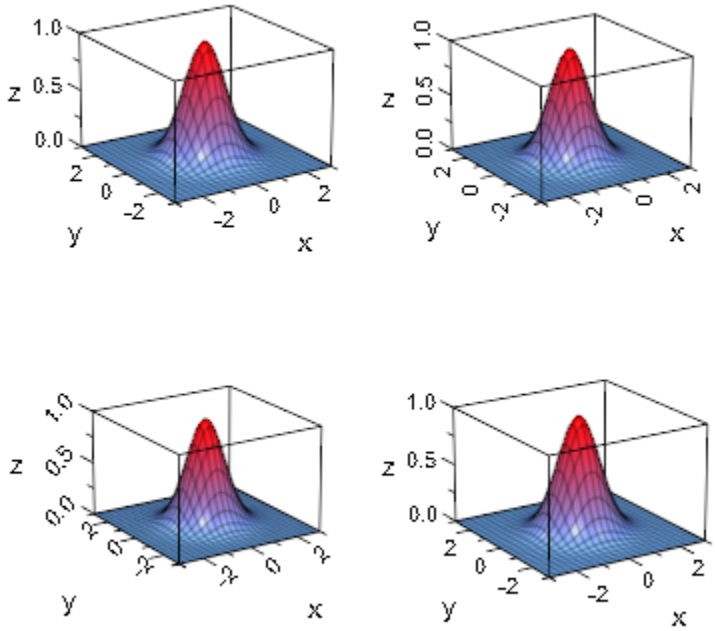
We demonstrate the styles for the ticks labels:

```
f := plot::Function2d(sin(x), x = 0 .. 2*PI):
S1 := plot::Scene2d(f, TicksLabelStyle = Horizontal):
S2 := plot::Scene2d(f, TicksLabelStyle = Vertical):
S3 := plot::Scene2d(f, TicksLabelStyle = Diagonal):
S4 := plot::Scene2d(f, TicksLabelStyle = Shifted):
plot(S1, S2, S3, S4, Height = 10*unit::cm,
     Width = 10*unit::cm):
```



Here is a corresponding picture in 3D:

```
f := plot::Function3d(exp(-x^2 - y^2), x = -3..3,
                      y = -3..3, Submesh = [2, 2]):
S1 := plot::Scene3d(f, TicksLabelStyle = Horizontal):
S2 := plot::Scene3d(f, TicksLabelStyle = Vertical):
S3 := plot::Scene3d(f, TicksLabelStyle = Diagonal):
S4 := plot::Scene3d(f, TicksLabelStyle = Shifted):
plot(S1, S2, S3, S4, Height = 10*unit::cm,
     Width = 10*unit::cm):
```

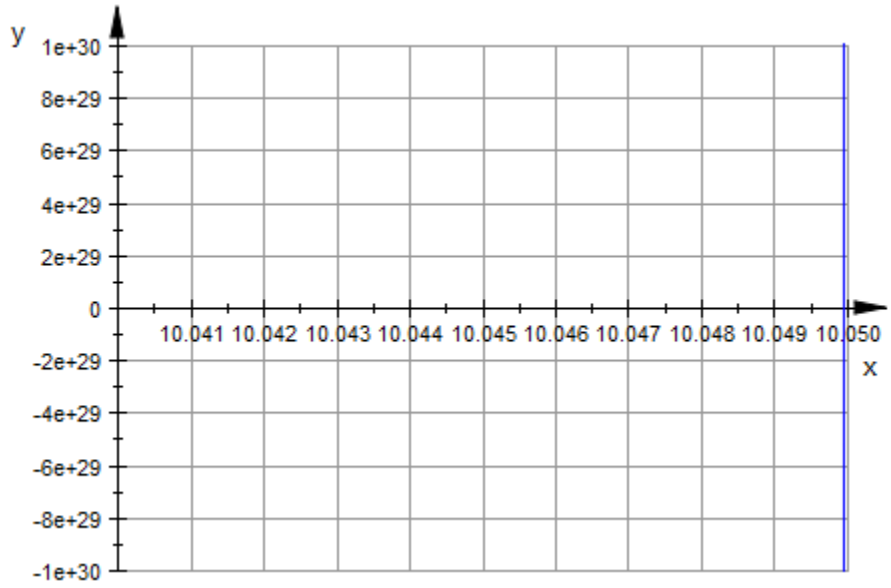



```
delete f, S1, S2, S3, S4:
```

Example 2

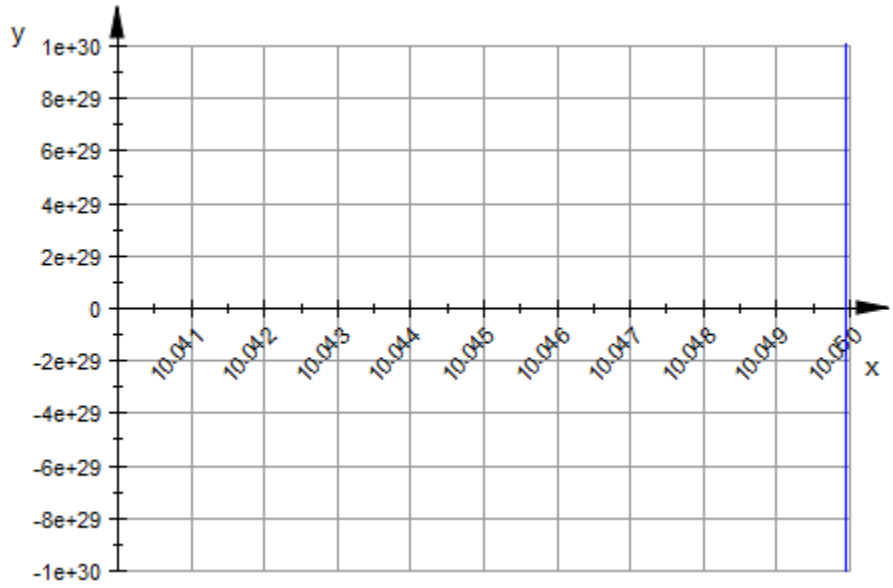
The tick labels along the *x*-axis nearly collide in the following plot:

```
f := plot::Function2d(exp(30*x)*sin(x*100*PI),
                      x = 10.04 .. 10.05):
plot(f, GridVisible = TRUE, XTicksNumber = High):
```



Tilting the labels yields a more tidy looking graphics:

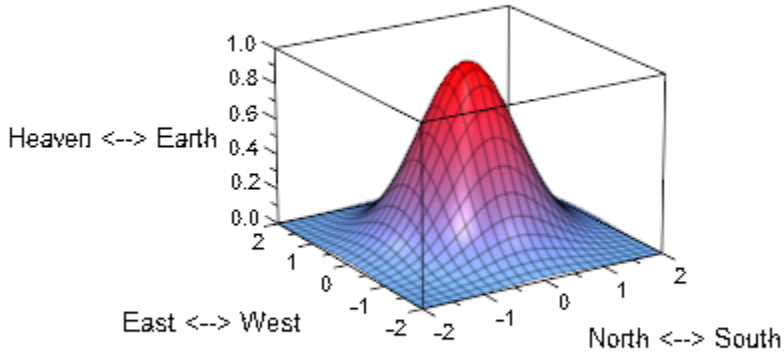
```
plot(f, GridVisible = TRUE, XTicksNumber = High,
     XTicksLabelStyle = Diagonal):
```



Example 3

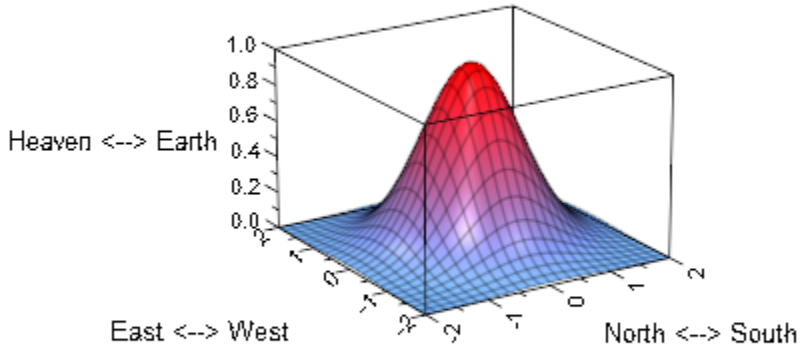
In the following graphics, there is not enough space to squeeze in the requested high number of ticks in the *x* and *y* direction:

```
plot(plot::Function3d(exp(-x^2 - y^2), x = -2..2, y = -2..2,
                      Submesh = [2, 2]),
     TicksNumber = High,
     AxesTitles = ["North <--> South",
                   "East <--> West",
                   "Heaven <--> Earth"]):
```



The tick labels fit with `Vertical` and `Diagonal` orientation:

```
plot(plot::Function3d(exp(-x^2 - y^2), x = -2..2, y = -2..2,
                      Submesh = [2, 2]),
     TicksNumber = High,
     XTicksLabelStyle = Vertical,
     YTicksLabelStyle = Diagonal,
     ZTicksLabelStyle = Horizontal,
     AxesTitles = ["North <--> South",
                   "East <--> West",
                   "Heaven <--> Earth"]):
```



See Also

MuPAD Functions

`TicksAnchor` | `TicksAt` | `TicksBetween` | `TicksDistance` | `TicksLabelFont` | `TicksLabelsVisible` | `TicksLength` | `TicksNumber` | `TicksVisible`

TicksLength

Length of axes tick marks

Value Summary

Inherited	Positive output size

Graphics Primitives

Objects	TicksLength Default Values
`plot::CoordinateSystem2d`, `plot::CoordinateSystem3d`	`2`

Description

The tick marks along the coordinate axes consist of “major” tick marks bearing labels and of “minor” tick marks without labels.

`TicksLength` sets the length for the major tick marks on all coordinate axes. The length of minor tick marks (cf. `TicksBetween`) is half of `TicksLength` .

The value should be specified as an absolute physical length including a length unit such as `TicksLength = 2.5*unit::mm`. Numbers without a physical unit give the length in mm.

`TicksLength` sets the length of automatic tick marks (cf. `TicksNumber`) as well of special tick marks set via `TicksAt`.

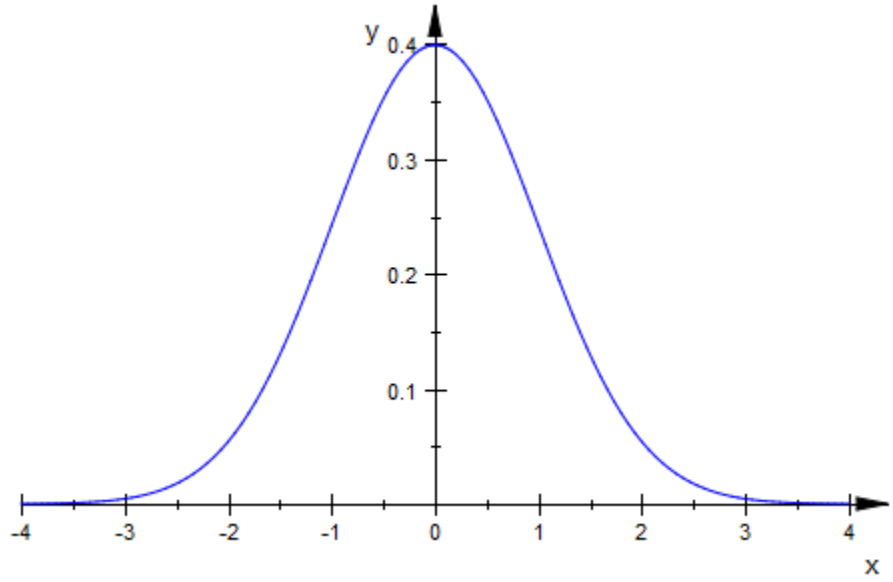
It is not possible to change the length of tick marks on any single axis alone.

Examples

Example 1

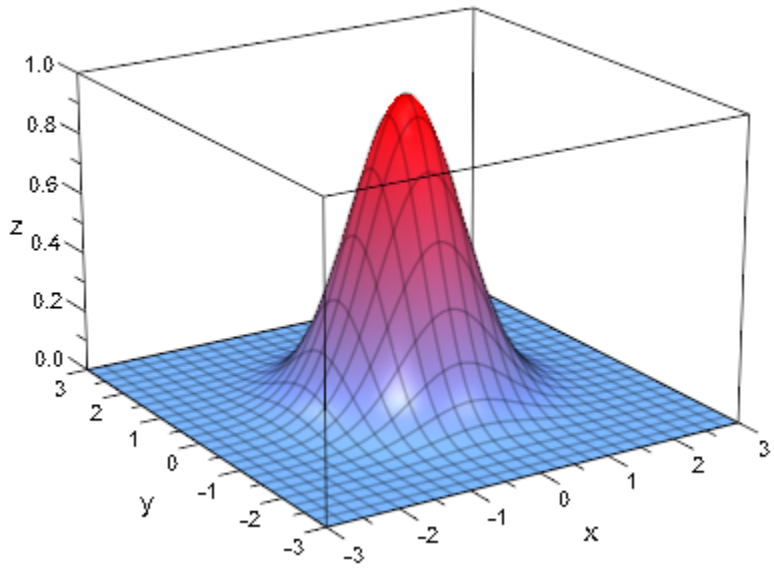
We plot the density of the standard normal distribution. Compared to the default length of 2 mm, the ticks length is increased by 50%:

```
plot(plot::Function2d(stats::normalPDF(0, 1)(x), x = -4..4),
     TicksLength = 3*unit::mm):
```



A corresponding plot in 3D:

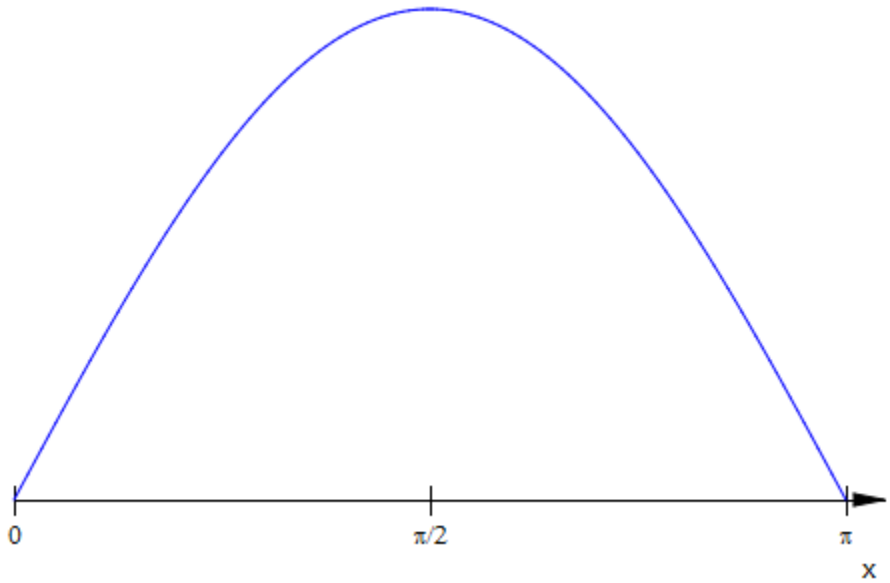
```
plot(plot::Function3d(exp(-x^2 - y^2), x = -3..3, y = -3..3,
                      Submesh = [2, 2]),
     TicksLength = 3*unit::mm)
```



Example 2

In the following plot of the sine function, we switch the automatic tick marks along the *x*-axis off via `XTicksNumber = None`. Some extra ticks are set via `XTicksAt`:

```
plot(plot::Function2d(sin(x), x = 0 .. PI),
     XTicksAt = [0 = "0", PI/2 = "p/2", PI = "p"],
     XTicksNumber = None, TicksLength = 4*unit::mm,
     TicksLabelFont = ["Symbol"], YAxisVisible = FALSE)
```



See Also

MuPAD Functions

`TicksAnchor` | `TicksAt` | `TicksBetween` | `TicksDistance` | `TicksLabelFont` | `TicksLabelStyle` | `TicksLabelsVisible` | `TicksNumber` | `TicksVisible`

TicksNumber, XTicksNumber, YTicksNumber, ZTicksNumber

Number of axes tick marks

Value Summary

`TicksNumber`	Library wrapper for "{`XTicksNumber`, `YTicksNumber`}" (2D), "{`XTicksNumber`, `YTicksNumber`, `ZTicksNumber`}" (3D)	See below
`XTicksNumber`, `YTicksNumber`, `ZTicksNumber`	Inherited	`High`, `Low`, `None`, or `Normal`

Graphics Primitives

Objects	Default Values
`plot::CoordinateSystem2d`	`TicksNumber`, `XTicksNumber`, `YTicksNumber`: `Normal`
`plot::CoordinateSystem3d`	`TicksNumber`, `XTicksNumber`, `YTicksNumber`, `ZTicksNumber`: `Normal`

Description

`TicksNumber` directs the internal routine that chooses tick marks along coordinate axes to produce no, few, or many ticks.

With `XTicksNumber` etc., the number of tick marks can be controlled separately for each single axis.

The tick marks along the coordinate axes consist of "major" tick marks bearing labels and of "minor" tick marks without labels.

The attributes `TicksNumber`, `XTicksNumber` etc. only refer to the labeled "major" tick marks. The "minor" tick marks are governed by the attribute `TicksBetween`.

Automatically generated equidistant tick marks are displayed along the coordinate axes, unless the user specifies the ticks explicitly via the attributes `TicksAnchor` and `TicksDistance`.

`TicksNumber` provides a hint for the automatic computation process, how many tick marks are to be displayed. The possible values are `None`, `Low`, `Normal`, and `Many`.

With `XTicksNumber` etc., ticks numbers may be controlled separately for each single axis.

If equidistant tick marks are set explicitly via `TicksAnchor` and `TicksDistance`, the attributes `TicksNumber`, `XTicksNumber` etc. are ignored.

There is no influence on special tick marks set via `TicksAt`, `XTicksAt` etc. either.

With `TicksNumber` = `None`, `XTicksNumber` = `None` etc., no automatically generated tick marks are displayed.

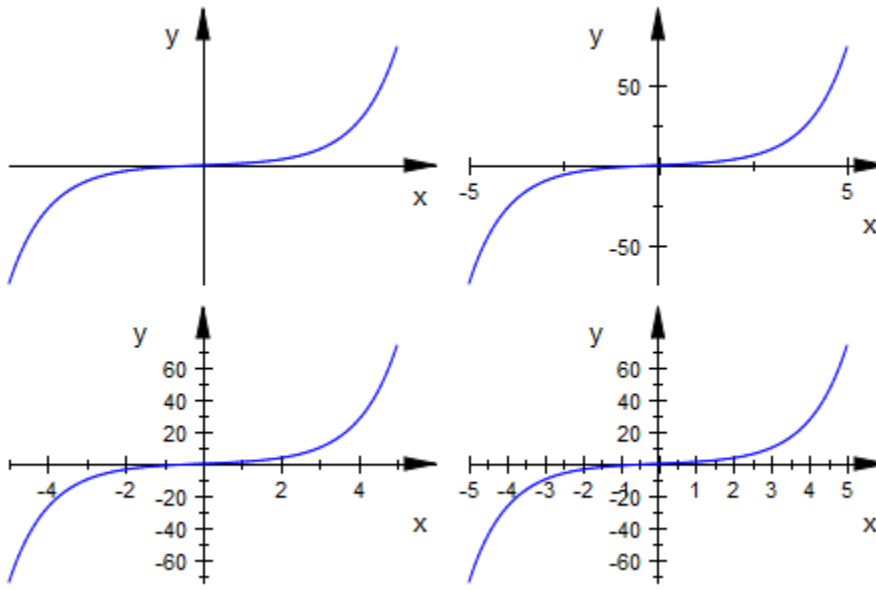
Tick marks may also be suppressed via `TicksVisible` = `FALSE`, `XTicksVisible` = `FALSE` etc. However, in contrast to `TicksNumber` = `None`, `XTicksNumber` = `None` etc., this also suppresses equidistant tick marks set explicitly via `TicksAnchor`, `TicksDistance` as well as special tick marks set via `TicksAt`, `XTicksAt` etc.

Examples

Example 1

We demonstrate the effect of various `TicksNumber` values:

```
f := plot::Function2d(sinh(x), x = -5 .. 5):
S1 := plot::Scene2d(f, TicksNumber = None):
S2 := plot::Scene2d(f, TicksNumber = Low):
S3 := plot::Scene2d(f, TicksNumber = Normal):
S4 := plot::Scene2d(f, TicksNumber = High):
plot(S1, S2, S3, S4):
```

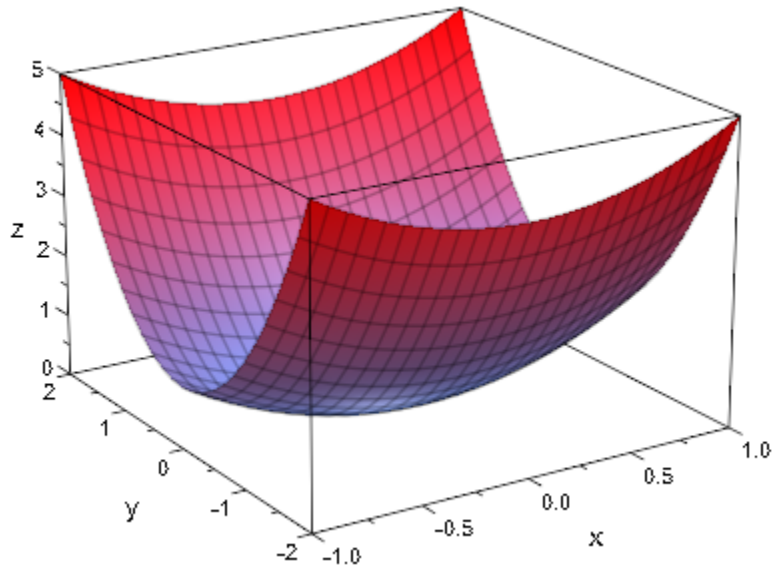



```
delete f, S1, S2, S3, S4:
```

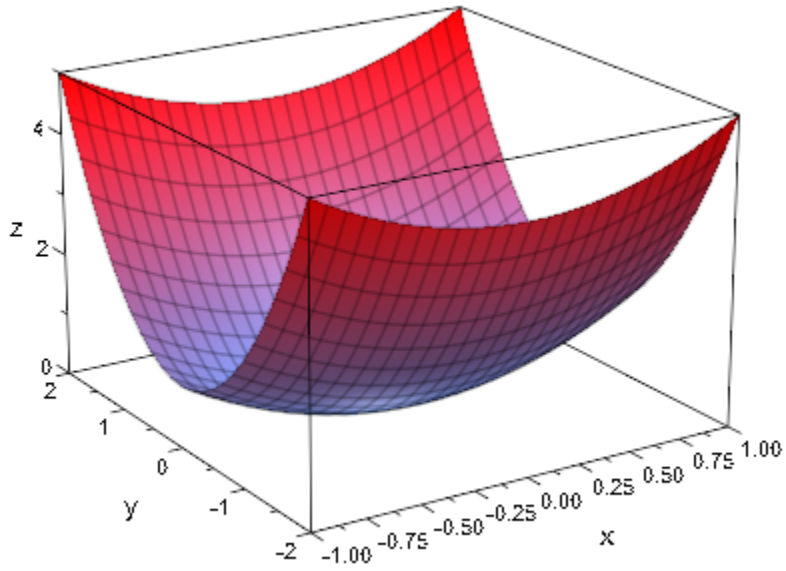
Example 2

We demonstrate the effect of various `TicksNumber` values in a 3D plot:

```
s := plot::Function3d(x^2 + y^2, x = -1..1, y = -2..2):
plot(s):
```

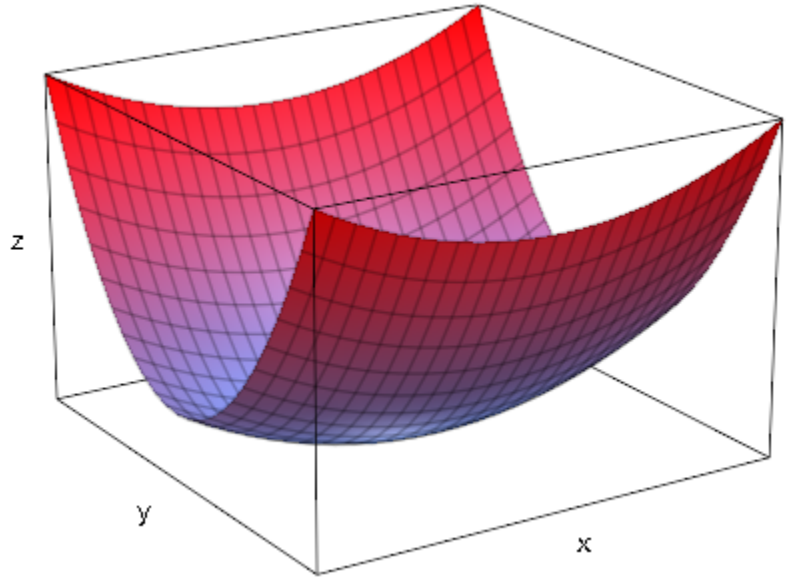



```
plot(s, XTicksNumber = High, YTicksNumber = Normal,
     ZTicksNumber = Low):
```



All tick marks are suppressed:

```
plot(s, TicksNumber = None)
```




```
delete s:
```

See Also

MuPAD Functions

TicksAnchor | TicksAt | TicksBetween | TicksDistance | TicksLabelFont | TicksLabelStyle | TicksLabelsVisible | TicksLength | TicksVisible

TicksVisible, XTicksVisible, YTicksVisible, ZTicksVisible

Display axes tick marks?

Value Summary

`TicksVisible`	Library wrapper for "{`XTicksVisible`, `YTicksVisible`}" (2D), "{`XTicksVisible`, `YTicksVisible`, `ZTicksVisible`}" (3D)	`TRUE`, `FALSE`, or list of 2 or 3 of these, depending on the dimension
`XTicksVisible`, `YTicksVisible`, `ZTicksVisible`	Inherited	`FALSE`, or `TRUE`

Graphics Primitives

Objects	Default Values
`plot::CoordinateSystem2d`	`TicksVisible`, `XTicksVisible`, `YTicksVisible`: `TRUE`
`plot::CoordinateSystem3d`	`TicksVisible`, `XTicksVisible`, `YTicksVisible`, `ZTicksVisible`: `TRUE`

Description

`TicksVisible = TRUE` versus `TicksVisible = FALSE` switches tick marks along all coordinate axes on or off.

With `XTicksVisible = TRUE/FALSE` etc., the tick marks can be switched on or off separately for each single axis.

With `TicksVisible = FALSE`, the tick marks along all coordinate axes are switched off. The labels of the tick marks, however, remain visible.

`TicksVisible` etc. refers to automatically generated tick marks (cf. `TicksNumber`), to equidistant tick marks that are requested explicitly via `TicksAnchor`, `TicksDistance` as well as to special tick marks set via `TicksAt`.

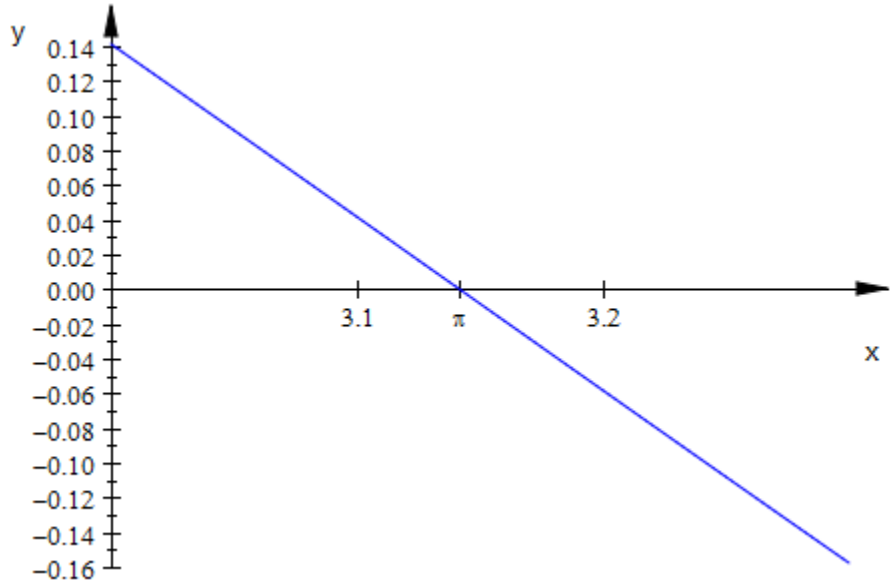
Ticks can also be suppressed via `TicksNumber = None`, `XTicksNumber = None` etc. In contrast to `TicksVisible = FALSE`, however, this affects only the automatically generated ticks and their labels. Ticks set by `TicksAnchor`, `TicksDistance`, `TicksAt` are not affected.

Examples

Example 1

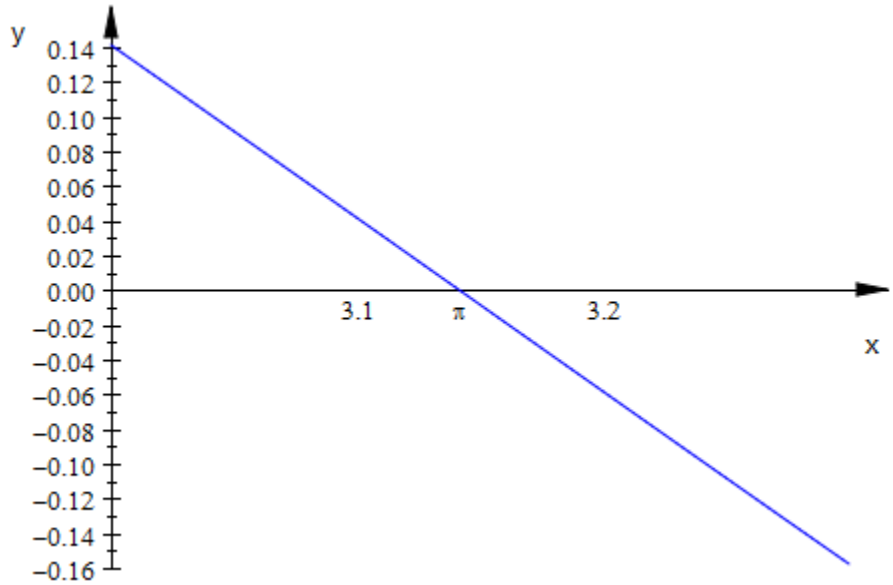
Visualizing that the sine function is nearly linear near its zeroes, we suppress the automatic tick marks along the `x`-axis via `XTicksNumber = None`. Three special ticks are set via `XTicksAt`:

```
f := plot::Function2d(sin(x), x = 3.0 .. 3.3):
plot(f, XTicksNumber = None, XTicksAt = [3.1, PI = "p", 3.2],
     TicksLabelFont = ["Symbol"])
```



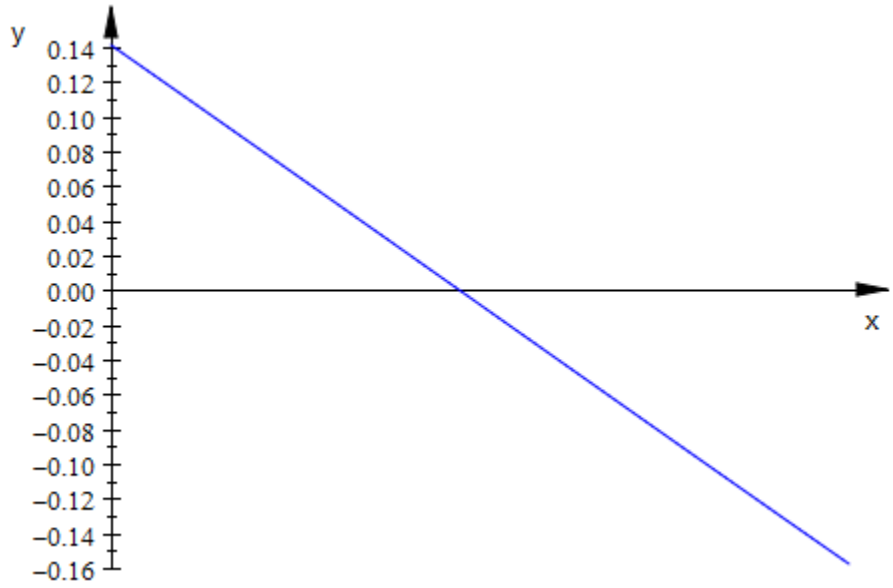
The tick marks along the *x*-axis are switched off:

```
plot(f, XTicksNumber = None, XTicksAt = [3.1, PI = "p", 3.2],
     TicksLabelFont = ["Symbol"], XTicksVisible = FALSE)
```



The labels of the ticks are switched off, too:

```
plot(f, XTicksNumber = None, XTicksAt = [3.1, PI = "p", 3.2],
     TicksLabelFont = ["Symbol"], XTicksVisible = FALSE,
     XTicksLabelsVisible = FALSE)
```




```
delete f:
```

See Also

MuPAD Functions

`TicksAnchor` | `TicksAt` | `TicksBetween` | `TicksDistance` | `TicksLabelFont` | `TicksLabelStyle` | `TicksLabelsVisible` | `TicksLength` | `TicksNumber`

TicksLabelsVisible, XTicksLabelsVisible, YTicksLabelsVisible, ZTicksLabelsVisible

Display axes tick labels?

Value Summary

`TicksLabelsVisible`	Library wrapper for "{`XTicksLabelsVisible`, `YTicksLabelsVisible`}" (2D), "{`XTicksLabelsVisible`, `YTicksLabelsVisible`, `ZTicksLabelsVisible`}" (3D)	See below
`XTicksLabelsVisible`, `YTicksLabelsVisible`, `ZTicksLabelsVisible`	Inherited	`FALSE`, or `TRUE`

Graphics Primitives

Objects	Default Values
`plot::CoordinateSystem2d`	`TicksLabelsVisible`, `XTicksLabelsVisible`, `YTicksLabelsVisible`: `TRUE`
`plot::CoordinateSystem3d`	`TicksLabelsVisible`, `XTicksLabelsVisible`, `YTicksLabelsVisible`, `ZTicksLabelsVisible`: `TRUE`

Description

`TicksLabelsVisible` = `TRUE` versus `TicksLabelsVisible` = `FALSE` switches the labeling of the tick marks along all coordinate axes on or off.

With `TicksLabelsVisible` = `FALSE`, the labeling of the tick marks along all coordinate axes is switched off. The tick marks themselves, however, remain visible. They are switched off via `TicksVisible` = `FALSE`.

With `XTicksLabelsVisible` = `TRUE`/`FALSE` etc., the tick labeling can be switched on or off separately for each single axis.

`TicksLabelsVisible`, `XTicksLabelsVisible` etc. refer to automatically generated tick marks (cf. `TicksNumber`), to equidistant tick marks that are requested explicitly via `TicksAnchor`, `TicksDistance` as well as to special tick marks set via `TicksAt`.

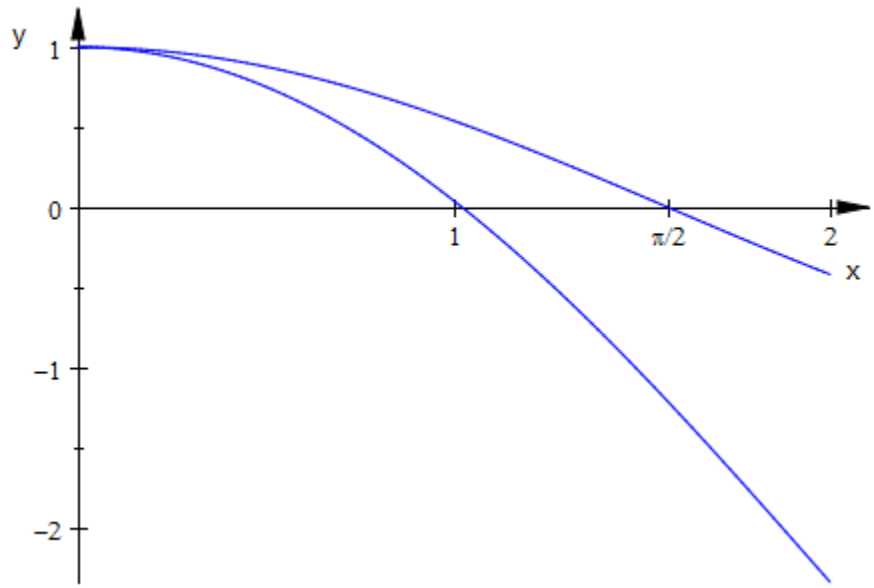
Ticks can also be suppressed via `TicksNumber` = `None`, `XTicksNumber` = `None` etc. In contrast to `TicksLabelsVisible` = `FALSE`, however, this affects only the automatically generated ticks and their labels. Ticks set by `TicksAnchor`, `TicksDistance`, `TicksAt` are not affected.

Examples

Example 1

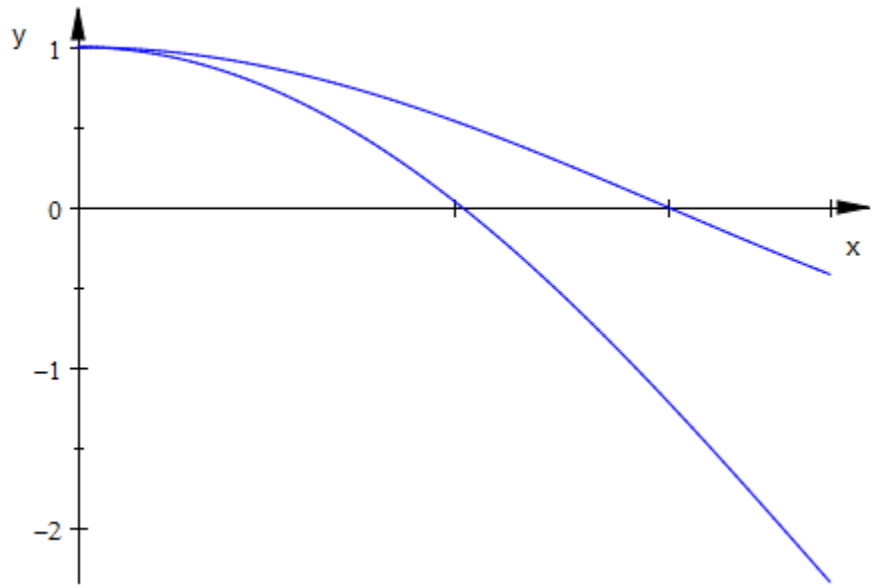
We approximate the cosine function by a fourth order polynomial (a Taylor polynomial around the expansion point 0). The automatic tick marks along the *x*-axis are suppressed via `XTicksNumber` = `None`. Some special tick marks including the zero of the cosine function at $x = \frac{\pi}{2}$ are inserted via `XTicksAt`:

```
f1 := plot::Function2d(cos(x), x = 0..2):
f2 := plot::Function2d(1 - x^2 + x^4/4!, x = 0..2,
                       LineColor = RGB::Blue):
plot(f1, f2, XTicksNumber = None,
     XTicksAt = [1, PI/2 = "p/2", 2],
     TicksLabelFont = ["Symbol"])
```



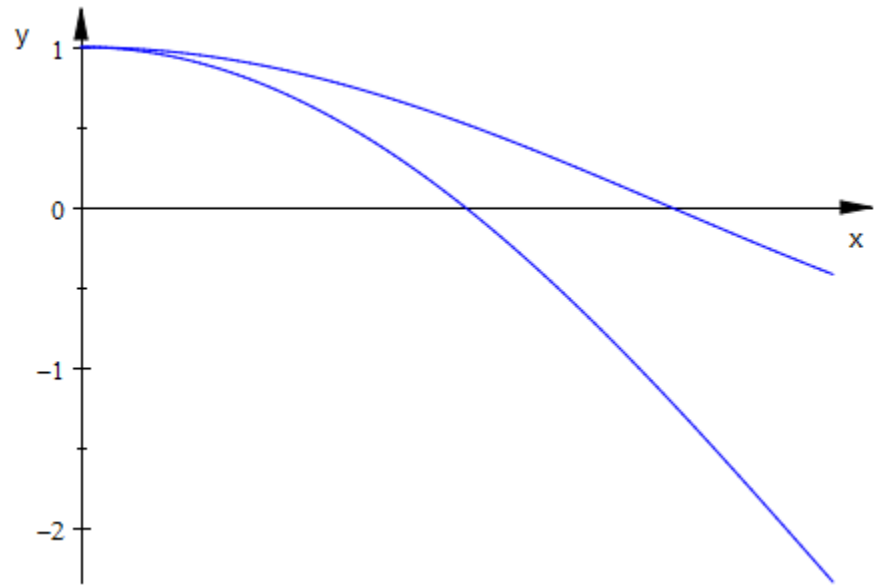
The labeling of the tick marks along the *x*-axis is switched off:

```
plot(f1, f2, XTicksNumber = None,
     XTicksAt = [1, PI/2 = "p/2", 2],
     TicksLabelFont = ["Symbol"],
     XTicksLabelsVisible = FALSE)
```



The ticks themselves are switched off, too:

```
plot(f1, f2, XTicksNumber = None,
     XTicksAt = [1, PI/2 = "p/2", 2],
     TicksLabelFont = ["Symbol"], XTicksLabelsVisible = FALSE,
     XTicksVisible = FALSE)
```




```
delete f1, f2:
```

See Also

MuPAD Functions

TicksAnchor | TicksAt | TicksBetween | TicksDistance | TicksLabelFont | TicksLabelStyle | TicksLength | TicksNumber | TicksVisible

GridInFront

Coordinate grid in front of or behind graphical objects?

Value Summary

Inherited	FALSE, or TRUE

Graphics Primitives

Objects	GridInFront Default Values
plot::CoordinateSystem2d	FALSE

Description

GridInFront = TRUE versus GridInFront = FALSE places 2D coordinate lines in front of or behind the graphical objects in the scene.

Setting GridVisible = TRUE, SubgridVisible = TRUE, one can display a coordinate grid extending the tick marks on the coordinate axes. See the help page of GridVisible for further information.

By default, the lines of the coordinate grid are plotted behind the graphical objects in a scene. Consequently, the objects may cover the coordinate grid. If only line objects and points are present in a 2D scene, this is desirable in most cases.

However, if there are filled areas such as filled polygons in the scene, the view to the coordinate grid may be totally blocked. In such a situation, you may want to draw the grid lines in front of the objects to guarantee visibility of the coordinate grid.

Although the default setting is GridInFront = FALSE, some objects which create filled areas send GridInFront = TRUE as a "hint" (see the section Primitives Requesting Special Scene Attributes: "Hints" of this documentation).

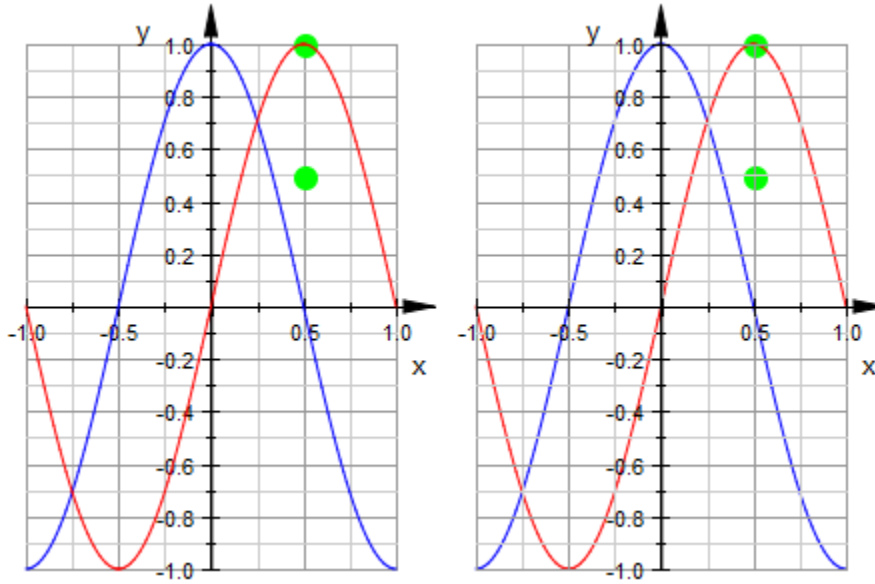
This attribute is available only in 2D.

Examples

Example 1

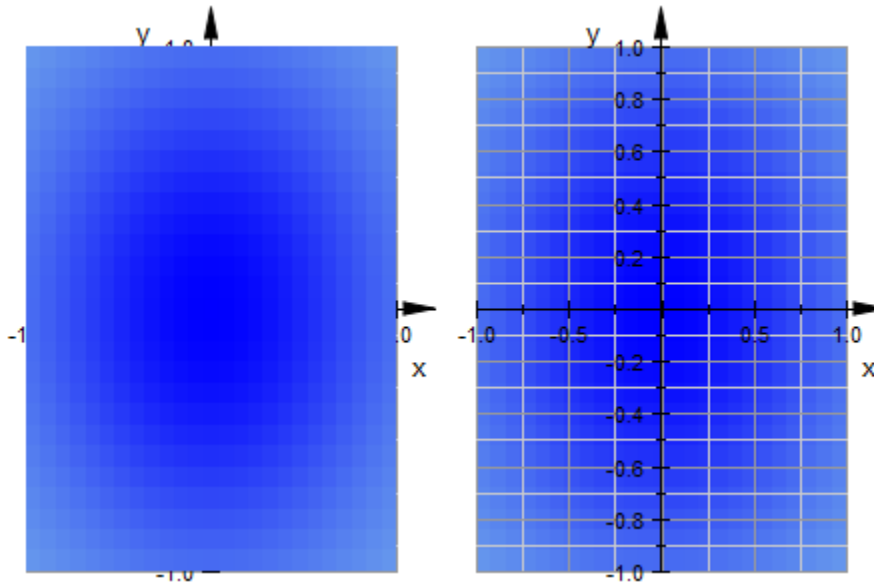
It is usually desirable to let line objects and points cover the coordinate grid:

```
p1 := plot::Point2d(0.5, 0.5, PointSize = 3*unit::mm,
                    Color = RGB::Green):
p2 := plot::Point2d(0.5, 1.0, PointSize = 3*unit::mm,
                    Color = RGB::Green):
f1 := plot::Function2d(cos(x*PI), x = -1 .. 1,
                       Color = RGB::Blue):
f2 := plot::Function2d(sin(x*PI), x = -1 .. 1, Color = RGB::Red):
plot(plot::Scene2d(p1, p2, f1, f2,
                   AxesInFront = FALSE, GridInFront = FALSE),
     plot::Scene2d(p1, p2, f1, f2,
                   AxesInFront = TRUE, GridInFront = TRUE),
     GridVisible = TRUE, SubgridVisible = TRUE):
```



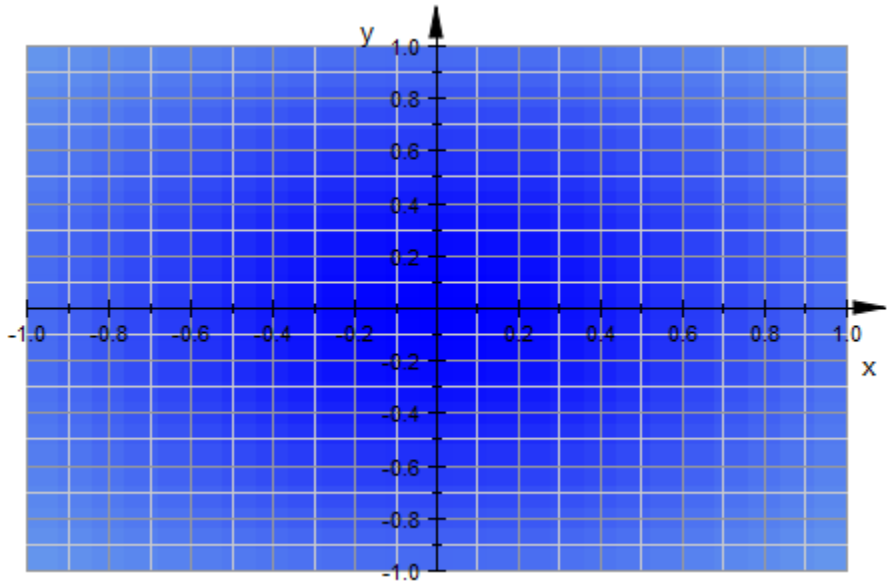
However, you probably want to have the coordinate grid visible in front of the following density plot:

```
d := plot::Density(exp(-x^2 - y^2), x = -1..1, y = -1 ..1,
                   FillColor = RGB::Blue):
plot(
 plot::Scene2d(d, AxesInFront = FALSE, GridInFront = FALSE),
 plot::Scene2d(d, AxesInFront = TRUE, GridInFront = TRUE),
 GridVisible = TRUE, SubgridVisible = TRUE,
 Layout = Horizontal
):
```



Note that density objects of type `plot::Density` automatically send the "hint" `GridInFront = TRUE`, so there is no need to set this attribute explicitly:

```
plot(d, GridVisible = TRUE, SubgridVisible = TRUE):
```




```
delete p1, p2, f1, f2, d:
```

See Also

MuPAD Functions

`GridLineColor` | `GridLineStyle` | `GridLineWidth` | `GridVisible`

GridLineColor, SubgridLineColor

Line color of the coordinate grid

Value Summary

`GridLineColor`, `SubgridLineColor`	Inherited	Color

Graphics Primitives

Objects	Default Values

`plot::CoordinateSystem2d`, `plot::CoordinateSystem3d`	`GridLineColor`: `RGB::Grey60`
	
	`SubgridLineColor`: `RGB::Grey80`

Description

`GridLineColor`, `SubgridLineColor` govern the color of coordinate grid and subgrid lines extending the tick marks on coordinate axes.

Setting `GridVisible` = `TRUE`, `SubgridVisible` = `TRUE`, one can display a coordinate grid extending the tick marks on the coordinate axes. See the help page of `GridVisible` for further information.

`GridLineColor`, `SubgridLineColor` set the RGB color for the coordinate grid and subgrid lines.

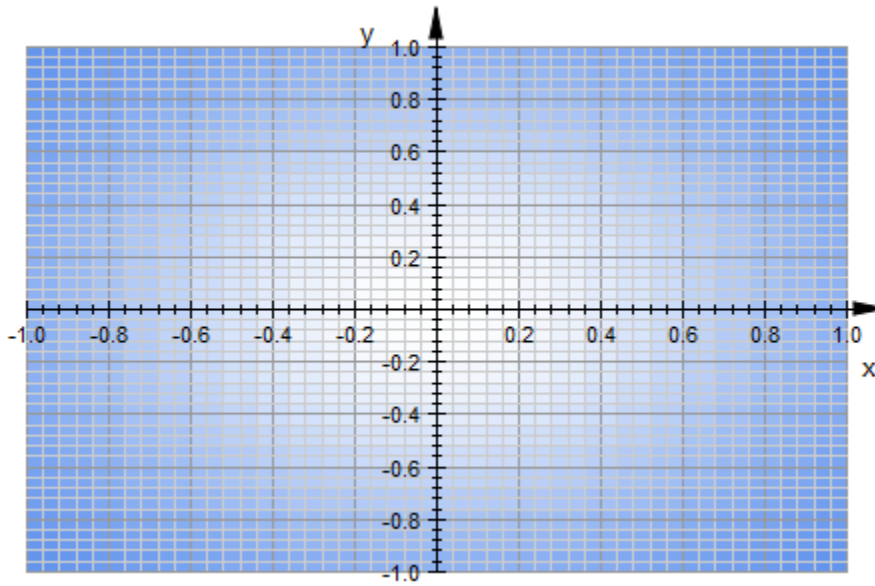
The color of the coordinate grid lines cannot be specified separately for the single coordinate directions.

Examples

Example 1

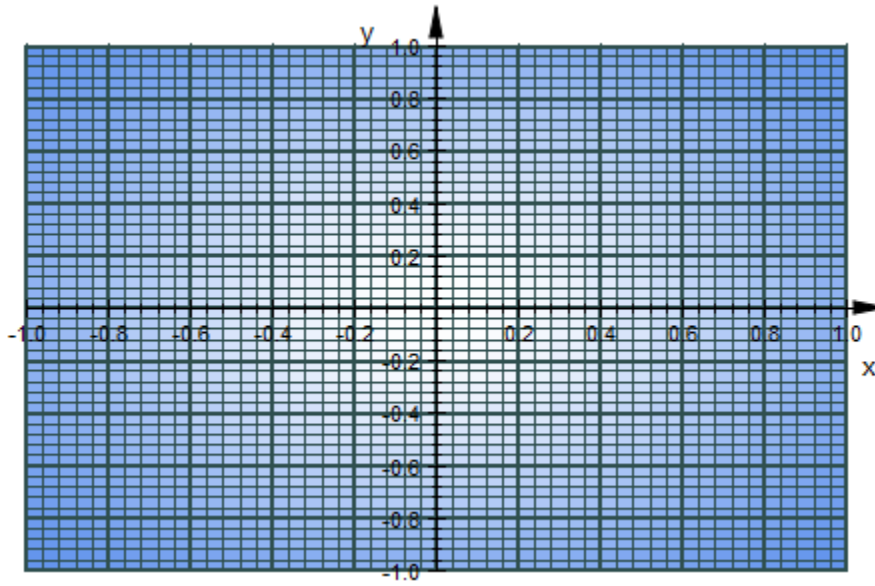
The usual grey lines of the coordinate grid are not appropriate for the following density graphics:

```
d := plot::Density(exp(-x^2 - y^2), x = -1..1, y = -1 ..1,
                   FillColor = RGB::White):
plot(d, TicksNumber = Normal, TicksBetween = 4,
     GridVisible = TRUE, SubgridVisible = TRUE)
```



We change the grid color to a darker grey:

```
plot(d, TicksNumber = Normal, TicksBetween = 4,
     GridVisible = TRUE, SubgridVisible = TRUE,
     GridLineColor = RGB::SlateGreyDark,
     SubgridLineColor = RGB::SlateGreyDark,
     GridLineWidth = 0.5*unit::mm)
```




```
delete d:
```

See Also

MuPAD Functions

`GridInFront` | `GridLineStyle` | `GridLineWidth` | `GridVisible`

GridLineStyle, SubgridLineStyle

Line style of the coordinate grid

Value Summary

`GridLineStyle`, `SubgridLineStyle`	Inherited	`Dashed`, `Dotted`, or `Solid`

Graphics Primitives

Objects	Default Values
`plot::CoordinateSystem2d`, `plot::CoordinateSystem3d`	`GridLineStyle`, `SubgridLineStyle`: `Solid`

Description

`GridLineStyle`, `SubgridLineStyle` govern the style of the coordinate grid lines and subgrid lines extending the tick marks on coordinate axes.

Setting `GridVisible` = `TRUE`, `SubgridVisible` = `TRUE`, one can display a coordinate grid extending the tick marks on the coordinate axes. See the help page of `GridVisible` for further information.

Styles for coordinate grid and subgrid lines can be either `Solid`, `Dashed`, or `Dotted`.

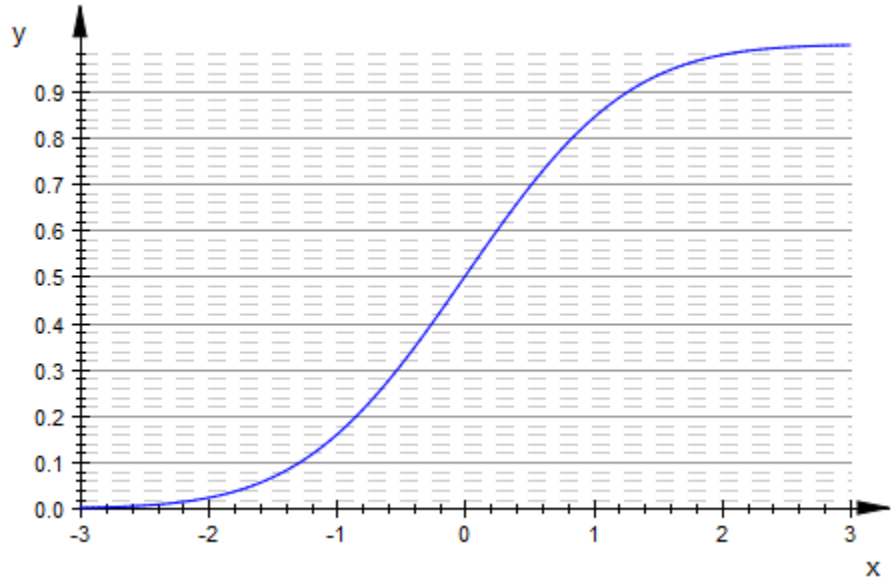
The line style of the coordinate grid cannot be specified separately for the single coordinate directions.

Examples

Example 1

We use horizontal coordinate lines to visualize quantiles for the normal distribution:

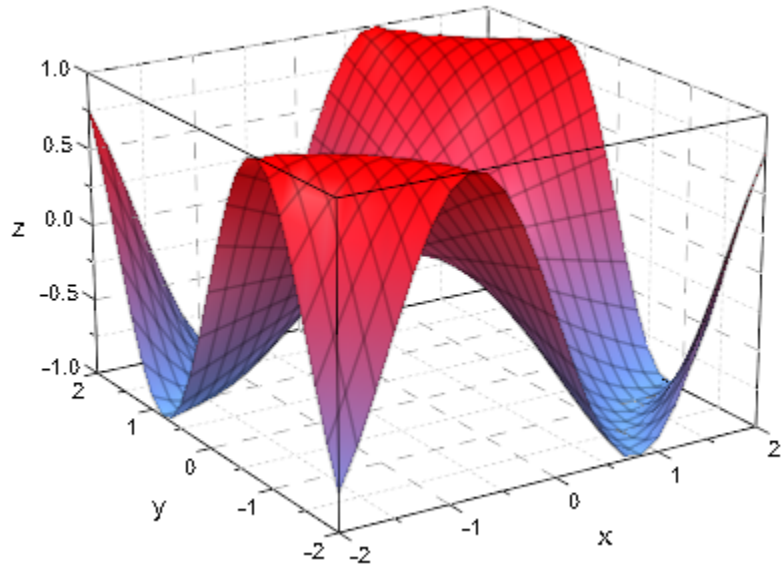
```
plot(plot::Function2d(stats::normalCDF(0, 1)(x), x = -3..3),
     Axes = Frame, TicksBetween = 4,
     YGridVisible = TRUE, YSubgridVisible = TRUE,
     GridLineStyle = Solid, SubgridLineStyle = Dashed):
```



Example 2

Here is an example of a function graph in 3D with different style settings for the coordinate grid and subgrid:

```
plot(plot::Function3d(sin(x*y), x = -2..2, y = -2..2),
     GridVisible = TRUE, SubgridVisible = TRUE,
     GridLineStyle = Dashed, SubgridLineStyle = Dotted):
```



See Also

MuPAD Functions

`GridInFront` | `GridLineColor` | `GridLineWidth` | `GridVisible`

GridLineWidth, SubgridLineWidth

Width of coordinate grid lines

Value Summary

`GridLineWidth`, `SubgridLineWidth`	Inherited	Positive output size

Graphics Primitives

Objects	Default Values
`plot::CoordinateSystem2d`, `plot::CoordinateSystem3d`	`GridLineWidth`, `SubgridLineWidth`: `0.1`

Description

`GridLineWidth`, `SubgridLineWidth` govern the width of coordinate grid lines and subgrid lines extending the tick marks on coordinate axes.

Setting `GridVisible` = `TRUE`, `SubgridVisible` = `TRUE`, one can display a coordinate grid extending the tick marks on the coordinate axes. See the help page of `GridVisible` for further information.

`GridLineWidth`, `SubgridLineWidth` set the linewidth for the coordinate grid and the subgrid, respectively. The values should be specified as absolute physical lengths including a length unit such as `GridLineWidth = 0.5*unit::mm`. Numbers without a physical unit give the size in mm.

`GridLinesWidth` and `SubgridLinesWidth` set a common line width for the grid lines in all coordinate directions.

`XGridLinesWidth` and `XSubgridLinesWidth` set the line width only for the grid lines extending the axes tick marks on the *x*-axis.

`YGridLinesWidth` etc. work correspondingly for the other coordinate directions.

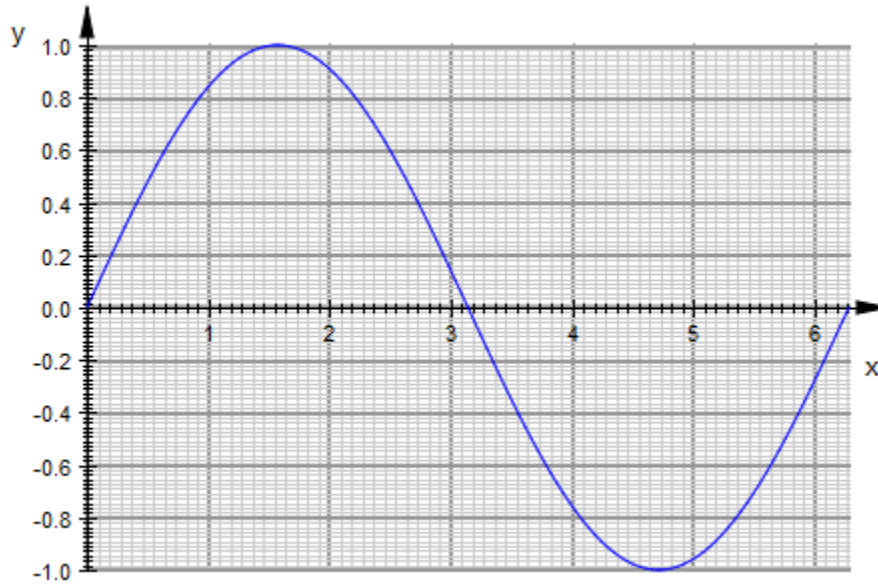
Note that the graphics cannot always react to small changes of the line width because of the discretization into pixels.

Examples

Example 1

We use the coordinate lines to plot the sine function on "lined paper". Because of the rather high number of subgrid lines set by `TicksBetween = 10`, we use extra fine lines for the subgrid:

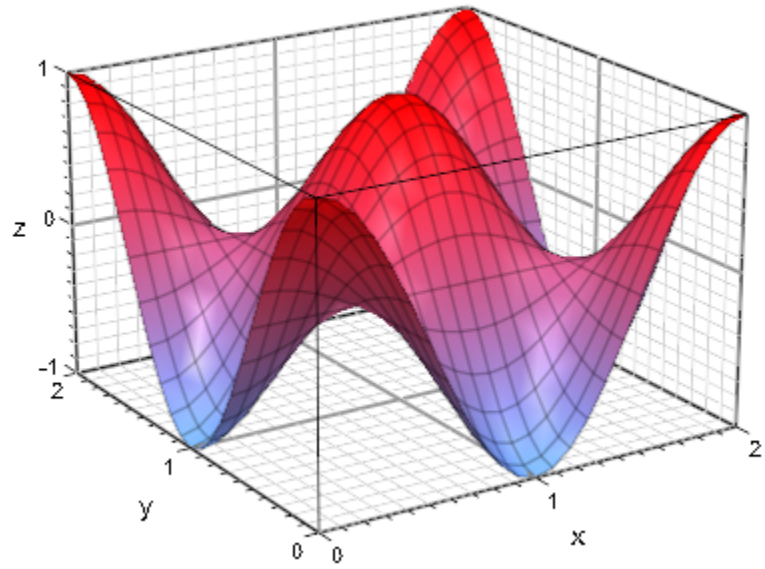
```
plot(plot::Function2d(sin(x), x = 0 .. 2*PI),
     TicksNumber = Normal, TicksBetween = 10,
     GridVisible = TRUE, SubgridVisible = TRUE,
     GridLineWidth = 0.5*unit::mm,
     SubgridLineWidth = 0.1*unit::mm)
```



Here is a corresponding plot in 3D:

```
plot(plot::Function3d(cos(x*PI)*cos(y*PI), x = 0 .. 2,
```

```
                  y = 0 .. 2),
TicksNumber = Low, TicksBetween = 9,
GridVisible = TRUE, SubgridVisible = TRUE,
GridLineWidth = 0.5*unit::mm,
SubgridLineWidth = 0.1*unit::mm)
```



See Also

MuPAD Functions

`GridInFront` | `GridLineColor` | `GridLineStyle` | `GridVisible`

GridVisible, SubgridVisible, XGridVisible, XSubgridVisible, YGridVisible, YSubgridVisible, ZGridVisible, ZSubgridVisible

Display a coordinate grid?

Value Summary

`GridVisible`	Library wrapper for "{`XGridVisible`, `YGridVisible`}" (2D), "{`XGridVisible`, `YGridVisible`, `ZGridVisible`}" (3D)	See below
`SubgridVisible`	Library wrapper for "{`XSubgridVisible`, `YSubgridVisible`}" (2D), "{`XSubgridVisible`, `YSubgridVisible`, `ZSubgridVisible`}" (3D)	See below
`XGridVisible`, `XSubgridVisible`, `YGridVisible`, `YSubgridVisible`, `ZGridVisible`, `ZSubgridVisible`	Inherited	`FALSE`, or `TRUE`

Graphics Primitives

Objects	Default Values
`plot::CoordinateSystem2d`	`GridVisible`, `SubgridVisible`, `XGridVisible`, `XSubgridVisible`, `YGridVisible`, `YSubgridVisible`: `FALSE`

Objects	Default Values
`plot::CoordinateSystem3d`	`GridVisible`, `SubgridVisible`, `XGridVisible`, `XSubgridVisible`, `YGridVisible`, `YSubgridVisible`, `ZGridVisible`, `ZSubgridVisible`: `FALSE`

Description

With `GridVisible` = `TRUE` versus `GridVisible` = `FALSE`, a coordinate grid extending the “major” axes tick marks is “switched on” or “off”.

With `SubgridVisible`, additional grid lines extending the “minor” axes tick marks are switched on or off.

With `XGridVisible`, `XSubgridVisible` etc., the coordinate lines can be switched on or off separately for each single coordinate direction.

The regular equidistant tick marks along the coordinate axes consist of “minor” tick marks without labels (cf. `TicksBetween`) between “major” tick marks bearing labels (cf. `TicksNumber`, `TicksAnchor`, `TicksDistance`).

Extending the major tick marks, one obtains a grid of coordinate lines. Likewise, extending the minor tick marks yields a refined subgrid of coordinate lines.

With `GridVisible` = `TRUE`, the coordinate grid extending the major tick marks is displayed. With `SubgridVisible` = `TRUE`, the refined subgrid is displayed.

With `XGridVisible` = `TRUE`, `XSubgridVisible` = `TRUE`, only the coordinate lines passing through the ticks along the x-axis are displayed. Likewise, `YGridVisible`, `YSubgridVisible`, `ZGridVisible`, `ZSubgridVisible` allow to display the coordinate lines passing through the ticks along the y and z-axis, respectively.

The coordinate grid is controlled by the ticks marks displayed along the coordinate axes.

Use `TicksNumber` to control the number of automatically generated major tick marks. Alternatively, use `TicksAnchor`, `TicksDistance` to specify the major tick marks explicitly.

Use `TicksBetween` to control the number of minor tick marks.

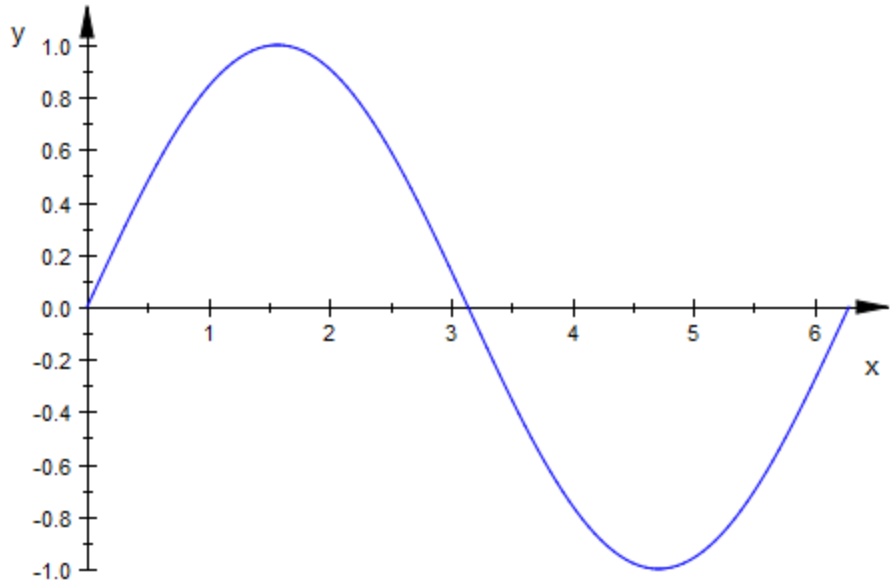
Non-regular tick marks added via `TicksAt` do not generate additional grid lines.

Examples

Example 1

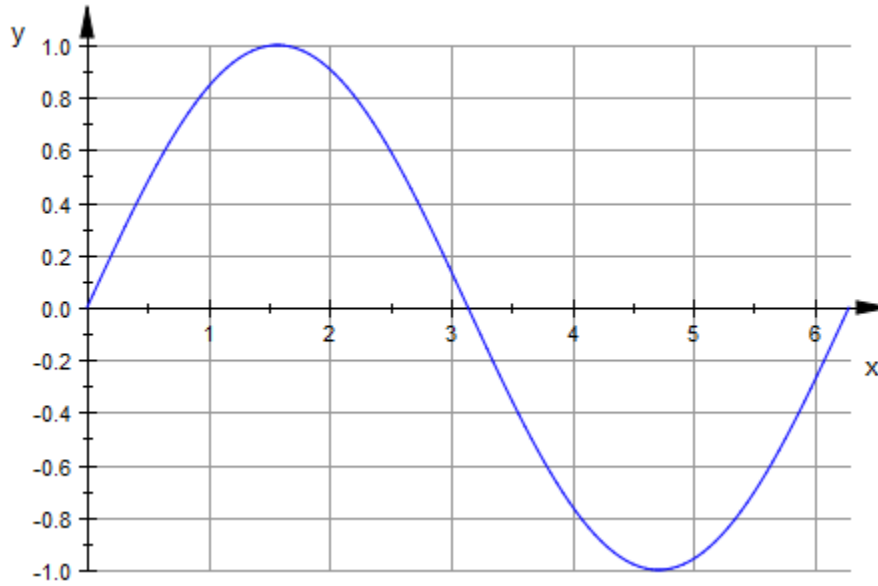
We plot the graph of the sine function without grid lines:

```
plot(plot::Function2d(sin(x), x = 0..2*PI),
     XTicksNumber = Normal, YTicksNumber = High)
```



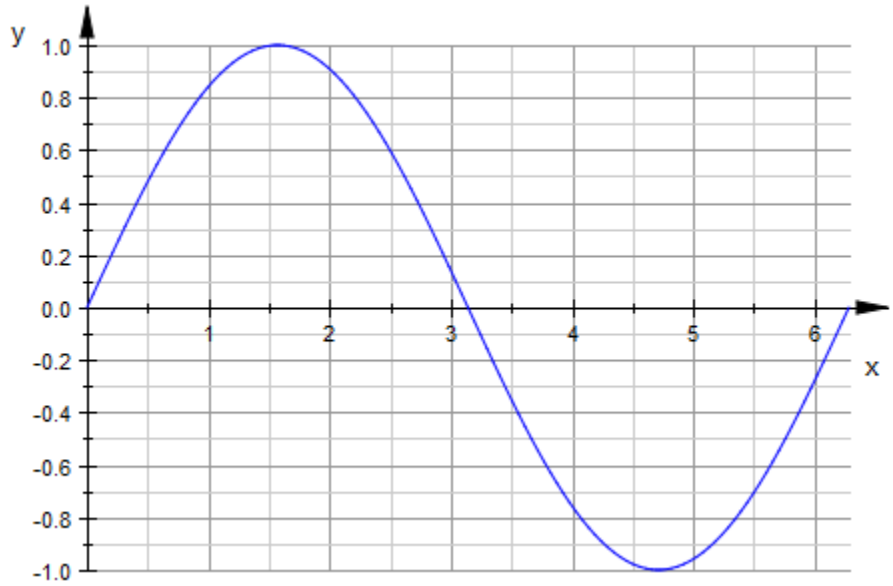
The grid lines are “switched on”:

```
plot(plot::Function2d(sin(x), x = 0..2*PI),
     XTicksNumber = Normal, YTicksNumber = High,
     GridVisible = TRUE):
```



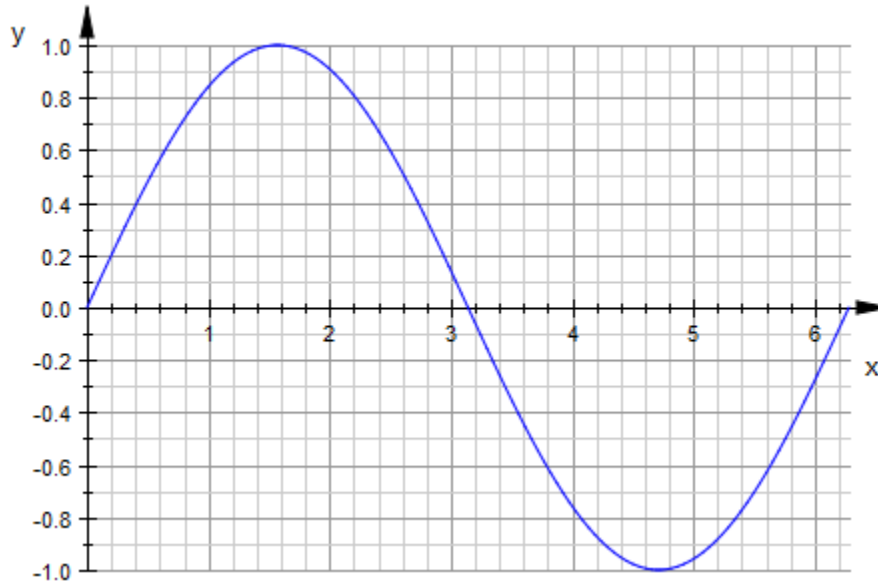
The subgrid lines are switched on as well:

```
plot(plot::Function2d(sin(x), x = 0..2*PI),
     XTicksNumber = Normal, YTicksNumber = High,
     GridVisible = TRUE, SubgridVisible = TRUE):
```



We refine the subgrid in the x-direction via `XTicksBetween`:

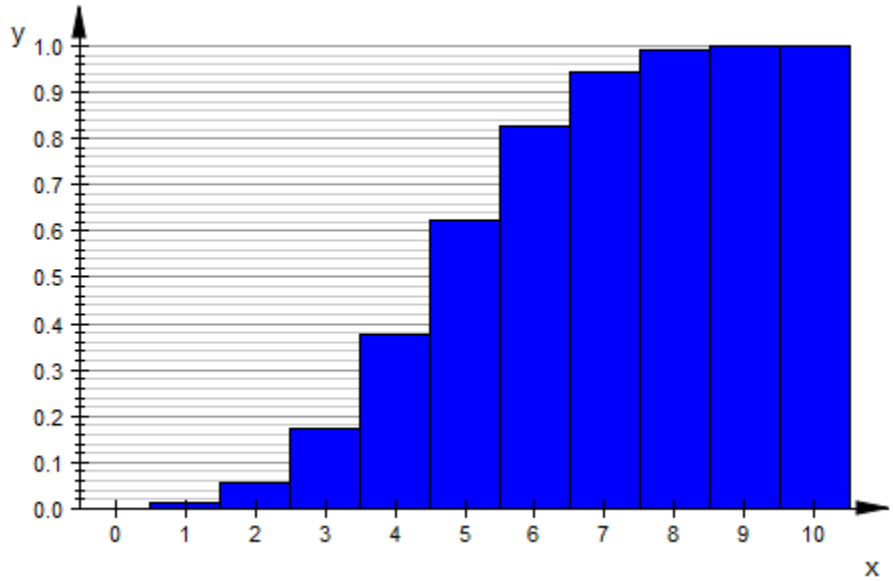
```
plot(plot::Function2d(sin(x), x = 0..2*PI),
     XTicksNumber = Normal, XTicksBetween = 4,
     YTicksNumber = High,
     GridVisible = TRUE, SubgridVisible = TRUE):
```



Example 2

We consider the probabiliy of at least `k` successes when performing `10` independent experiments each with a `50%` chance of success. Consider for this the cumulative density of the binomial distribution given by `stats::binomialCDF`. Quantiles are visualized by introducing horizontal grid lines:

```
f := stats::binomialCDF(10, 0.5):
plot(plot::Bars2d([f(k) $ k = 0..10]),
     XTicksDistance = 1, XTicksBetween = 0,
     XAxisVisible,
     YTicksDistance = 0.1, YTicksBetween = 4,
     YGridVisible, YSubgridVisible)
```

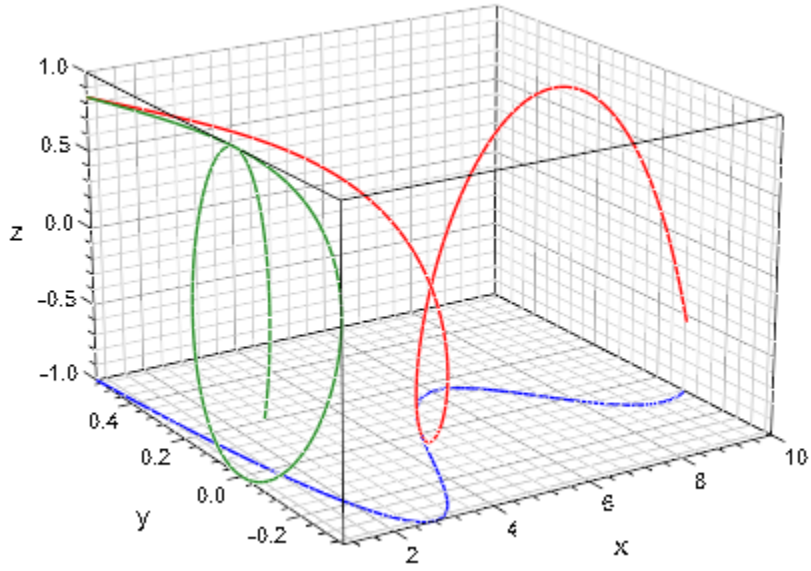



```
delete f:
```

Example 3

Consider a curve in 3D with two of its projections to the coordinate planes. We render the coordinate grid visible:

```
c1 := plot::Curve3d([t, cos(t)/t, sin(t)], t = 1..10,
                    LineColor = RGB::Red):
c2 := plot::Curve3d([1, cos(t)/t, sin(t)], t = 1..10,
                    LineColor = RGB::ForestGreen):
c3 := plot::Curve3d([t, cos(t)/t, -1], t = 1..10,
                    LineColor = RGB::Blue):
plot(c1,c2, c3, TicksBetween = 4, GridVisible = TRUE,
     SubgridVisible = TRUE)
```

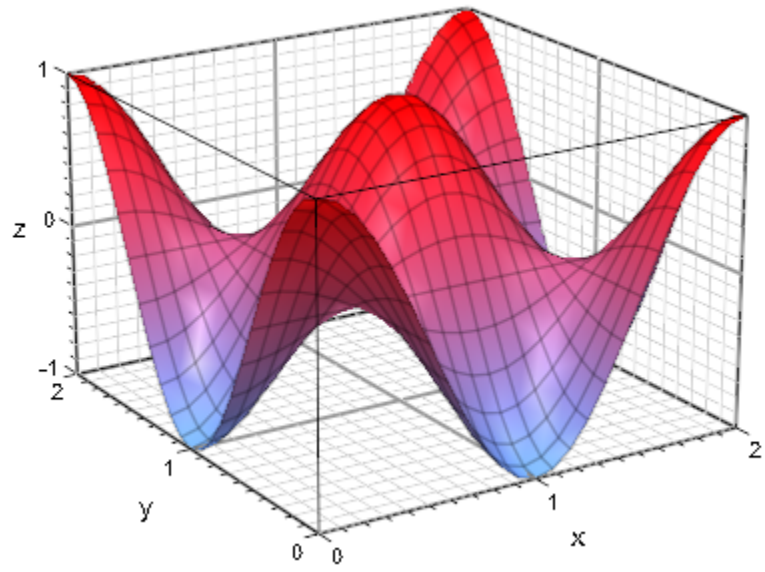



```
delete c1, c2, c3:
```

Example 4

Because of the rather large number of grid lines in the following plot, we use extra fine lines to render the subgrid:

```
plot(plot::Function3d(cos(x*PI)*cos(y*PI), x = 0 .. 2,
                      y = 0 .. 2),
     TicksNumber = Low, TicksBetween = 9,
     GridVisible = TRUE, SubgridVisible = TRUE,
     GridLineWidth = 0.5*unit::mm,
     SubgridLineWidth = 0.1*unit::mm)
```



See Also

MuPAD Functions

`GridInFront` | `GridLineColor` | `GridLineStyle` | `GridLineWidth`

AnimationStyle

Behaviour of the animation toolbar

Value Summary

Inherited	`BackAndForth`, `Loop`, or `RunOnce`

Graphics Primitives

Objects	AnimationStyle Default Values
`plot::Canvas`	`RunOnce`

Description

`AnimationStyle` determines how an animation is played in VCam once it is activated.

`AnimationStyle` determines what has to be done when an animation reaches the end of its playing time. With `RunOnce` the animation stops, with `BackAndForth` the animation reverts and runs through to the beginning and with `Loop` it jumps back to the beginning and runs on from there on.

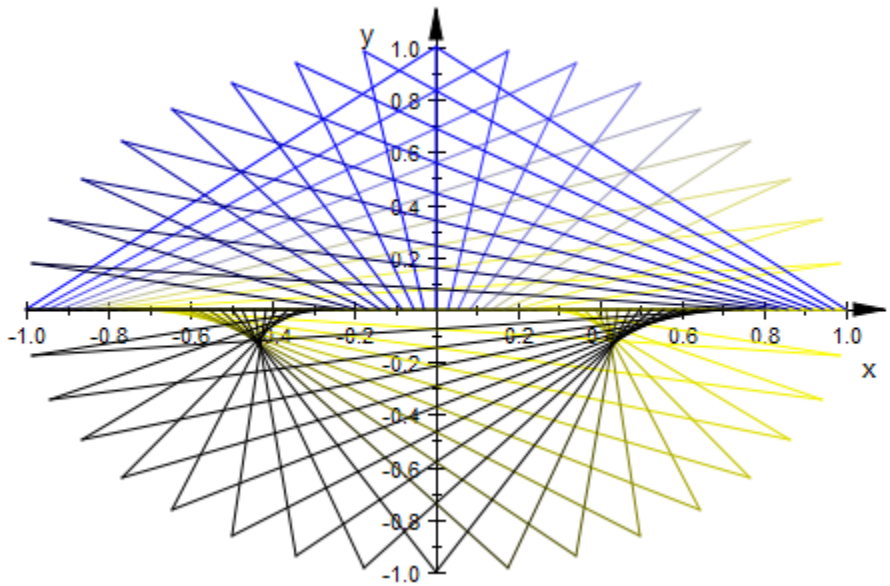
`AnimationStyle` sets the initial value of the `Animation Style` menu in the animation toolbar according to its value.

Examples

Example 1

This examples shows an animation which builds up a picture frame by frame and at the end of the animation time the complete picture is visible. For this kind of animation the value `RunOnce` is a good choice for `AnimationStyle`:

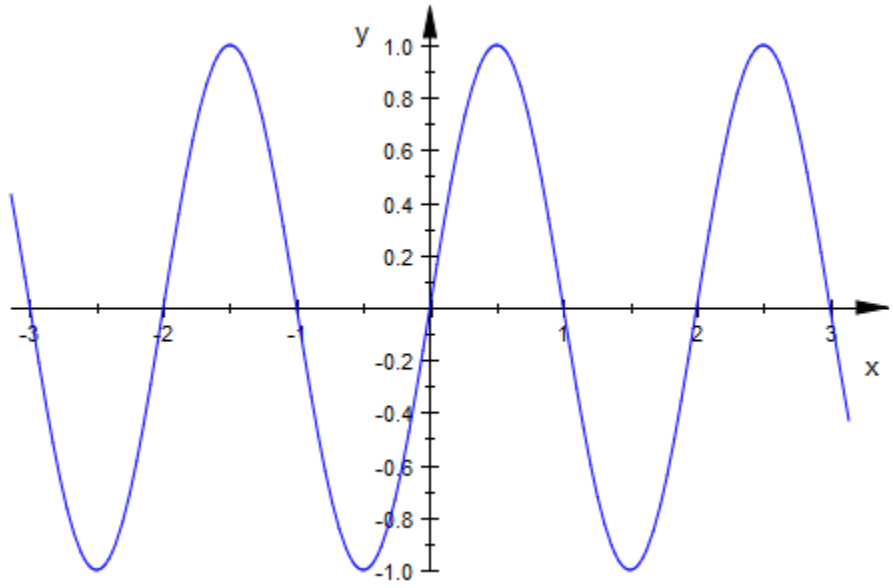
```
plot(plot::Line2d([a/36, 0], [sin(a/18*PI), cos(a/18*PI)],
                  VisibleAfter = a/7.2,
                  Color = [sin(a/18*PI), sin(a/18*PI), cos(a/18*PI)])
                  $ a = -36..36,
     AnimationStyle = RunOnce)
```



Example 2

This example plays in an endless loop and the value `Loop` is choosen because first frame is the natural successor of the last frame of the animation:

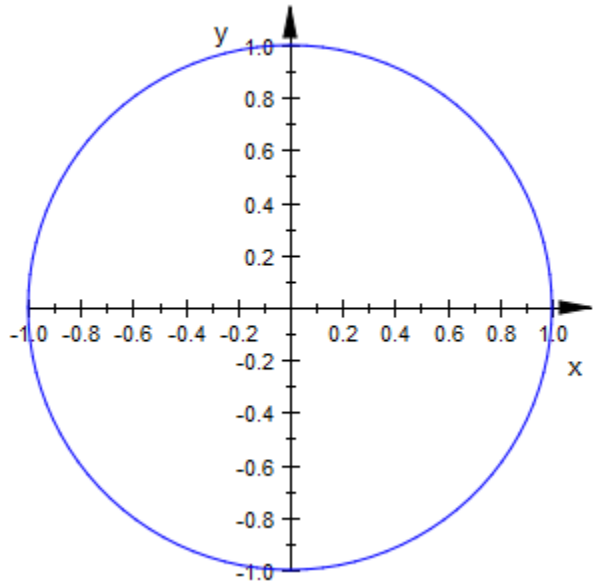
```
plot(plot::Function2d(sin(a*x), x = -PI..PI, a = -PI..PI),
     AnimationStyle = Loop)
```



Example 3

In this example the circle grows from radius 0 to radius 1. With `AnimationStyle = BackAndForth` the circle grows and shrinks in an endless loop:

```
plot(plot::Circle2d(a, a = 0..1),
     AnimationStyle = BackAndForth)
```



See Also

MuPAD Functions
`InitialTime`

AutoPlay

Start animations automatically

Value Summary

Optional	FALSE, or TRUE

Graphics Primitives

Objects	AutoPlay Default Values
`plot::Canvas`	TRUE

Description

When plotting with `AutoPlay = TRUE`, animations will start automatically when the plot is activated. This is the default setting.

Animations created with `AutoPlay = FALSE` start when the corresponding button is pressed.

Frames

Number of frames in an animation

Value Summary

Inherited	Positive integer

Description

`Frames` determines the number of frames in the animation of an object.

`Frames` = `n` with a positive integer `n` sets the number of frames for the animation of an object to `n`.

These frames are played during the real time period given by `TimeBegin` = t_0 and `TimeEnd` = t_1 (in seconds).

The resulting frame rate is $n/(t_1 - t_0)$ (frames per second).

Increasing the number of frames does not mean that the animation lasts longer, because the renderer does not work with a fixed number of frames per second.

Keeping the play period from `TimeBegin` = t_0 to `TimeEnd` = t_1 fixed, an increased number of frames just produces a higher frame rate leading to a smoother animation.

Note that the human eye cannot distinguish between different frames if they change with a rate of more than 25 frames per second. Thus, the number of frames n for an animation should satisfy $n < 25\ (t_1 - t_0)$.

With the default time range `TimeBegin` = t_0 = 0, `TimeEnd` = t_1 = 10 (seconds), it does not make sense to specify `Frames` = `n` with `n > 250`. If a higher n is required to obtain a sufficient resolution of the animated object, one should increase the time for the animation by a sufficiently high value of `TimeEnd`.

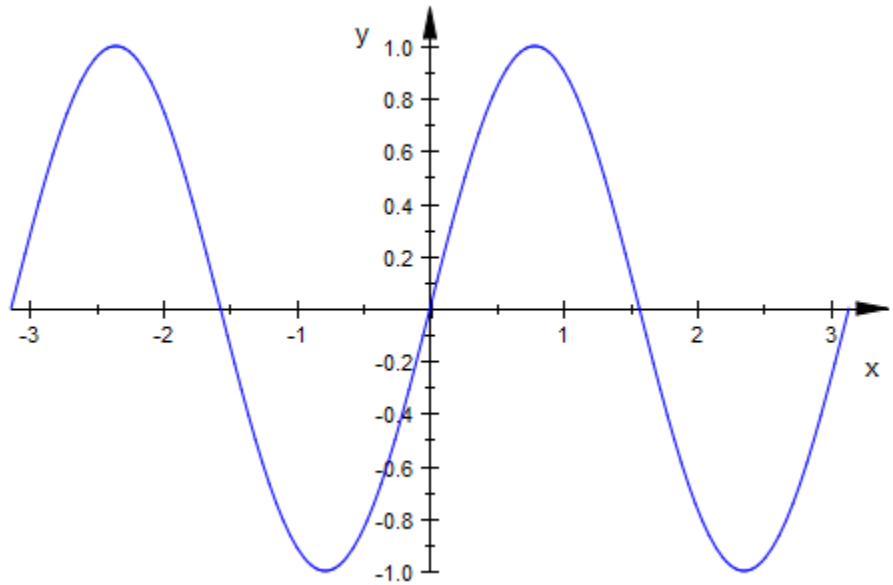
Since the values of `Frames`, `TimeBegin`, `TimeEnd` may be set separately for different objects, it is possible to animate objects in a scene with different frame rates. Cf. “Example 2” on page 24-1652.

Examples

Example 1

We set the number of frames for the following animation to 40. The default animation range of 10 seconds is used. This results in a frame rate of 4 frames per second:

```
plot(plot::Function2d(sin(a*x), x = -PI..PI,
                      a = 1..2, Frames = 40)):
```



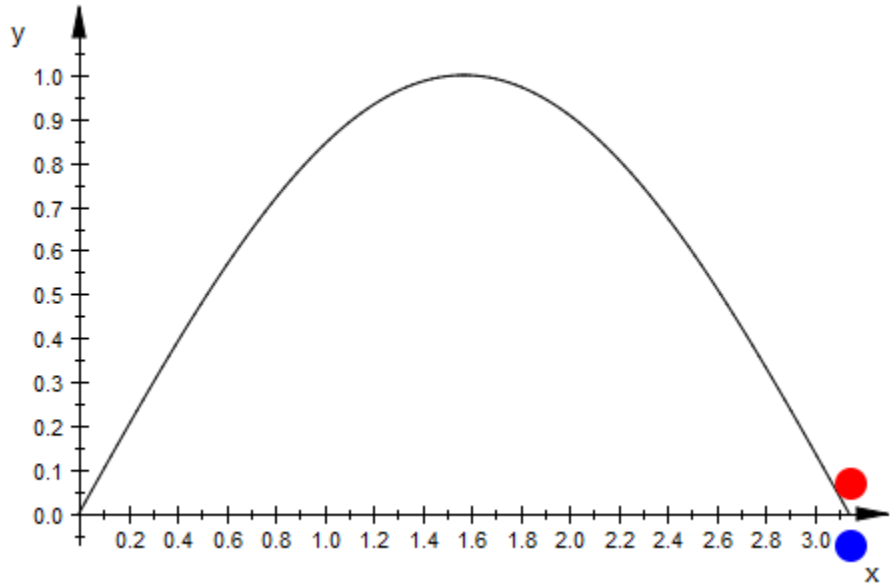
Example 2

Here is an example of different frame rates in one plot. The default animation range of 10 seconds is used.

The red point is sampled with 30 frames in 10 seconds, the blue one with 100 frames in 10 seconds. The animation of the blue point is much smoother:

```
plot(plot::Function2d(sin(x), x = 0..PI,
                      Color = RGB::Black),
```

```
plot::Point2d([a , sin(a) + 0.07], a = 0 .. PI,
              Color = RGB::Red, Frames = 30),
plot::Point2d([a , sin(a) - 0.07], a = 0 .. PI,
              Color = RGB::Blue, Frames = 100),
PointSize = 4*unit::mm):
```



See Also

MuPAD Functions

`ParameterBegin` | `ParameterEnd` | `ParameterName` | `ParameterRange` | `TimeBegin` | `TimeEnd` | `TimeRange` | `VisibleAfter` | `VisibleAfterEnd` | `VisibleBefore` | `VisibleBeforeBegin` | `VisibleFromTo`

More About

- "The Number of Frames and the Time Range"

TimeBegin, TimeEnd, TimeRange, InitialTime

Time of the animation

Value Summary

`InitialTime`, `TimeBegin`, `TimeEnd`	Inherited	Real number
`TimeRange`	[`TimeBegin` .. `TimeEnd`]	Range of arithmetical expressions

Graphics Primitives

Objects	Default Values
`plot::Canvas`	

Description

`TimeBegin` = t_0 defines the starting time t_0 of the animation of an object.

`TimeEnd` = t_1 defines the time t_1 for the end of the animation.

`TimeRange` = `t_0 .. t_1` is a short cut for setting both `TimeBegin` = t_0 and `TimeEnd` = t_1.

`InitialTime` = t_2 defines the time t_2 for the initial position of the animation slider.

Animations are defined object by object. Each animated object is animated for a certain time span specified by `TimeBegin` and `TimeEnd` setting the real start and end time in seconds.

The total real time span of an animated plot is the physical real time given by the minimum of the `TimeBegin` values of all animated objects in the plot to the maximum of the `TimeEnd` values of all the animated objects:

- When a plot containing animated objects is created, the real time clock is set to the minimum of the `TimeBegin` values of all animated objects in the plot. The real time

clock is started when pushing the `play' button for animations in the graphical user interface.

- Before the real time reaches the `TimeBegin` value t_0 of an animated object, this object is static in the state corresponding to the begin of its animation. Depending on the attribute `VisibleBeforeBegin`, it may be visible or invisible before t_0.
- During the time from t_0 to t_1, the object changes from its original to its final state.
- After the real time reaches the `TimeEnd` value t_1, the object stays static in the state corresponding to the end of its animation. Depending on the attribute `VisibleAfterEnd`, it may stay visible or become invisible after t_1.
- The animation of the entire plot ends with the physical time given by the maximum of the `TimeEnd` values of all animated objects in the plot.

If all animated objects in a plot share the same values `TimeBegin` $= t_0$ and `TimeEnd` $= t_1$, the physical time span of the animation is $t_1 - t_0$ (in seconds). During this time, all animated objects change from their initial to their final state.

Separate settings for `TimeBegin` and `TimeEnd` in different animated objects allow to synchronize the animations.

With the optional attribute `InitialTime` the initial position of the animation slider can be set to any time value t_2 between t_0 and t_1. If `InitialTime` is not set, the slider will be placed at the beginning of the animation.

The attributes `VisibleAfter`, `VisibleBefore`, and `VisibleFromTo` allow special “visibility animations” in which objects are visible for a limited time only.

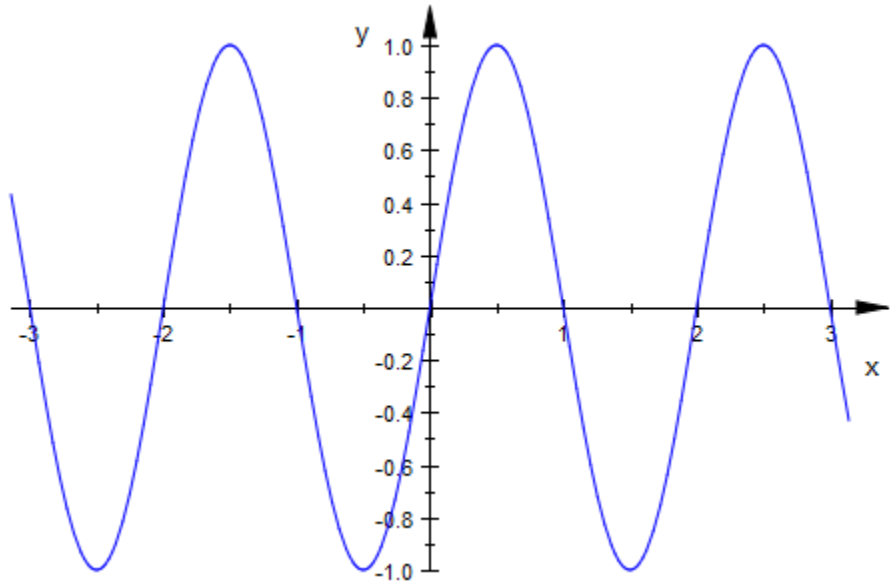
Note: The attributes `VisibleAfter`, `VisibleBefore`, and `VisibleFromTo` implicitly set values for `TimeBegin` and `TimeEnd` (and, therefore, also for `TimeRange`). Consequently, these attributes should not be used simultaneously in the definition of an animated object.

Examples

Example 1

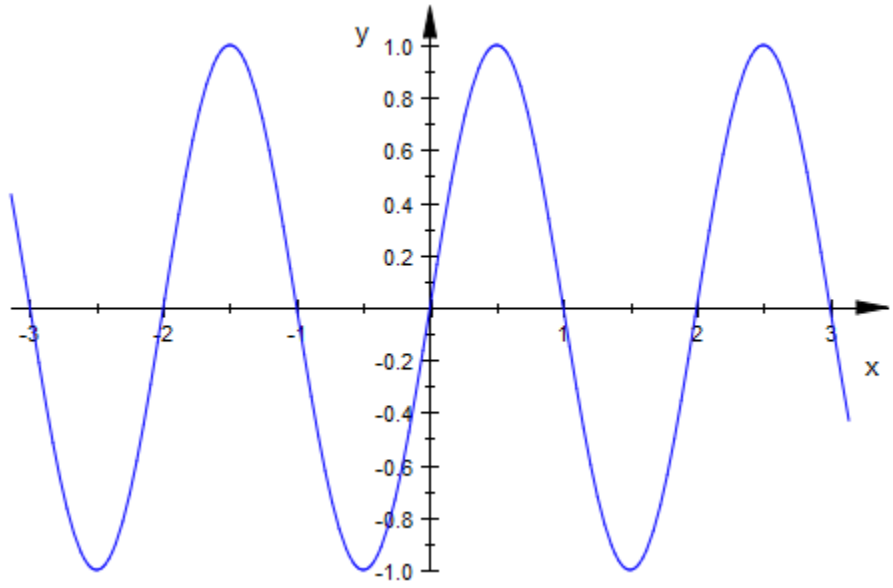
By default, an animation plays for 10 seconds. Keeping the default value `TimeBegin = 0`, this time can be reduced to 5 seconds by setting `TimeEnd = 5`:

```
plot(plot::Function2d(sin(a*x), x= - PI .. PI, a = -PI..PI,
                      TimeEnd = 5)):
```



The total time of the animation is the difference between `TimeEnd` and `TimeBegin`. Hence, the following animation plays 5 seconds as well:

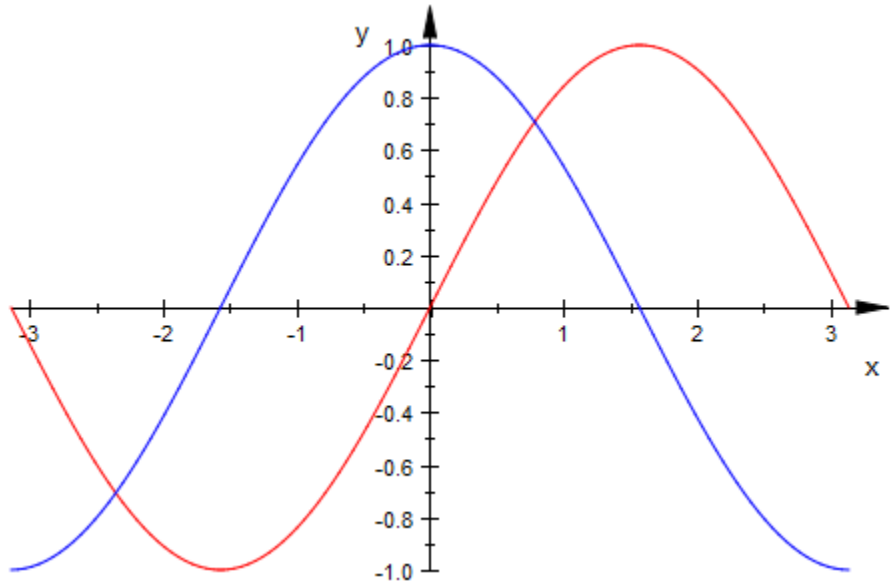
```
plot(plot::Function2d(sin(a*x), x = -PI..PI, a = -PI..PI,
                      TimeRange = 5..10)):
```



Example 2

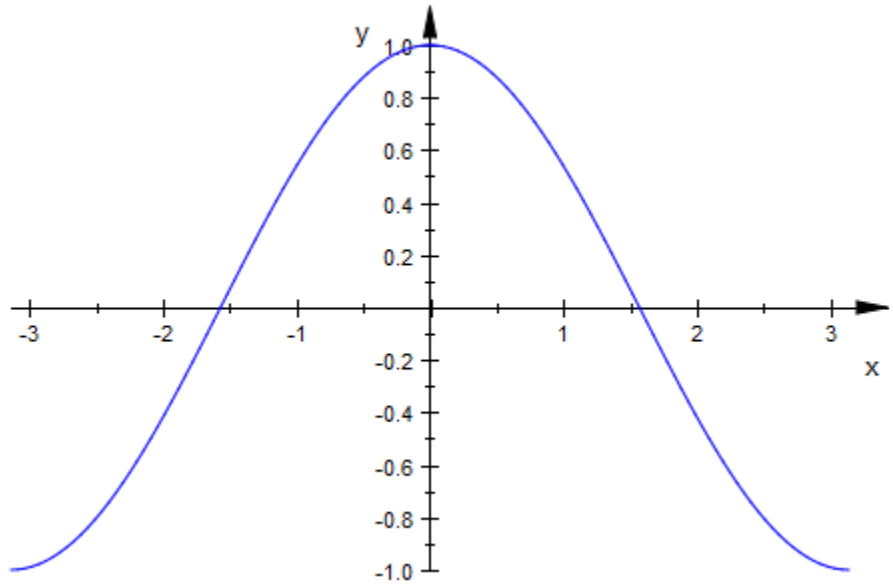
Using different time ranges allows to synchronize the animations of different objects. Here we plot two functions. The first function is animated from 0 to 5 (seconds) and then stays static in its final state. The second function stays static in its initial state for 5 seconds and is then animated in the range from 5 to 10 (seconds):

```
f1 := plot::Function2d(a*sin(x), x = -PI..PI, a = -1..1,
                       Color = RGB::Red, TimeRange = 0..5):
f2 := plot::Function2d(a*cos(x), x = -PI..PI, a = -1..1,
                       Color = RGB::Blue, TimeRange = 5..10):
plot(f1 ,f2):
```



Both functions are visible outside the time range of their animations. We use the attributes `VisibleAfterEnd` and `VisibleBeforeBegin` to make them visible only during their animations:

```
f1::VisibleAfterEnd := FALSE:
f2::VisibleBeforeBegin := FALSE:
plot(f1, f2):
```

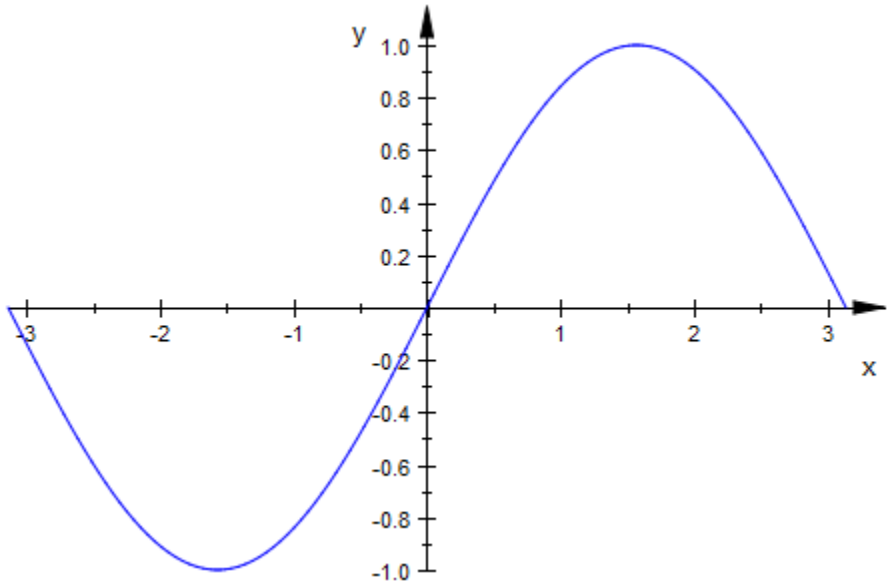



```
delete f1, f2:
```

Example 3

The following animation uses the implicitly given `TimeRange` from 0 to 10 seconds, but the first image shown is at the time 5:

```
plot(plot::Function2d(a*sin(x), x = -PI..PI, a = -1..1),
                      InitialTime=5):
```



See Also

MuPAD Functions

`AnimationStyle` | `Frames` | `ParameterBegin` | `ParameterEnd` | `ParameterName` | `ParameterRange` | `VisibleAfter` | `VisibleAfterEnd` | `VisibleBefore` | `VisibleBeforeBegin` | `VisibleFromTo`

More About

- "The Number of Frames and the Time Range"
- "Advanced Animations: The Synchronization Model"

VisibleAfter, VisibleBefore, VisibleFromTo

Object visible at time value

Value Summary

`VisibleAfter`, `VisibleBefore`, `VisibleFromTo`	Library wrapper for "`TimeEnd`, `TimeBegin`, `VisibleAfterEnd`, and `VisibleBeforeBegin`"	Non-negative real number

Description

`VisibleAfter = t0` renders an object invisible until the real time `t0` has elapsed in an animation. Then the object becomes visible.

`VisibleBefore = t1` renders an object visible until the time `t1`. Then the object becomes invisible.

`VisibleFrom = t_0 .. t_1` renders an object invisible until the time `t0`. Then the object becomes visible. After the time `t1` it becomes invisible again.

`VisibleAfter`, `VisibleBefore`, `VisibleFromTo` allow to implement animated visibility of objects. This also includes otherwise static objects, which become animated objects when one of these attributes is used.

The attributes `VisibleBeforeBegin` and `VisibleAfterEnd` control the visibility of objects outside the time range of their animation set by `TimeBegin` and `TimeEnd`. See `TimeBegin`, `TimeEnd` for details.

`VisibleAfter`, `VisibleBefore`, `VisibleFromTo` provide short cuts for suitable settings of the attributes `TimeBegin`, `VisibleBeforeBegin`, `TimeEnd`, `VisibleAfterEnd` that produce the desired visibility effects.

`VisibleAfter = t0` is a short cut for setting the following attribute values:

TimeBegin = t_0, VisibleBeforeBegin = FALSE,

TimeEnd = t_0, VisibleAfterEnd = TRUE.

The resulting effect is that the corresponding object is invisible at the beginning of the animation. It becomes visible at the time t_0, staying visible until the end of the animation.

The time t_0 has to be a real numerical value giving the real time in seconds.

VisibleBefore = t_1 is a short cut for setting the following attribute values:

TimeBegin = t_1, VisibleBeforeBegin = TRUE,

TimeEnd = t_1, VisibleAfterEnd = FALSE.

The resulting effect is that the corresponding object is visible at the beginning of the animation. At the time t_1 it becomes invisible, staying invisible until the end of the animation.

The time t_1 has to be a real numerical value giving the real time in seconds.

VisibleFromTo = t_0 .. t_1 is a short cut for setting the following attribute values:

TimeBegin = t_0, VisibleBeforeBegin = FALSE,

TimeEnd = t_1, VisibleAfterEnd = FALSE.

The resulting effect is that the corresponding object is visible only from the time t_0 until the time t_1.

Note: The attributes VisibleAfter = t_0 and VisibleBefore = t_1 should not be combined to create visibility for the time range between t_0 and t_1. (Conflicting values are set implicitly for VisibleBeforeBegin etc.) Use VisibleFromTo = t_0 .. t_1 instead.

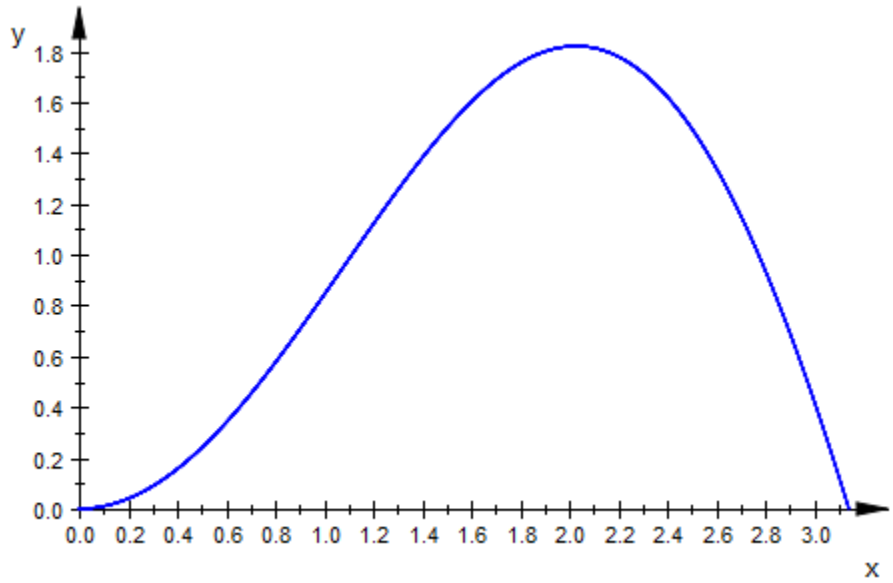
Note: VisibleAfter, VisibleBefore, VisibleFromTo should not be combined with any of the the attributes TimeBegin, TimeEnd, VisibleBeforeBegin or VisibleAfterEnd, since implicit values for these attributes are set.

Examples

Example 1

The following animation consists of 100 pieces of the graph of the function $x\,sin(x)$. At the times $t = 0.1, 0.2$ etc., an additional piece of the function becomes visible until, finally, the whole graph is built up:

```
plot(plot::Function2d(x*sin(x), x = (i - 1)*PI/100 .. i*PI/100,
                      VisibleAfter = i/10) $ i = 1..100)
```

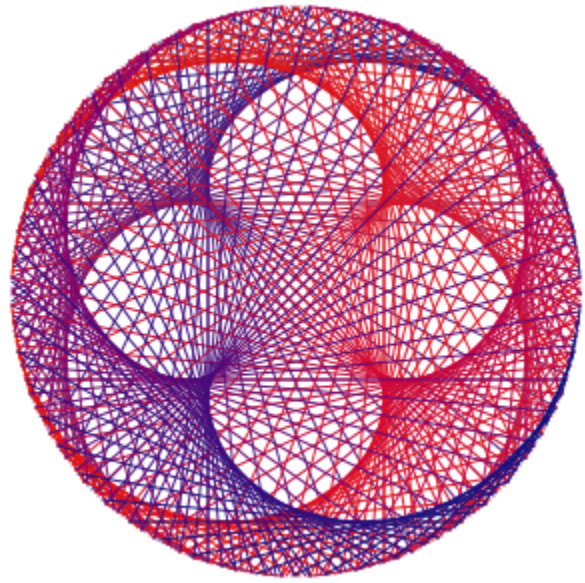


Example 2

This example creates an animated “spider net”. It consists of several lines which appear one after the other at the times given by `VisibleAfter` until the full net is visible at the end of the animation:

```
SpiderNet :=
proc(move, move1, rc, gc, bc)
  local r, lines, x, y, x1, y1;
```

```
begin
  r := 1.0:
  lines := [FAIL $ 361]:
  for i from 0 to 360 do
    thet := float(i*PI/180);
    x     := r * cos(move  * thet);
    y     := r * sin(move  * thet);
    x1    := r * cos(move1 * thet);
    y1    := r * sin(move1 * thet);
    lines[i+1] :=
      plot::Line2d([x, y] ,[x1, y1],
                   Color = [abs(rc*sin(i*PI/360)),
                            abs(gc*sin(i*PI/360 + PI/4)),
                            abs(bc*sin(i*PI/360 + PI/2))],
                   VisibleAfter = i/36
      );
  end_for:
  plot::Group2d(op(lines), Name = "SpiderNet",
                Axes = None, Scaling = Constrained)
end_proc:

plot(SpiderNet(3, 7, 0.9, 0.1, 0.5))
```



```
delete SpiderNet:
```

Example 3

This example creates an animated “Maurer rose”. Here the animation starts with the full object. During the animation the lines disappear at the times given by `VisibleBefore`:

```
MaurerRose := proc(n, b, rc, gc, bc)
  local lines, i, thet, r, x, y, x1, y1;
begin
  r := 1.0;
  lines := [FAIL $ 361]:
  b := float(b*PI/180);
  for i from 0 to 360 do
    thet := float(i*PI/180);
    x    := r * sin(n*thet)    * cos(thet);
    y    := r * sin(n*thet)    * sin(thet);
    x1   := r * sin(n*(thet+b))* cos(thet+b);
    y1   := r * sin(n*(thet+b))* sin(thet+b);
    lines[i+1] :=
      plot::Line2d([x, y], [x1, y1],
                   Color = [abs(rc*sin(i*PI/360)),
                            abs(gc*sin(i*PI/360 + PI/4)),
                            abs(bc*sin(i*PI/360 + PI/2))],
                   VisibleBefore = i/36
      );
  end_for:
  plot::Group2d(op(lines), Name = "MaurerRose",
                Axes = None, Scaling = Constrained):
end_proc:

plot(MaurerRose(4, 120, 0.1, 0.5, 0.9)):
```

```
delete MaurerRose:
```

Example 4

This example creates an animated “Lissajous net”. It is built up from lines that have a life span of only 2 seconds each, set by `VisibleFromTo`:

```
LissajousNet := proc(r, a, b, R, A, B, rc, gc, bc)
  local lines, i, thet;
begin
  lines := [FAIL $ 361]:
  for i from 0 to 360 do
    thet := float(i*PI/180);
    x     := r * cos(a*thet);
    y     := r * sin(b*thet);
    x1    := R * cos(A*thet);
    y1    := R * sin(B*thet);
    lines[i+1] :=
      plot::Line2d([x, y], [x1, y1],
                   Color = [abs(rc*sin(i*PI/360)),
                            abs(gc*sin(i*PI/360 + PI/4)),
```

```
                              abs(bc*sin(i*PI/360 + PI/2))],
                    VisibleFromTo = i/36 .. i/36 + 2
      );
  end_for:
  plot::Group2d(op(lines), Name = "LissajousNet",
                Axes = None, Scaling = Constrained):
end_proc:

plot(LissajousNet(2, 3, 4, 1, 6, 3, 0.7, 0.1, 0.99))
```



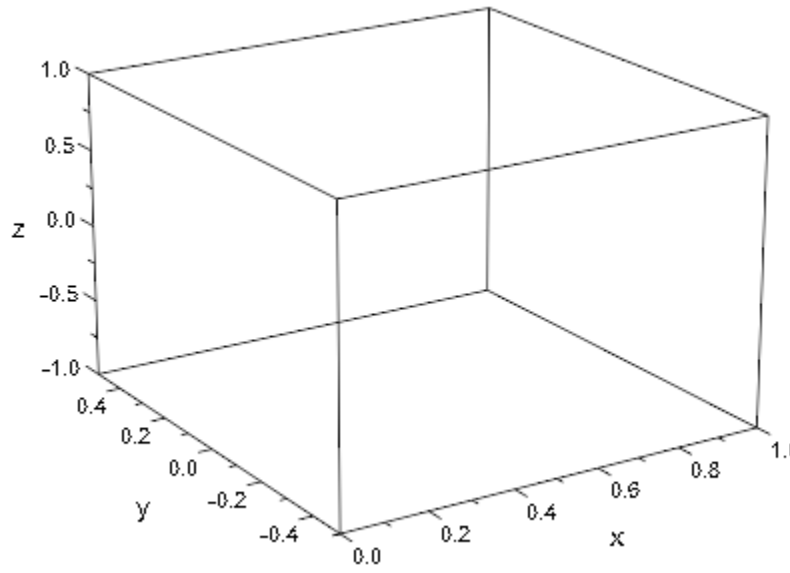
```
delete LissajousNet:
```

Example 5

Here is a 3D example of an animation. A “spider net” is built up with lines that have a life span of 4 seconds each:

```
SpiderNet3d := proc(a, b, c, rc, gc, bc)
  local r, lines, i, x, x1, y, y1, thet, z1, z;
begin
```

```
  r := 1.0:
  lines := [FAIL $ 361]:
  for i from 0 to 360 do
    thet := float(i*PI/180);
    x     := r * cos(thet)*cos(thet);
    y     := r * sin(thet)*cos(thet);
    z     := r * sin(thet):
    x1    := r * cos(a*thet)*cos(a*thet);
    y1    := r * sin(b*thet)*cos(b*thet);
    z1    := r * sin(c*thet):
    lines[i+1] :=
      plot::Line3d([x,y,z],[x1,y1,z1],
                   Color = [abs(rc*sin(i*PI/360)),
                            abs(gc*sin(i*PI/360 + PI/4)),
                            abs(bc*sin(i*PI/360 + PI/2))],
                   VisibleFromTo = i/36 .. i/36 + 4
      );
  end_for:
  plot::Group3d(op(lines), Name = "SpiderNet3d"):
end_proc:

plot(SpiderNet3d(2, 1, 3, 0.99, 0.9, 0.1))
```



Algorithms

The last examples on this page are taken from the mathPAD Online Edition (`http://www.mupad.com/mathpad/recreations.html`) written by Prof. Mirek Majewski. See there for details about the mathematics behind the examples above.

See Also

MuPAD Functions

`Frames` | `ParameterBegin` | `ParameterEnd` | `ParameterName` | `ParameterRange` | `TimeBegin` | `TimeEnd` | `TimeRange` | `VisibleAfterEnd` | `VisibleBeforeBegin`

More About

- "Animations"
- "Frame by Frame Animations"

VisibleBeforeBegin, VisibleAfterEnd

Object visible before or after its animation time starts?

Value Summary

`VisibleAfterEnd`, `VisibleBeforeBegin`	Inherited	`FALSE`, or `TRUE`

Description

`VisibleBeforeBegin`, `VisibleAfterEnd` determine the visibility of an object before the begin and after the end of its own animation time span, respectively.

Animations are defined object by object. Each animated object is animated for a certain time span specified by `TimeBegin` and `TimeEnd` setting the real start and end time in seconds.

The total real time span of an animated plot is the physical real time given by the minimum of the `TimeBegin` values of all animated objects in the plot to the maximum of the `TimeEnd` values of all the animated objects.

Thus, the time span of an animated plot may be larger than the time spans of the animations of individual objects.

With `VisibleBeforeBegin = TRUE`, an object is visible as a static object when the animation of the entire plot starts. Its state correponds to the start of its own animation. It begins to change, when the starting time of its own animation set by `TimeBegin` is reached.

With `VisibleBeforeBegin = FALSE`, an object is invisible when the animation of the entire plot starts. It becomes visible when the starting time of its own animation is reached.

With `VisibleAfterEnd = TRUE`, an object stays visible in the final state of its animation after the end of its own animation time span set by `TimeEnd`.

With `VisibleAfterEnd = FALSE`, an object becomes invisible at the end of its own animation.

`VisibleBeforeBegin`, `VisibleAfterEnd` is useful only in plots consisting of several animated objects with different time spans of their animations.

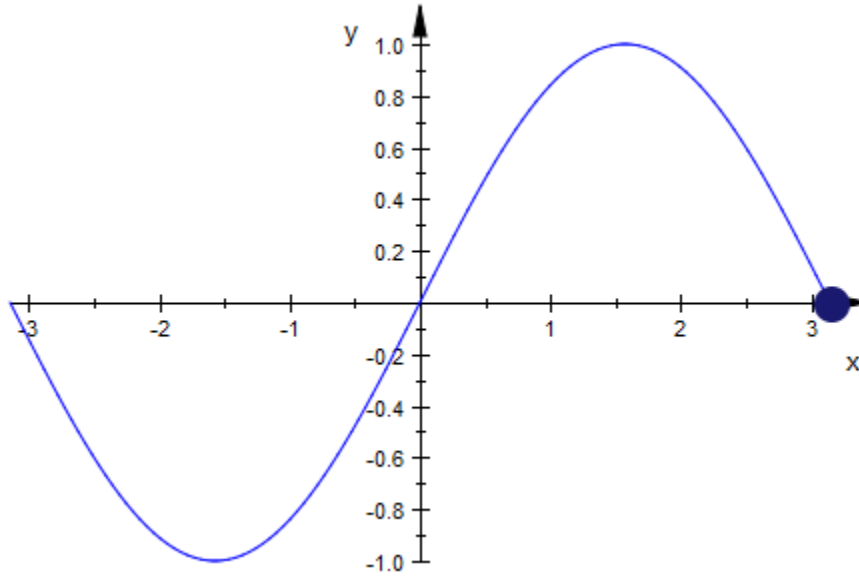
Also consider the attributes `VisibleAfter`, `VisibleBefore`, and `VisibleFromTo` to animate the visibility of objects.

Examples

Example 1

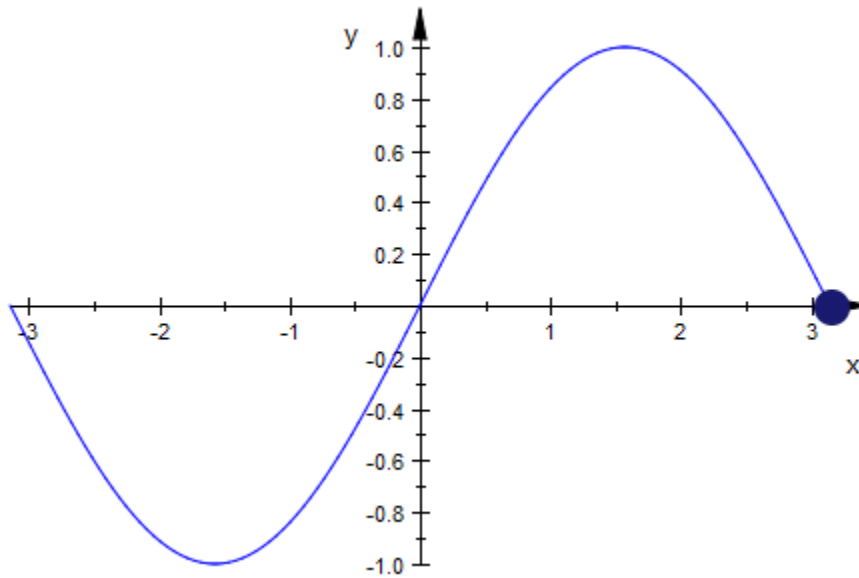
In the first 5 seconds of the following animation, the sine function draws itself. Afterwards, a point wanders along the graph:

```
f := plot::Function2d(sin(x), x = -PI..a, a = -PI..PI,
                      TimeRange = 0..5):
p := plot::Point2d(a, sin(a), PointSize = 5*unit::mm,
                   a = -PI..PI, TimeRange = 5..10):
plot(f, p)
```



The point is visible for the first 5 seconds, too, because it used the default setting `VisibleBeforeBegin = TRUE`. With `VisibleBeforeBegin = FALSE`, the point is invisible at the start of the animation. It appears after 5 seconds, when its own animation begins:

```
p::VisibleBeforeBegin := FALSE:
plot(f, p)
```



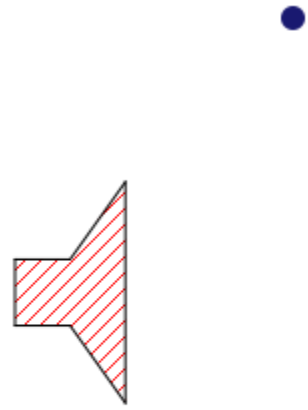

```
delete p, f:
```

Example 2

The `plot::Polar` object in the following animation is only visible in its `TimeRange` from the 3rd to the 7th second:

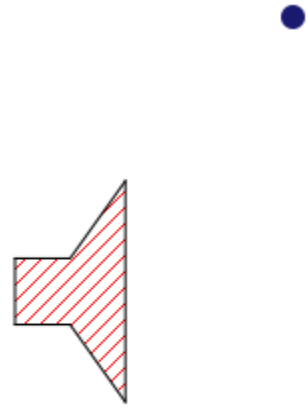
```
Speaker :=
     plot::Polygon2d([[0.5, -1], [0.5, 1], [0, 0.3],
                      [-0.5, 0.3], [-0.5, -0.3], [0, -0.3],
                      [0.5, -1]], Color = RGB::Black, Filled):
```

```
Point := plot::Point2d([2, a], a = -2.5..2.5,
                       PointSize = 3*unit::mm):

plot(plot::Polar([1 + 0.1*(2 + sin(20*a))*cos(20*phi), phi],
                 phi = -1..1, a = 0..3, TimeRange = 3..7,
                 VisibleBeforeBegin = FALSE,
                 VisibleAfterEnd = FALSE),
     Speaker, Point, Axes = None)
```



The previous command is equivalent to:

```
plot(plot::Polar([1 + 0.1*(2 + sin(20*a))*cos(20*phi), phi],
                 phi = -1..1, a = 0..3, VisibleFromTo = 3..7),
     Speaker, Point, Axes = None)
```

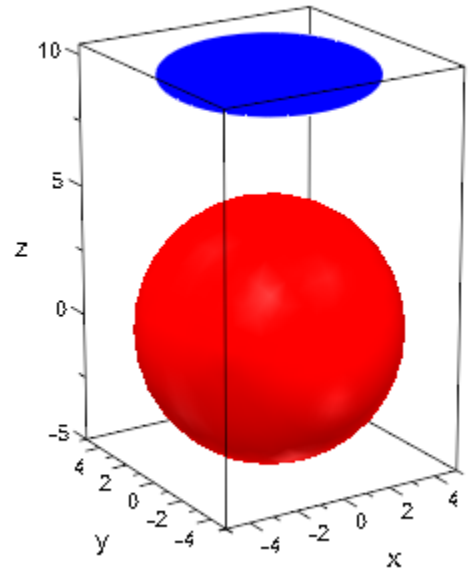


```
delete Speaker, Point:
```

Example 3

A `circle` is tumbling around in 3D. After 3 seconds, a growing sphere becomes visible. From the 5th second through to the end of the animation, the sphere stays visible with the constant radius 5, while the circle moves further out:

```
plot(plot::Circle3d(4, [sin(a), cos(a), a],
                    [sin(a), cos(a), a], a = 0..10,
                    Frames = 100, TimeRange = 0..10,
                    Filled = TRUE, FillColor = RGB::Blue),
     plot::Sphere(a, [0, 0, 0], Color=RGB::Red,
                  a = 3..5, TimeRange = 3 .. 5, Frames = 20,
                  VisibleBeforeBegin = FALSE)):
```



See Also

MuPAD Functions

`Frames` | `ParameterBegin` | `ParameterEnd` | `ParameterName` | `ParameterRange` | `TimeBegin` | `TimeEnd` | `TimeRange` | `VisibleAfter` | `VisibleBefore` | `VisibleFromTo`

Footer, Header

Footer text

Value Summary

`Footer`, `Header`	Optional	Text string

Graphics Primitives

Objects	Default Values
`plot::Canvas`, `plot::Scene2d`, `plot::Scene3d`	

Description

`Footer = "…"` sets a text to be displayed at the bottom of a scene or canvas.

`Header = "…"` sets a text to be displayed at the top of a scene or canvas.

As described in the introduction, each plot consists of a canvas containing one or more scenes. Using `Header` and `Footer`, you can set captions for both levels of nesting, above and/or below the contents.

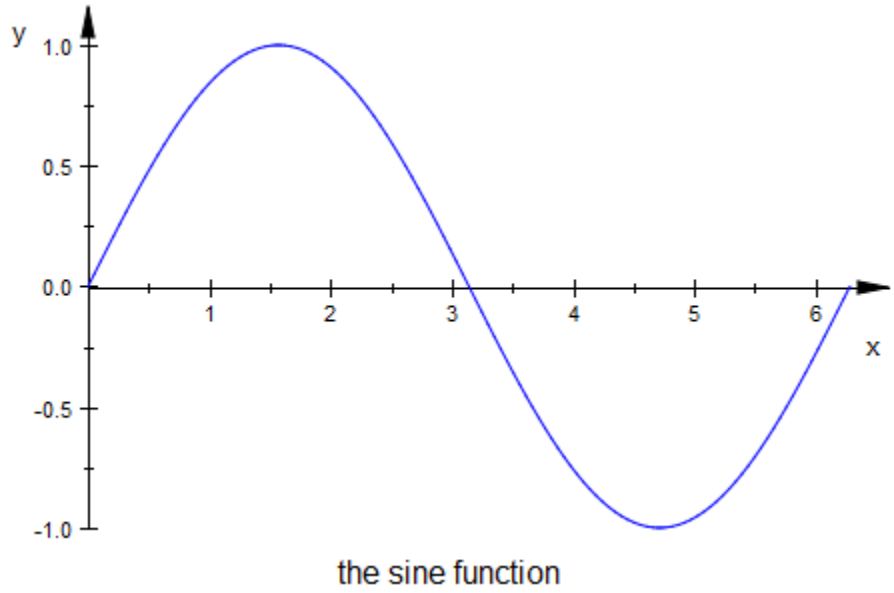
To change the appearance of the captions, please use the attributes `FooterAlignment` and `HeaderAlignment` for positioning and `FooterFont` and `HeaderFont` to control fonts and sizes.

Examples

Example 1

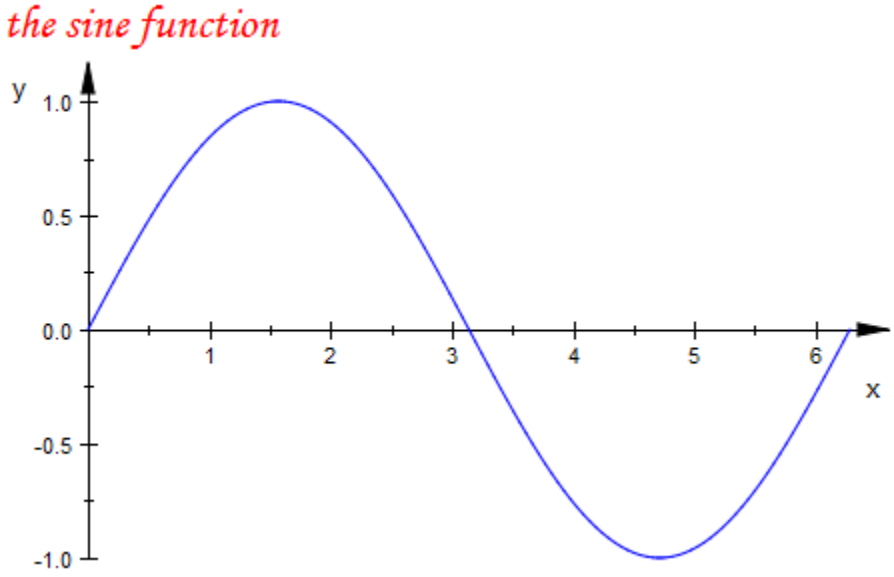
The easiest way of setting a caption is to include a canvas-caption in a `plot` command:

```
plot(plot::Function2d(sin(x), x = 0..2*PI),
     Footer = "the sine function"):
```



You can also set style controlling attributes in this context:

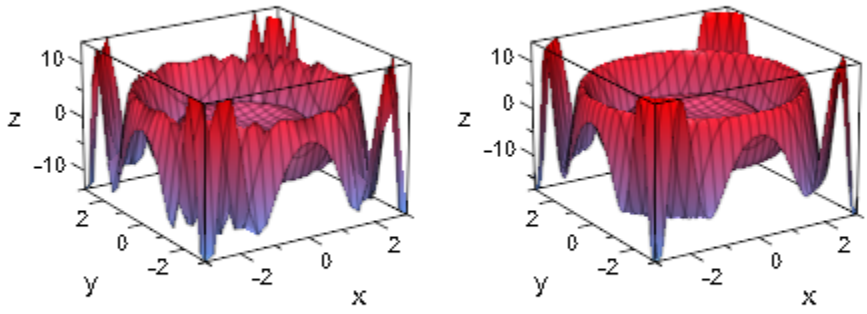
```
plot(plot::Function2d(sin(x), x = 0..2*PI),
     Header = "the sine function",
     HeaderFont = ["Monotype Corsiva", 17, RGB::Red],
     HeaderAlignment = Left):
```



Example 2

Advanced users may want to plot several scenes together. These can be given individual captions:

```
f1 := plot::Function3d(sin(x^2 + y^2)*(x^2 + y^2),
                       x = -3..3, y = -3 ..3,
                       AdaptiveMesh = 0):
f2 := plot::modify(f1, AdaptiveMesh = 2):

s1 := plot::Scene3d(f1, Footer = "AdaptiveMesh = 0"):
s2 := plot::Scene3d(f2, Footer = "AdaptiveMesh = 2"):
plot(s1, s2, Layout = Horizontal)
```



AdaptiveMesh = 0 AdaptiveMesh = 2

See Also

MuPAD Functions

`FooterAlignment` | `FooterFont` | `HeaderAlignment` | `HeaderFont` | `Title`

FooterAlignment, HeaderAlignment

Alignment of footer of canvas and scenes

Value Summary

`FooterAlignment`, `HeaderAlignment`	Inherited	`Center`, `Left`, or `Right`

Graphics Primitives

Objects	Default Values
`plot::Canvas`, `plot::Scene2d`, `plot::Scene3d`	`FooterAlignment`, `HeaderAlignment`: `Center`

Description

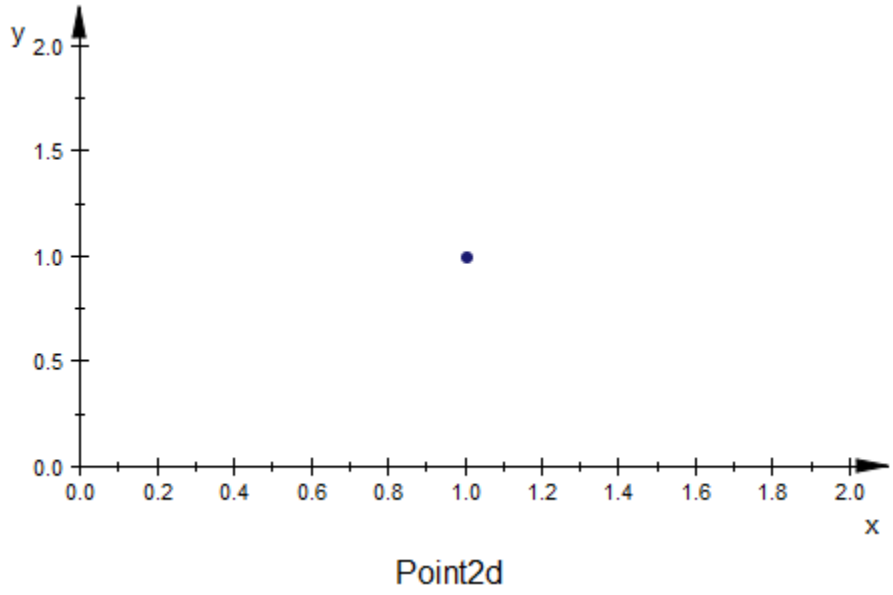
Using the attributes `Footer` and `Header`, a canvas or scene can be given a caption. `FooterAlignment` and `HeaderAlignment` control the horizontal alignment of these captions.

Examples

Example 1

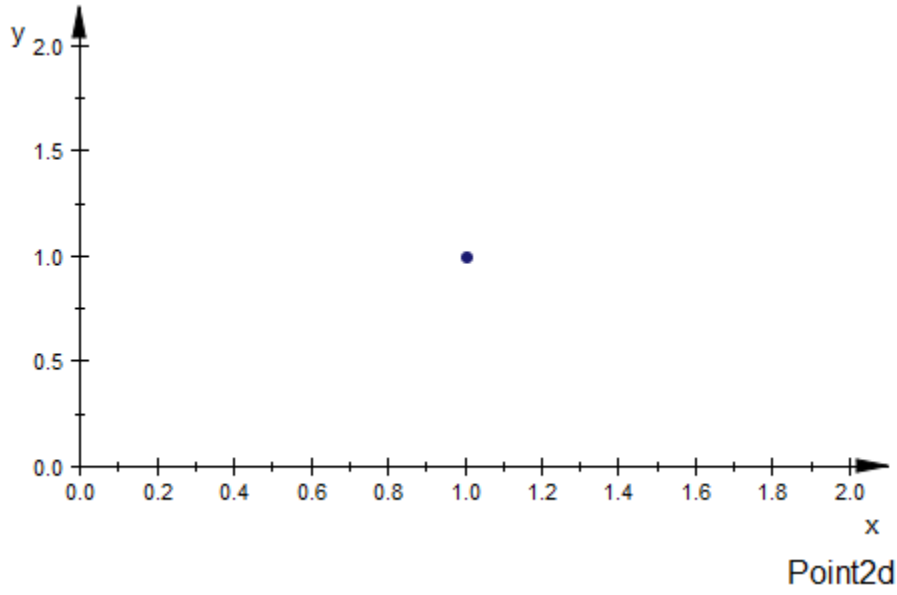
Using the default placement, a footer is centered:

```
plot(plot::Point2d([1,1]), Footer="Point2d")
```



We may want to place the footer more to the right:

```
plot(plot::Point2d([1,1]), Footer="Point2d",
     FooterAlignment = Right)
```



See Also

MuPAD Functions

`Footer` | `FooterFont` | `Header` | `HeaderFont` | `HorizontalAlignment` | `LegendAlignment` | `VerticalAlignment`

HorizontalAlignment, TitleAlignment, VerticalAlignment

Horizontal alignment of text objects w.r.t. their coordinates

Value Summary

`HorizontalAlignment`, `TitleAlignment`	Inherited	`Center`, `Left`, or `Right`
`VerticalAlignment`	Inherited	`BaseLine`, `Bottom`, `Center`, or `Top`

Graphics Primitives

Objects	Default Values

`plot::Text2d`, `plot::Text3d`	`HorizontalAlignment`: `Left`
	`TitleAlignment`: `Center`
	`VerticalAlignment`: `BaseLine`

`plot::Integral`	`HorizontalAlignment`: `Left`
	`TitleAlignment`: `Center`
	`VerticalAlignment`: `Bottom`

Description

`TitleAlignment` controls the interpretation of the `TitlePosition` of the titles of graphical objects.

`HorizontalAlignment` and `VerticalAlignment` control the interpretation of the coordinates of text objects.

Titles of graphical objects are placed at the position defined by `TitlePosition`. `TitleAlignment` determines whether the beginning, the center, or the end of the title text is aligned at this position. See “Example 1” on page 24-1684.

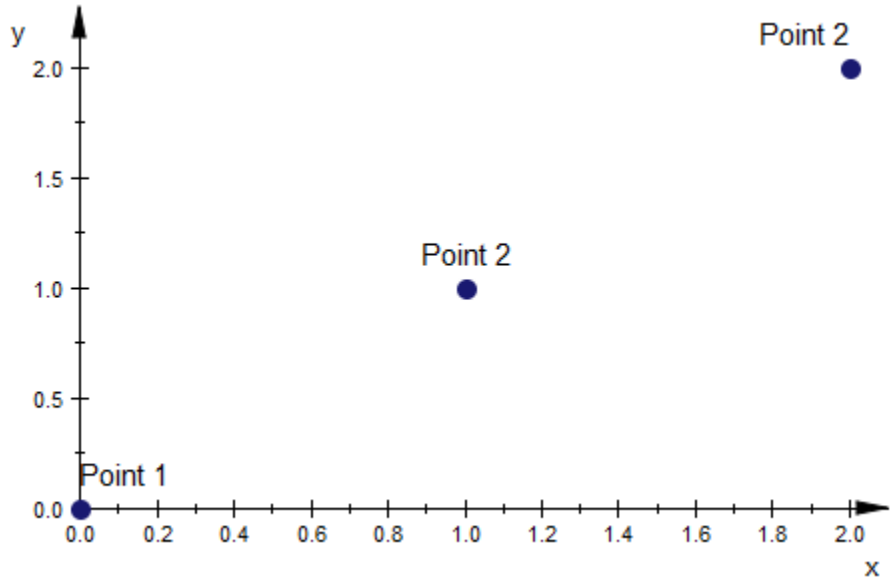
Text objects (i.e., objects of type `plot::Text2d` or `plot::Text3d`) carry, in their `Position` attribute, a position. `HorizontalAlignment` and `VerticalAlignment` together determine which point of the text this position refers to. For example, with `HorizontalAlignment = Left` and `HorizontalAlignment = Bottom`, the given position is the lower left corner of the rendered text.

Examples

Example 1

We plot three points with title positions 0.1 above each point. The titles are aligned such that the beginning of the text (`Left`), the center of the text (`Center`), or the end of the text (`Right`) is at the `TitlePosition`:

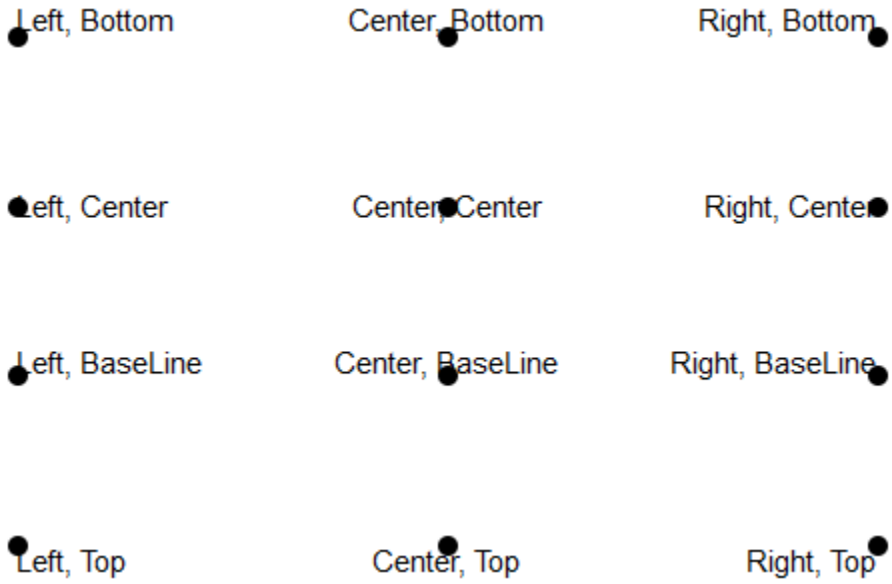
```
plot(plot::Point2d(0, 0, Title = "Point 1",
                   TitlePosition = [0, 0.1],
                   TitleAlignment = Left),
     plot::Point2d(1, 1, Title = "Point 2",
                   TitlePosition = [1, 1.1],
                   TitleAlignment = Center),
     plot::Point2d(2, 2, Title = "Point 2",
                   TitlePosition = [2, 2.1],
                   TitleAlignment = Right),
     PointSize = 2.5*unit::mm)
```



Example 2

The following call generates a table showing all the combinations of `HorizontalAlignment` and `VerticalAlignment`:

```
Hor := [Left, Center, Right]:
Vert := [Top, BaseLine, Center, Bottom]:
plot((plot::Text2d(expr2text(Hor[i], Vert[j]), [i, j],
                   HorizontalAlignment = Hor[i],
                   VerticalAlignment   = Vert[j]),
      plot::Point2d([i, j], Color = RGB::Black))
     $ i = 1..3 $ j = 1..4, Axes = None,
     TitleFont = [13], PointSize = 2.5*unit::mm)
```




```
delete Hor, Vert:
```

See Also

MuPAD Functions
`Position` | `Title` | `TitlePosition`

Legend

Makes a legend entry

Value Summary

Library wrapper for "`LegendText`, `LegendEntry`, and `LegendVisible`"	See below

Description

`Legend` makes a legend entry and activates the legend.

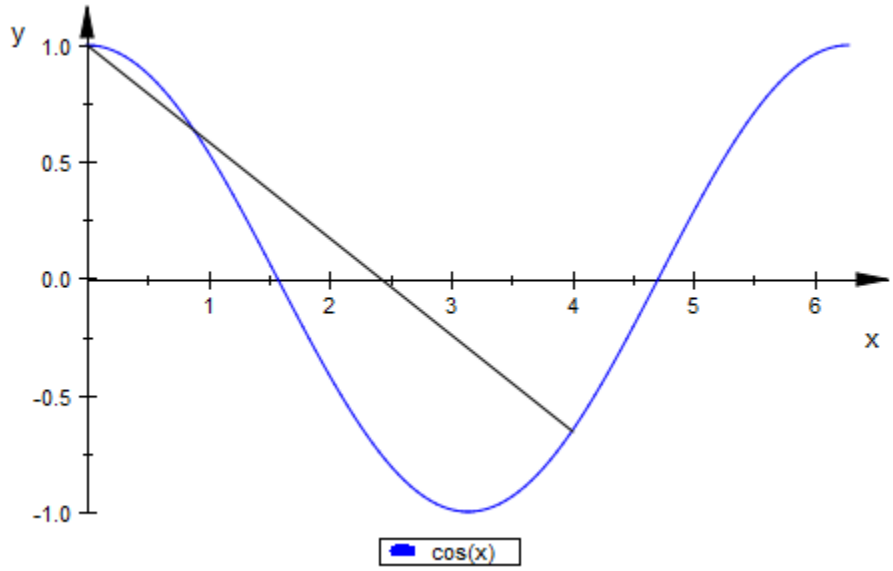
`Legend` is a library wrapper which sets a `LegendText` and simultaneously switches the legend on by setting `LegendEntry` and `LegendVisible` to `TRUE`.

Examples

Example 1

`Legend` is used to set a `LegendText` for the function and activate the legend. The line does not appear in the legend:

```
plot(plot::Function2d(cos(x), x = 0..2*PI, Legend = "cos(x)"),
     plot::Line2d([0, 1], [4, cos(4)], Color = RGB::Black))
```



See Also

MuPAD Functions

`LegendAlignment` | `LegendEntry` | `LegendPlacement` | `LegendText` | `LegendVisible`

More About

- "Legends"

LegendEntry

Add this object to the legend?

Value Summary

Inherited	FALSE, or TRUE

Description

`LegendEntry` turns legend entries of individual objects on and off, if `LegendVisible` is `TRUE`.

Note: `LegendEntry` is a technical internal attribute. You will most likely want to use the library interface attribute `Legend` in order to set legend entries.

If legends are active (i.e., `LegendVisible` is set to `TRUE`), `LegendEntry` controls which objects have entries in the legend. Only objects with `LegendEntry = TRUE` show up there.

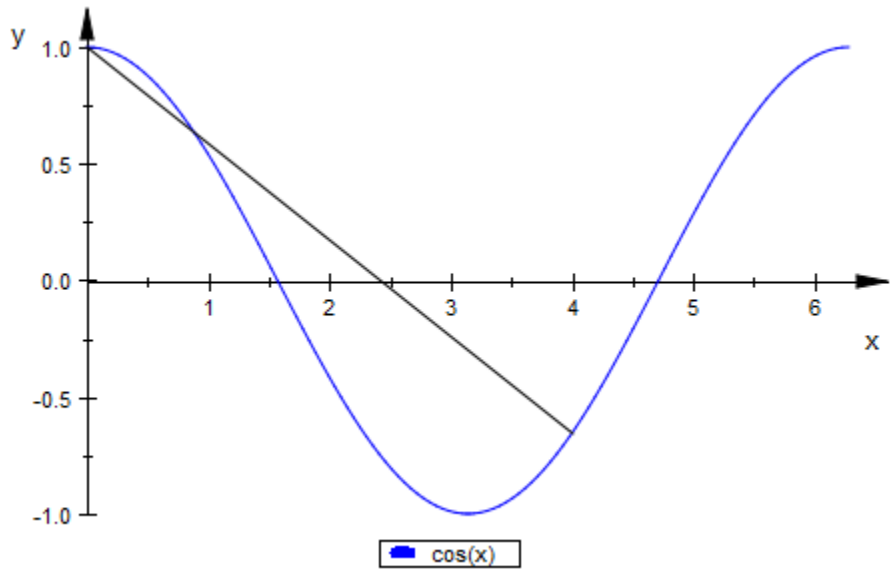
As long as `LegendVisible` has its default value of `FALSE`, `LegendEntry` has no effect whatsoever.

Examples

Example 1

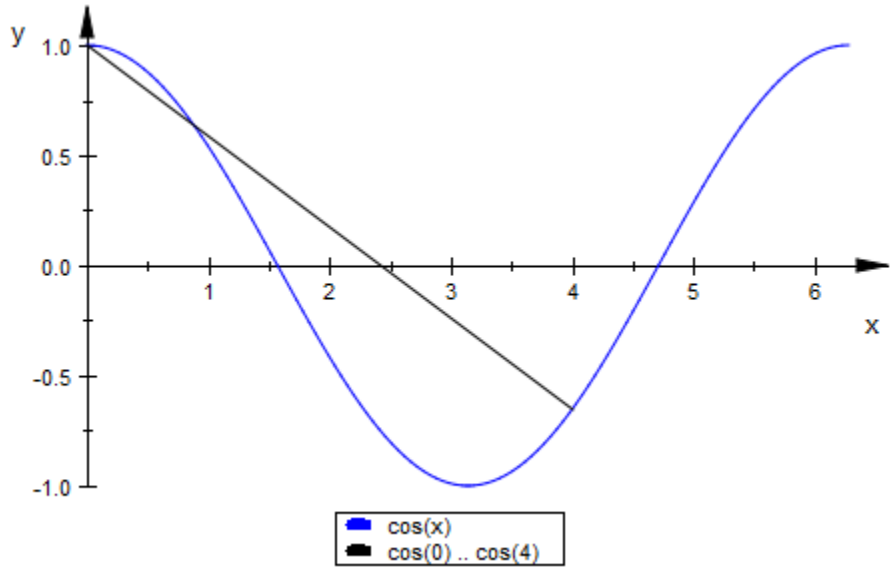
By default, functions have `LegendEntry = TRUE`, while, e.g., lines do not:

```
plot(plot::Function2d(cos(x), x = 0..2*PI, Name = "cos(x)"),
     plot::Line2d([0, 1], [4, cos(4)], Name = "cos(0) .. cos(4)",
                  Color = RGB::Black),
     LegendVisible)
```



Use `LegendEntry` to turn on the legend entry for the line:

```
plot(plot::Function2d(cos(x), x = 0..2*PI, Name = "cos(x)"),
     plot::Line2d([0, 1], [4, cos(4)], Name = "cos(0) .. cos(4)",
                  Color = RGB::Black, LegendEntry = TRUE),
     LegendVisible)
```



See Also

MuPAD Functions

`Legend` | `LegendAlignment` | `LegendPlacement` | `LegendText` | `LegendVisible`

More About

- "Legends"

LegendAlignment, LegendPlacement, LegendVisible

Legend at left, center, or right

Value Summary

`LegendAlignment`	Inherited	`Center`, `Left`, or `Right`
`LegendPlacement`	Inherited	`Bottom`, or `Top`
`LegendVisible`	Inherited	`FALSE`, or `TRUE`

Graphics Primitives

Objects	Default Values

`plot::Scene2d`, `plot::Scene3d`	`LegendAlignment`: `Center`
	
	`LegendPlacement`: `Bottom`
	
	`LegendVisible`: `FALSE`

Description

`LegendVisible` activates a legend identifying the individual objects in a plot.

`LegendAlignment` and `LegendPlacement` control the placement of this legend.

For complex plots with multiple objects, it is often helpful to include an explanation in form of a legend that states the connection from object color to object meaning.

The entry for "object meaning" is usually not provided automatically but must be given using `LegendText`. As an exception, `plotfunc2d` and `plotfunc3d` set the function terms as "meaning". Cf. "Example 1" on page 24-1693.

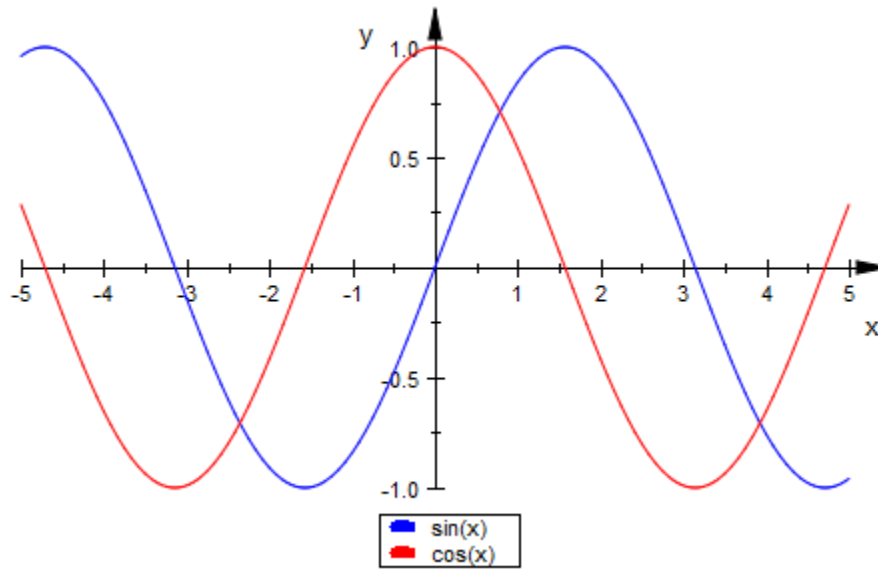
Using `LegendPlacement`, the legend can be moved from below the plot to above it. `LegendAlignment` controls whether the legend is displayed flush left, flush right, or centered (which is the default).

Examples

Example 1

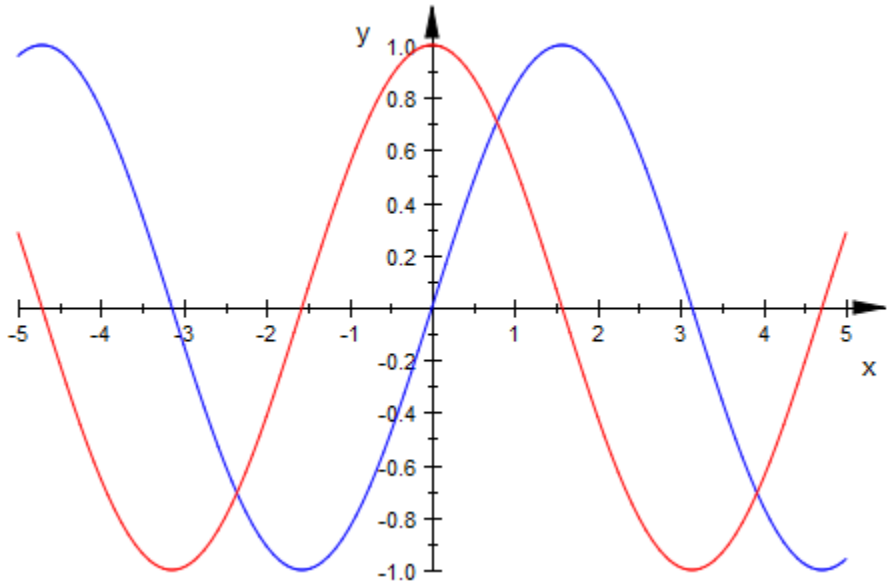
When plotting more than one object, `plotfunc2d` and `plotfunc3d` set `LegendVisible = TRUE`:

```
plotfunc2d(sin(x), cos(x))
```



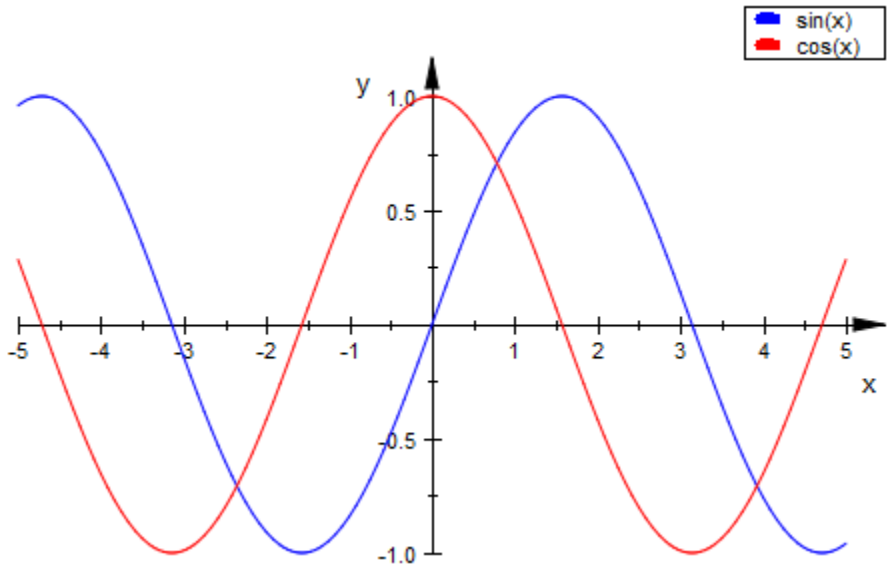
It is possible to explicitly switch this automatic legend off:

```
plotfunc2d(sin(x), cos(x), LegendVisible = FALSE)
```



Using `LegendPlacement` and `LegendAlignment`, we place the legend in the upper right corner of the graphics:

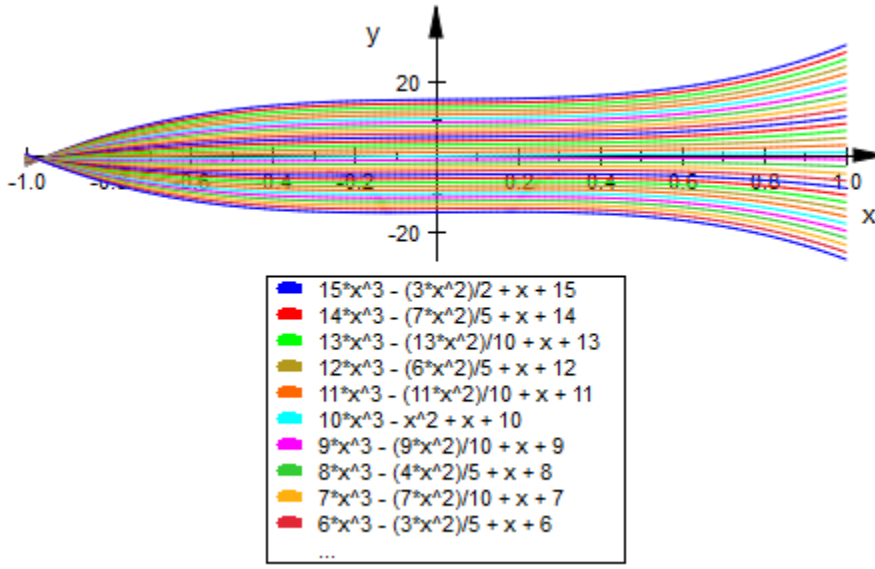
```
plotfunc2d(sin(x), cos(x),
           LegendPlacement = Top, LegendAlignment = Right)
```



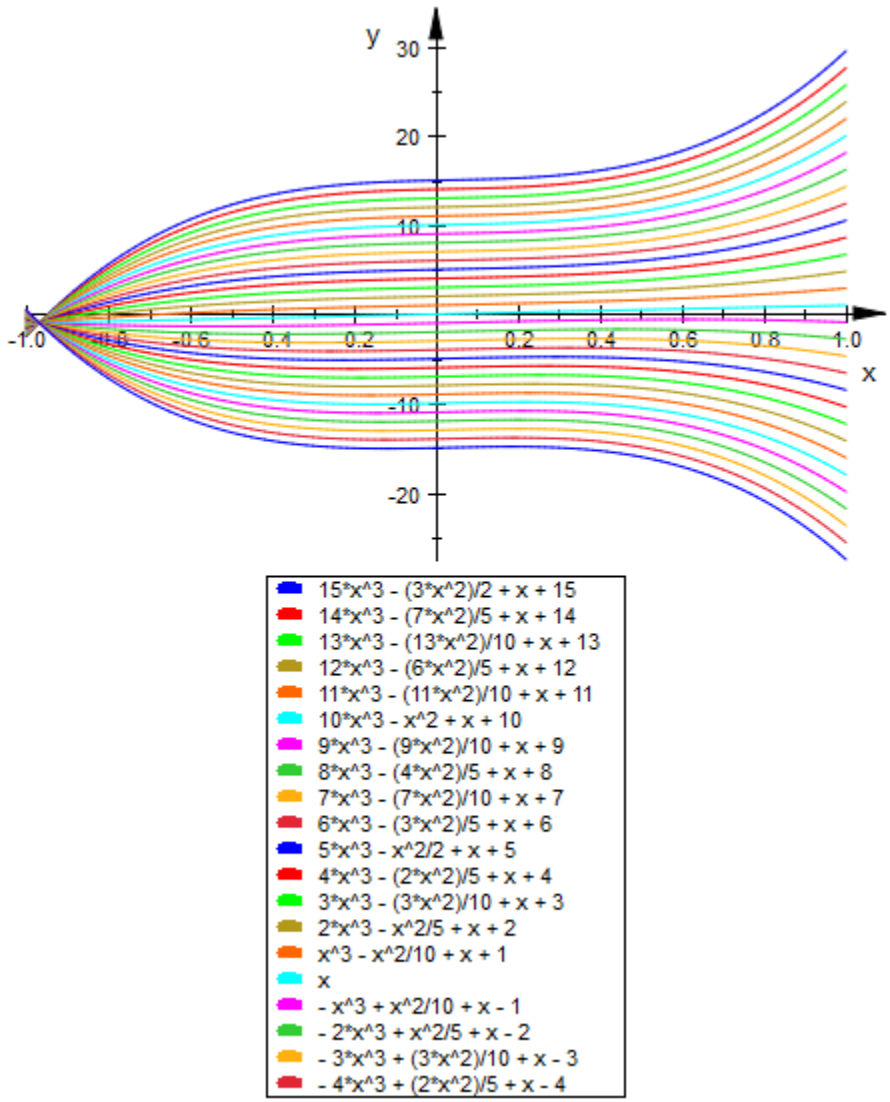
Example 2

When plotting many objects with active legend entries, the legend is abbreviated: It will never take up more than half of the scene height and it will not contain more than 20 entries:

```
plotfunc2d(-i*x^3+i/10*x^2+x-i $ i = -15..15, x=-1..1)
```




```
plotfunc2d(-i*x^3+i/10*x^2+x-i $ i = -15..15, x=-1..1,
           Height = 15*unit::cm)
```


y
30
20
10
-10
-20
-1.0
-0.6
-0.4
-0.2
0.2
0.4
0.6
0.8
1.0
x
15*x^3 - (3*x^2)/2 + x + 15
14*x^3 - (7*x^2)/5 + x + 14
13*x^3 - (13*x^2)/10 + x + 13
12*x^3 - (6*x^2)/5 + x + 12
11*x^3 - (11*x^2)/10 + x + 11
10*x^3 - x^2 + x + 10
9*x^3 - (9*x^2)/10 + x + 9
8*x^3 - (4*x^2)/5 + x + 8
7*x^3 - (7*x^2)/10 + x + 7
6*x^3 - (3*x^2)/5 + x + 6
5*x^3 - x^2/2 + x + 5
4*x^3 - (2*x^2)/5 + x + 4
3*x^3 - (3*x^2)/10 + x + 3
2*x^3 - x^2/5 + x + 2
x^3 - x^2/10 + x + 1
x
- x^3 + x^2/10 + x - 1
- 2*x^3 + x^2/5 + x - 2
- 3*x^3 + (3*x^2)/10 + x - 3
- 4*x^3 + (2*x^2)/5 + x - 4

See Also

MuPAD Functions
`Legend` | `LegendFont` | `LegendText`

LegendText

Short explanatory text for legend

Value Summary

Optional	Text string

Description

`LegendText` sets the text for the legend entry of an object.

Note: `LegendText` is a technical internal attribute. You will most likely want to use the library interface attribute `Legend` in order to set legend entries.

To have a legend entry, the object must have `Legend` set to `TRUE` and `LegendVisible` must be `TRUE` for the enclosing scene. Cf. “Example 1” on page 24-1699.

If `LegendText` is unset, but `Legend` and `LegendVisible` are `TRUE`, the legend entry is taken from the attribute `Name`. If that is unset, too, the name of the object type is displayed. Cf. “Example 2” on page 24-1701.

Examples

Example 1

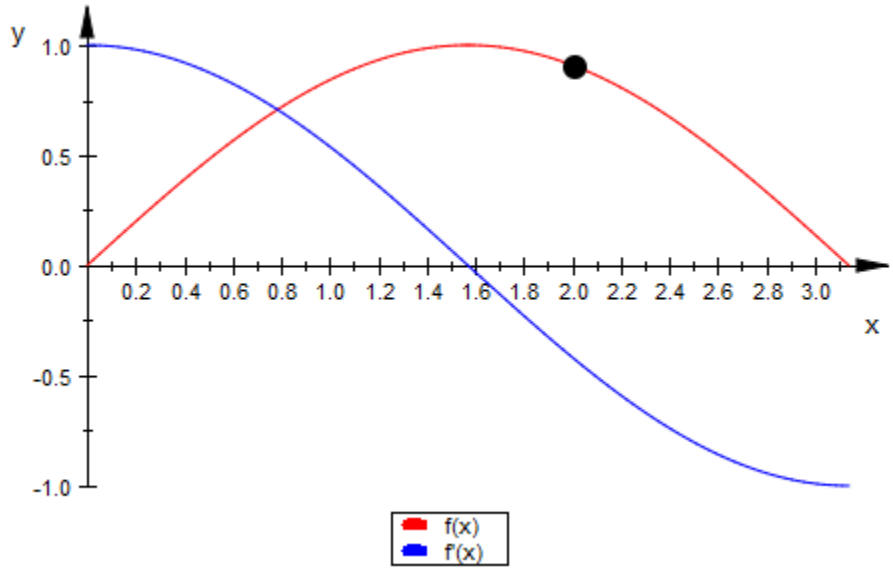
We create a few objects with values for `LegendText` set:

```
f := plot::Function2d(sin(x), x = 0..PI,
                      LegendText = "f(x)", Color = RGB::Red):
g := plot::Function2d(cos(x), x = 0..PI,
                      LegendText = "f'(x)", Color = RGB::Blue):
```

```
p := plot::Point2d([2, sin(2)], PointSize = 3*unit::mm,
                   LegendText = "(2; f(2))", Color = RGB::Black):
```

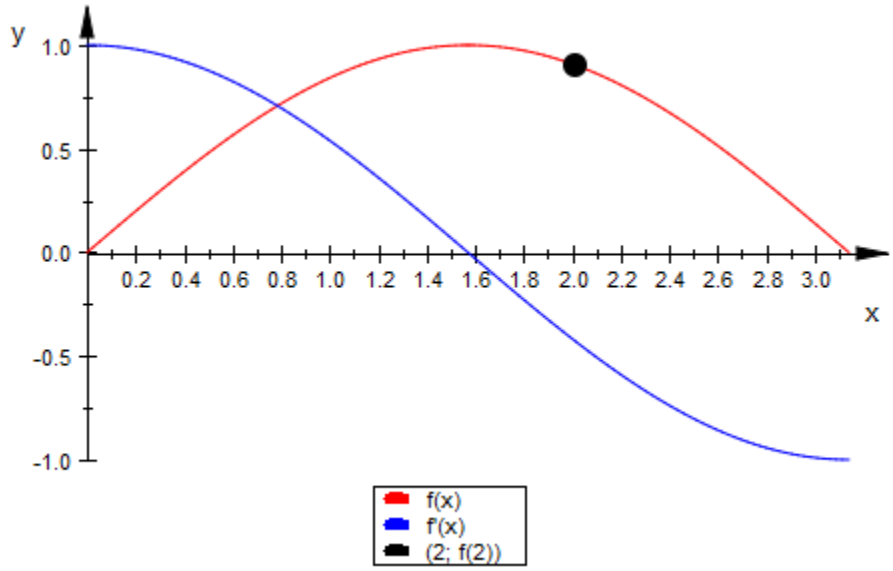
Switching on the legend, we plot these objects:

```
plot(f, g, p, LegendVisible = TRUE)
```



As we can see, only the function objects show up in the legend. If `p` is supposed to be shown there, too, we must explicitly set `LegendEntry` to `TRUE`:

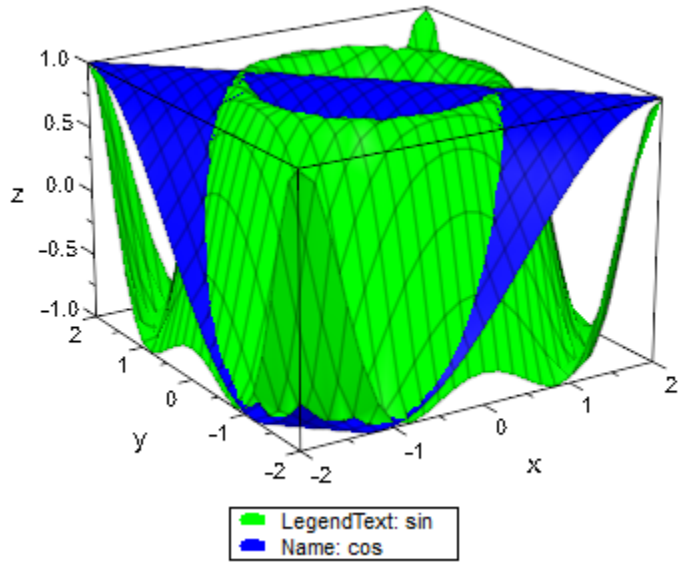
```
p::LegendEntry := TRUE:
plot(f, g, p, LegendVisible = TRUE)
```



Example 2

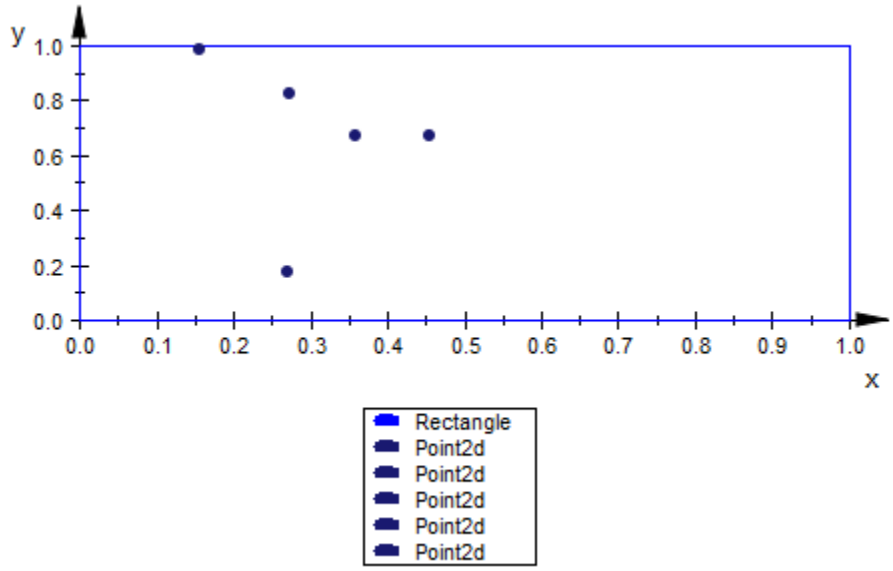
If an object has a legend entry, but `LegendText` is not set, the first fall-back is the `Name` attribute of the object:

```
plot(plot::Function3d(sin(x^2 + y^2), x = -2..2, y = -2..2,
                      Color = RGB::Green, FillColorType = Flat,
                      LegendText = "LegendText: sin",
                      Name = "Name: sin"),
     plot::Function3d(cos(x + y), x = -2..2, y = -2..2,
                      Color = RGB::Blue, FillColorType = Flat,
                      Name = "Name: cos"),
     LegendVisible)
```



As a last resort, the name of the type of object is used:

```
plot(plot::Rectangle(0..1, 0..1),
     plot::Point2d(frandom(), frandom()),
     plot::Point2d(frandom(), frandom()),
     plot::Point2d(frandom(), frandom()),
     plot::Point2d(frandom(), frandom()),
     plot::Point2d(frandom(), frandom()),
     LegendEntry = TRUE, LegendVisible = TRUE)
```



See Also

MuPAD Functions

`Legend` | `LegendVisible` | `Name`

More About

- "Legends"

ShowInfo

Information about integral approximation

Value Summary

Optional	List of arithmetical expressions

Graphics Primitives

Objects	ShowInfo Default Values
`plot::Integral`	[2, `IntMethod`, `Integral`]

Description

`ShowInfo` defines the text information displayed by `plot::Integral`.

In `plot::Integral`, text information about the used approximation method, the values of the approximation and the exact integral, the number of subintervals and the error of the approximation can be displayed within the approximation object.

The attribute is specified by `ShowInfo = [entry`$_1$`, entry`$_2$`, …]` with a list of various entries. The user may specify the entries in arbitrary order.

If the list is empty, no text information is displayed.

Each entry in the list can be of one of the following types:

- an arbitrary string

  In the text, this entry is appended to the current line. No white space or line break is prepended or appended. The string itself, however, may contain white space or a line break (given by `\n`).
- "" (empty string)

  This inserts an empty line in the text.
- `IntMethod`

In the text, this creates a new line

`name: float_value`,

where `name` is the value of the attribute `IntMethod` and `float_value` is the numerical value of the integral approximation. This value is computed internally and inserted in the text, automatically.

- `IntMethod` = name

  In the text, this creates a new line

  `name: float_value`,

  where `float_value` is the numerical value of the integral approximation.

  If `name` is one of the flags `RiemannLower` etc. listed on the help page of the attribute `IntMethod`, this flag is displayed in the text.

  Alternatively, `name` may be a string. When `name` is the empty string `""`, only the numerical approximation of the integral value is displayed.

- `Integral`

  In the text, this creates a new line

  `Integral: float_value`,

  where `float_value` is a high precision float approximation of the exact integral value.

- `Integral` = string

  In the text, this creates a new line

  `string: float_value`,

  where `string` is an arbitrary text string and `float_value` is a high precision float approximation of the exact integral value .

  When `string` is the empty string `""`, only the high precision approximation `float_value` is displayed.

- `Error`

In the text, this creates a new line

`Error: float_value`,

where `float_value` is the absolute difference between the numerical value obtained by the chosen approximation method and a high precision float approximation of the exact integral value.

- `Error` = string

  In the text, this creates a new line

  `string: float_value`,

  where `string` is an arbitrary text string and `float_value` is the absolute difference between the numerical value obtained by the chosen approximation method and a high precision float approximation of the exact integral value.

  When `string` is the empty string `""`, only the absolute quadrature error `float_value` is displayed.

- `Nodes`

  In the text, this creates a new line

  `Nodes: n`,

  where the integer `n` is the number of intervals used for the integral approximation.

- `Nodes` = string

  In the text, this creates a new line

  `string: n`,

  where `string` is an arbitrary text string and the integer `n` is the number of intervals used for the integral approximation.

  When `string` is the empty string `""`, only the integer `n` is displayed.

- `Position = [X, Y]`

  This entry determines the position of the text information. `X` and `Y` are the coordinates of the anchor point of the text. The alignment of the text with respect

to the anchor point can be chosen by the attributes `HorizontalAlignment` and `VerticalAlignment`.

- a positive integer `digits`

  The integer `digits` determines the number of digits after the decimal point for all following float values.

  Different float values can be displayed with a different number of digits by inserting several `digits` entries at appropriate positions in the list.

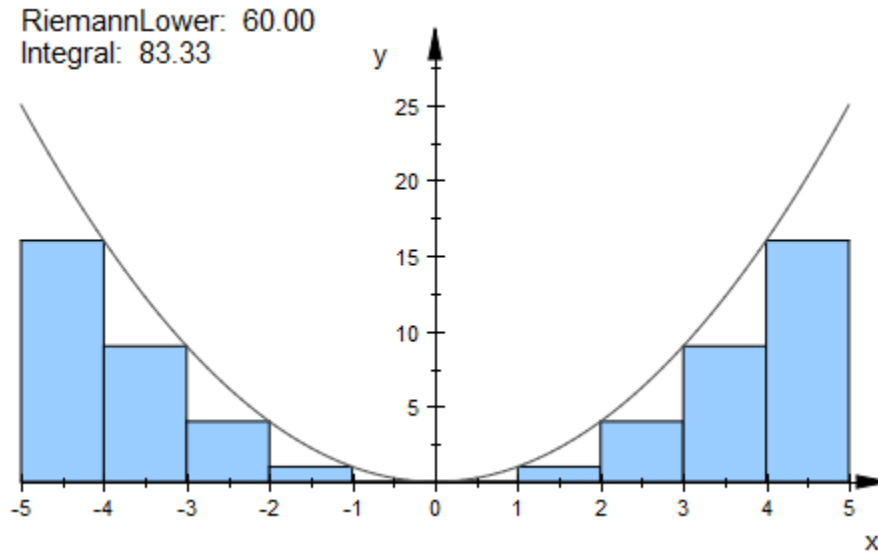
Without an explicit specification of `Position = [X, Y]`, the text is positioned automatically.

Examples

Example 1

By default, the approximation method, the value of approximation and the integral is displayed with 2 digits after the decimal point:

```
f := plot::Function2d(x^2, x = -5..5, Color = RGB::DarkGrey):
plot(plot::Integral(f, IntMethod = RiemannLower), f)
```

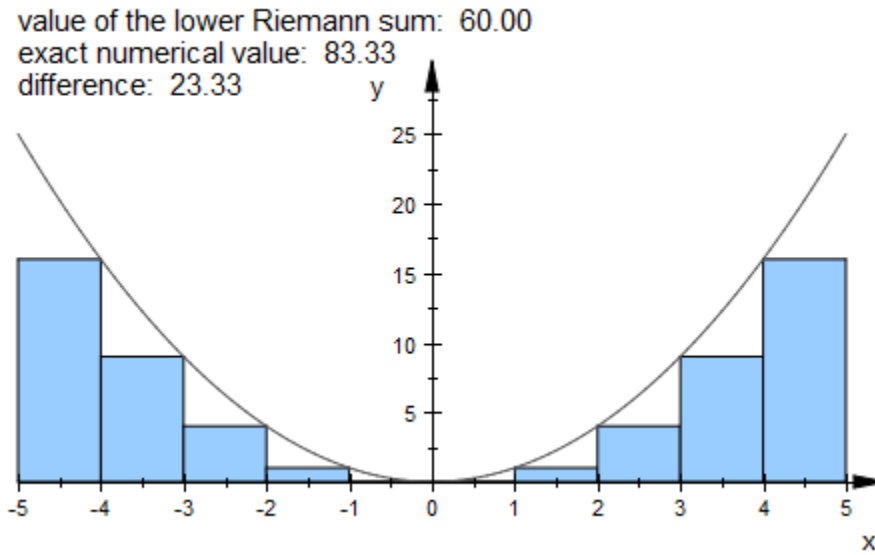


This call is equivalent to:

```
plot(plot::Integral(f, IntMethod = RiemannLower,
                    ShowInfo = [2, IntMethod, Integral]), f):
```

The text can be changed:

```
plot(plot::Integral(f, IntMethod = RiemannLower,
       ShowInfo = [IntMethod = "value of the lower Riemann sum",
                   Integral = "exact numerical value",
                   Error = "difference"]), f)
```

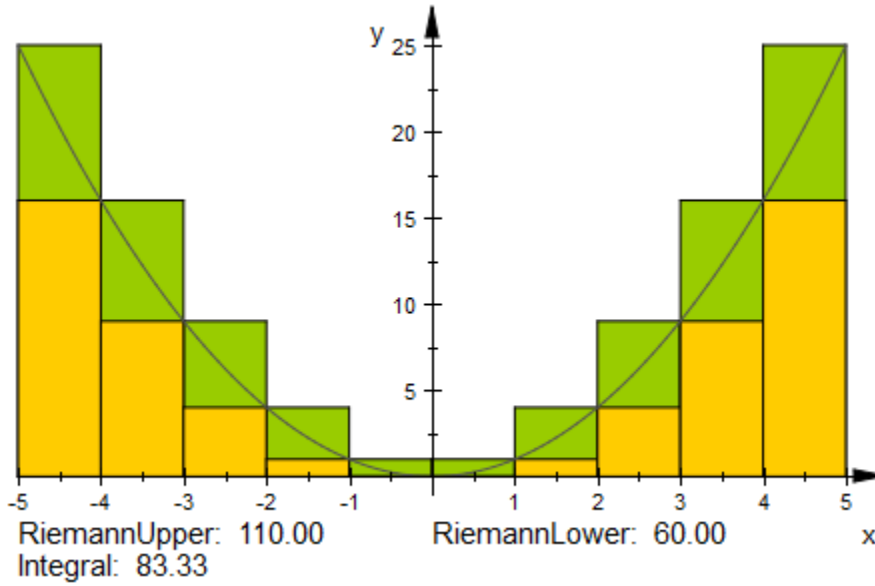



```
delete f:
```

Example 2

The position can be specified explicitely. In this case, the entries to be displayed must be specified explicitely, too. The text attribute `VerticalAlignment` aligns the text object:

```
f := plot::Function2d(x^2, x = -5..5, Color = RGB::DarkGrey):
plot(plot::Integral(f, IntMethod = RiemannUpper, Color = RGB::Lime,
                       ShowInfo = [IntMethod, Integral,
                                   Position = [-5, -1]],
                       VerticalAlignment = Top),
     plot::Integral(f, IntMethod = RiemannLower, Color = RGB::Gold,
                       ShowInfo = [IntMethod,
                                   Position = [0, -1]],
                       VerticalAlignment = Top),
     f)
```

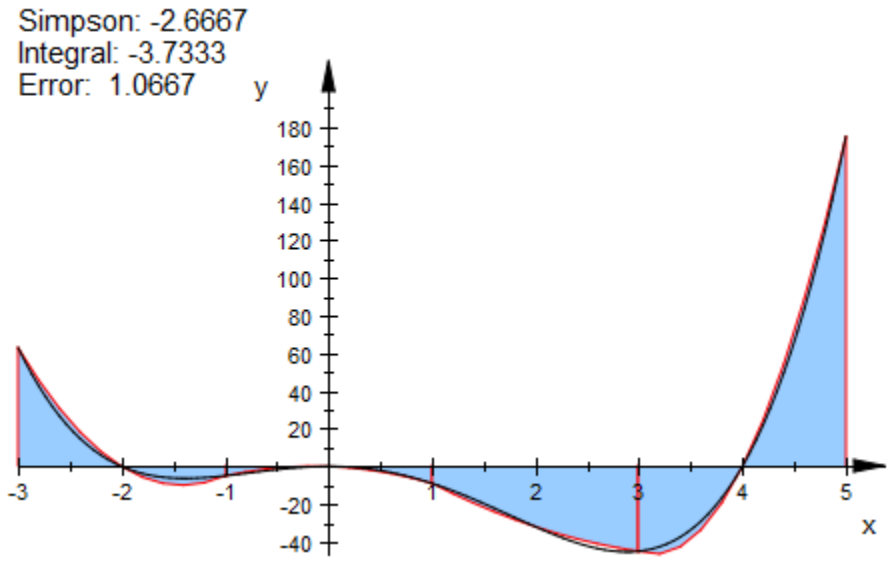



```
delete f:
```

Example 3

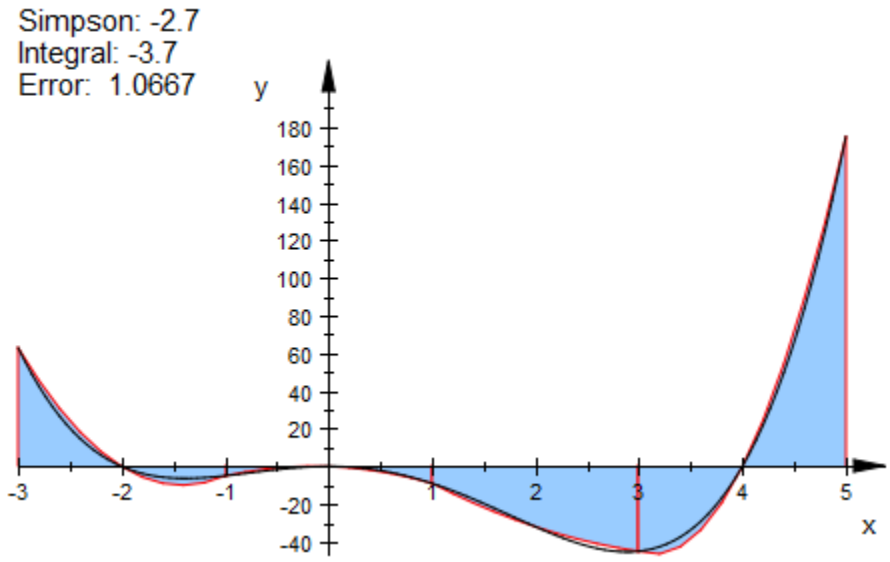
The number of digits after the decimal point can be specified for each value. In the following example all values are displayed with four digits:

```
f := plot::Function2d(x^2*(x-4)*(x+2), x = -3..5,
                      Color = RGB::Black):
plot(plot::Integral(f, 4, IntMethod = Simpson,
                    LineColor = RGB::Red,
                    ShowInfo = [4, IntMethod, Integral, Error]),
     f)
```



Only the error shall be displayed with four digits after the decimal point. All other values are shown with only one digit:

```
plot(plot::Integral(f, 4, IntMethod = Simpson, LineColor = RGB::Red,
                    ShowInfo = [1, IntMethod, Integral, 4, Error]),
     f)
```

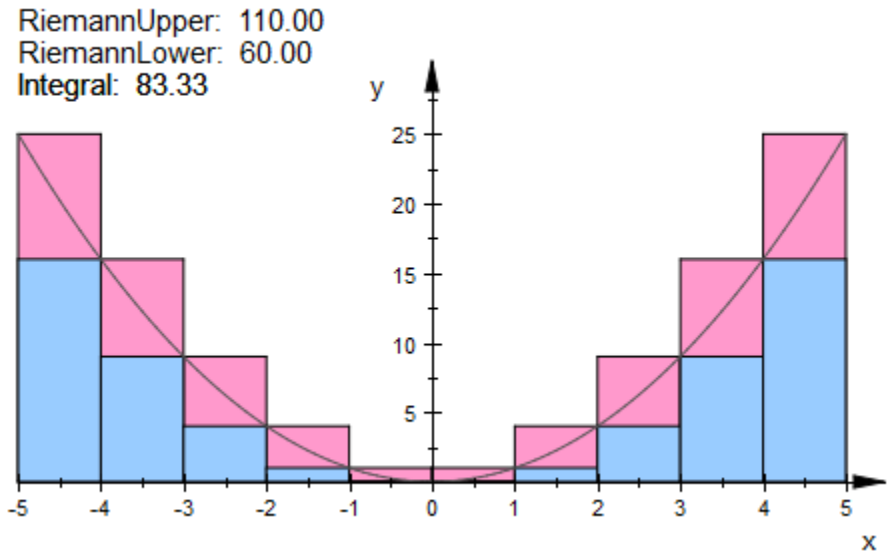



```
delete f:
```

Example 4

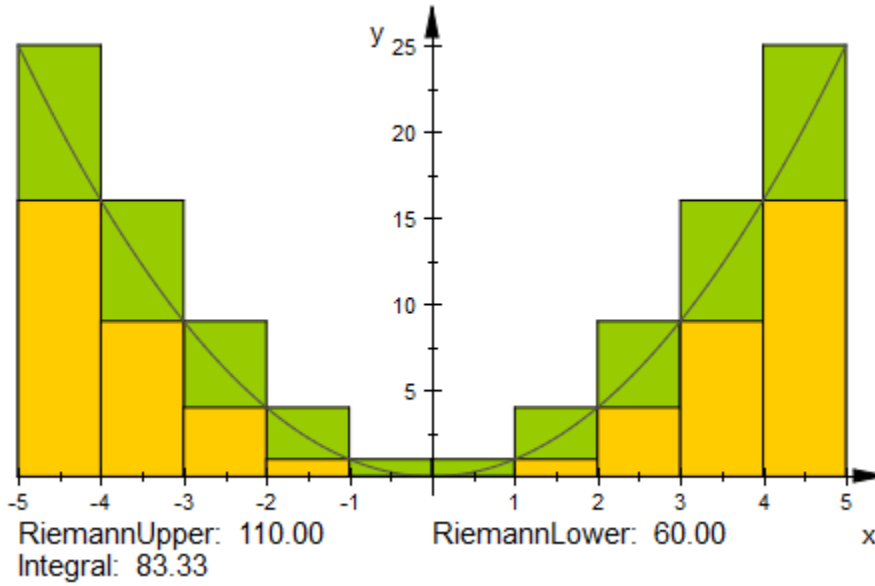
Two approximation objects shall be displayed in one plot. To prevent collision of the automatically positioned texts, we insert an empty line into the text of one of the objects to prevent collision of the automatically positioned texts:

```
f := plot::Function2d(x^2, x = -5..5, Color = RGB::DarkGrey):
plot(plot::Integral(f, IntMethod = RiemannUpper, Color = RGB::Rose,
                       ShowInfo = [IntMethod, "", Integral]),
     plot::Integral(f, IntMethod = RiemannLower,
                       ShowInfo = [IntMethod, Integral]),
     f)
```



Alternatively, the position can be given explicitly:

```
f := plot::Function2d(x^2, x = -5..5, Color = RGB::DarkGrey):
plot(plot::Integral(f, IntMethod = RiemannUpper, Color = RGB::Lime,
                       ShowInfo = [IntMethod, Integral,
                                   Position = [-5, -1]],
                       VerticalAlignment = Top),
     plot::Integral(f, IntMethod = RiemannLower, Color = RGB::Gold,
                       ShowInfo = [IntMethod,
                                   Position = [0, -1]],
                       VerticalAlignment = Top),
     f)
```

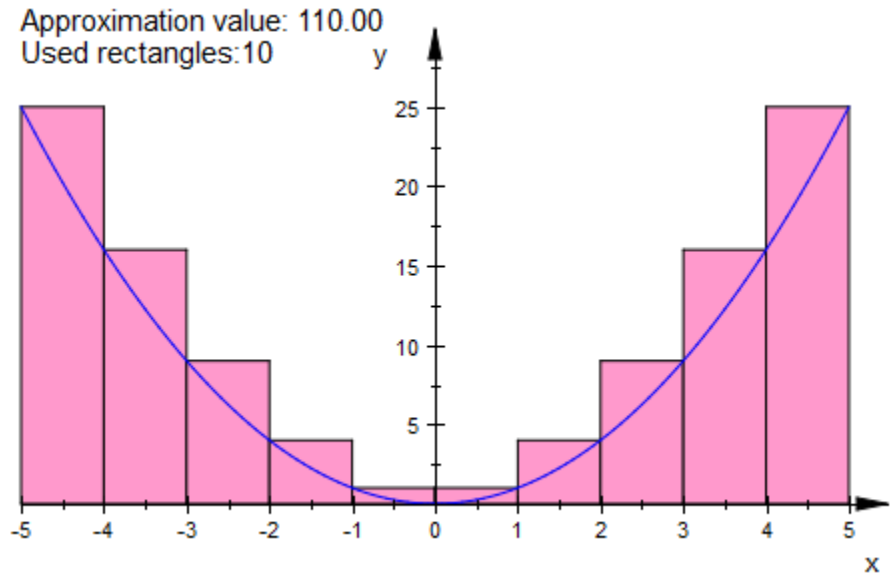



```
delete f:
```

Example 5

The text may contain additional messages:

```
f := plot::Function2d(x^2, x = -5..5):
plot(plot::Integral(f, IntMethod = RiemannUpper,
        ShowInfo = ["Approximation value:", IntMethod = "",
                    "\nUsed rectangles:", Nodes = ""],
        Color = RGB::Rose),
     f)
```




```
delete f:
```

See Also

MuPAD Functions

`HorizontalAlignment` | `IntMethod` | `TextFont` | `TextRotation` | `VerticalAlignment`

Title, Titles

Object title

Value Summary

`Title`, `Titles`	Optional	Text string

Graphics Primitives

Objects	Default Values
`plot::Piechart2d`, `plot::Piechart3d`	`Titles`: `[" "]`

Description

`Title` sets the title of an object to be displayed in the graphics.

`Titles` is a list of titles for parts of an object, e.g., the pieces of a pie chart.

Using `Title`, any graphical object can be given a title that will be displayed at the position given by the `TitlePosition` attribute.

The `Title` can additionally be horizontically aligned at the `TitlePosition` via `TitleAlignment`.

The object attribute `Visible` also affects the object's title: Invisible objects do not show their titles.

`Titles` is used to set a number of titles for sub-objects, such as the bars of a bar plot or the segments of a pie chart. These do not react to `TitlePosition`.

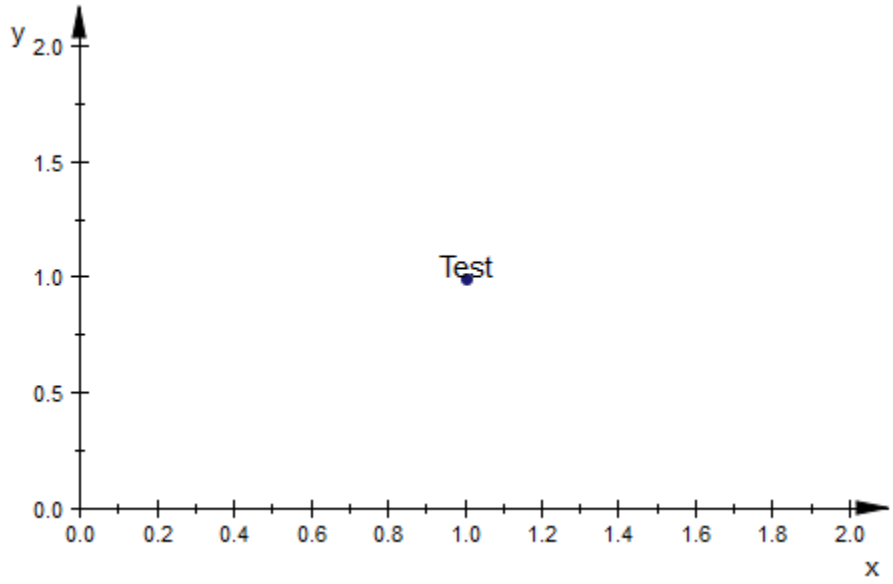
`Title` and `Titles` cannot be animated. But note that `TitlePosition` can.

Examples

Example 1

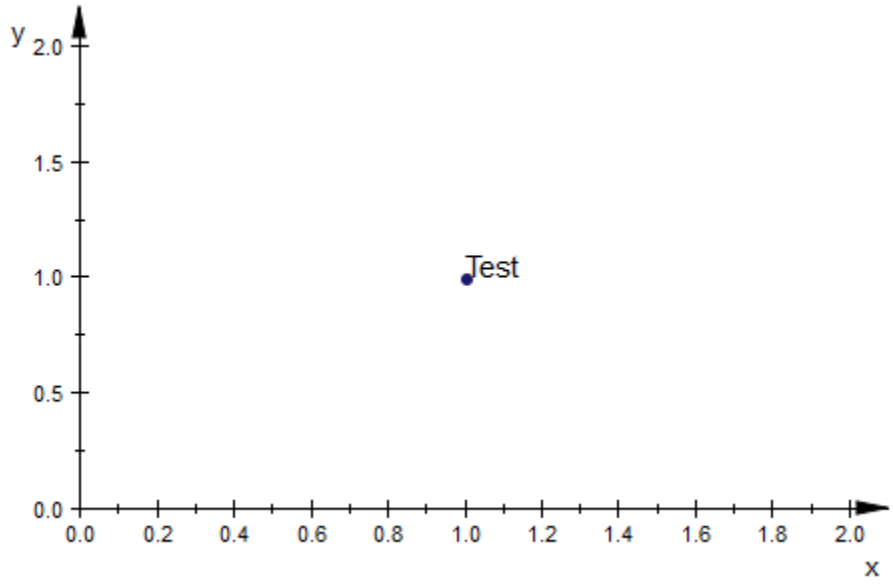
The default positioning of a title relative to `TitlePosition` is to have the lower left corner of the title at this place:

```
plot(plot::Point2d(1, 1, Title = "Test",
                   TitlePosition = [1, 1]))
```



This position depends on the title alignment:

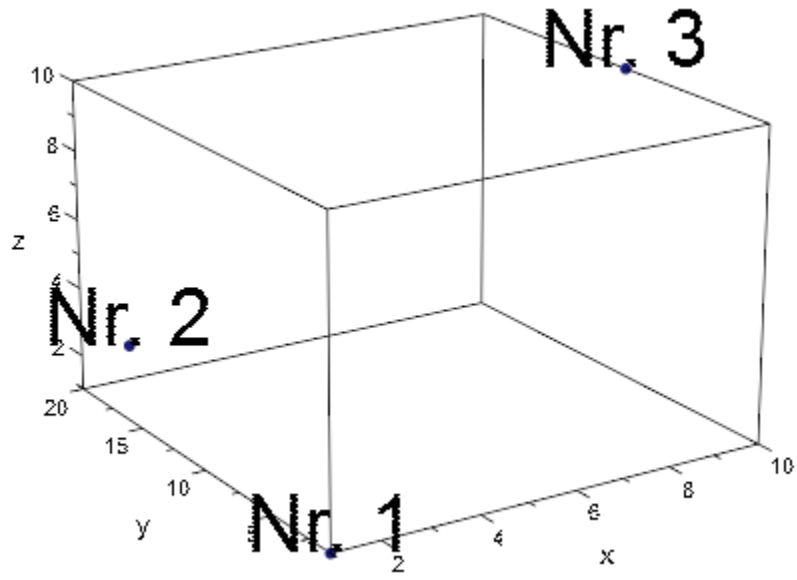
```
plot(plot::Point2d(1, 1, Title = "Test",
                   TitlePosition = [1, 1],
                   TitleAlignment = Left))
```



Example 2

In 3D, titles have so-called “bill-boarding”: Instead of having a fixed orientation, they are always drawn in a readable orientation and their sizes are not affected by zooming and perspective scaling:

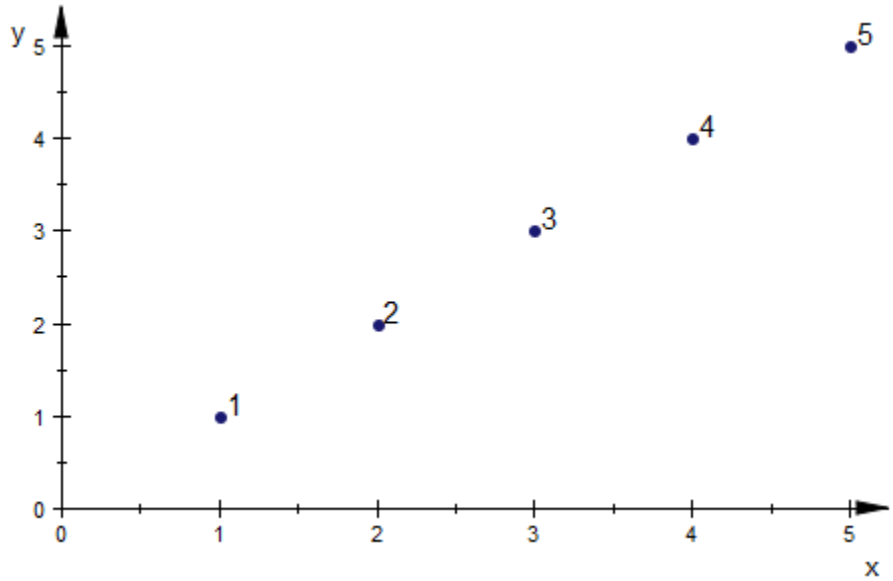
```
plot(plot::Point3d(1, 1, 1,
                   Title = "Nr. 1",
                   TitlePosition = [1, 1, 1]),
     plot::Point3d(2, 20, 2,
                   Title = "Nr. 2",
                   TitlePosition = [2, 20, 2]),
     plot::Point3d(10, 10, 10,
                   Title = "Nr. 3",
                   TitlePosition = [10, 10, 10]),
     TitleFont = [30])
```



Example 3

Titles of invisible objects are invisible themselves. This also applies to objects that are temporarily invisible:

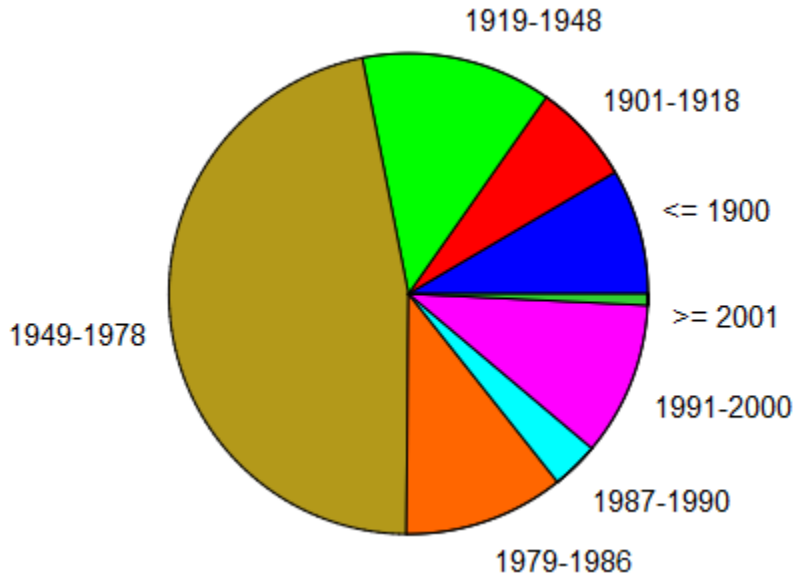
```
plot(plot::Point2d(i, i,
          Title = expr2text(i), TitlePosition = [i+1/10, i],
          VisibleAfter = i) $ i = 1..5,
     TimeRange = 0..5,
     ViewingBox = [0..5, 0..5])
```



Example 4

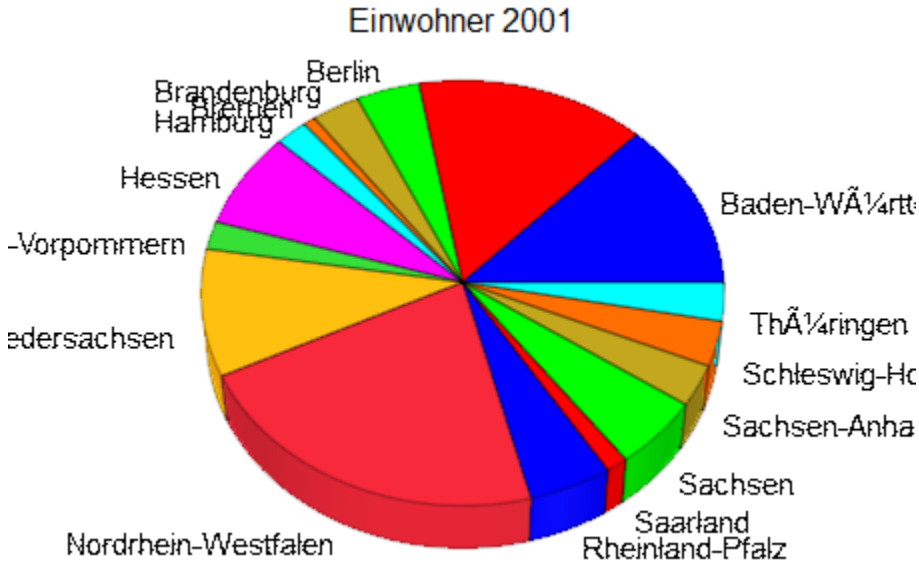
Use `Titles` to label individual parts of statistical plots such as pie charts:

```
plot(plot::Piechart2d([3267, 2629, 4970, 18094,
                       4189, 1236, 4003, 297],
                      Titles = ["<= 1900", "1901-1918",
                                "1919-1948", "1949-1978",
                                "1979-1986", "1987-1990",
                                "1991-2000", ">= 2001"]))
```

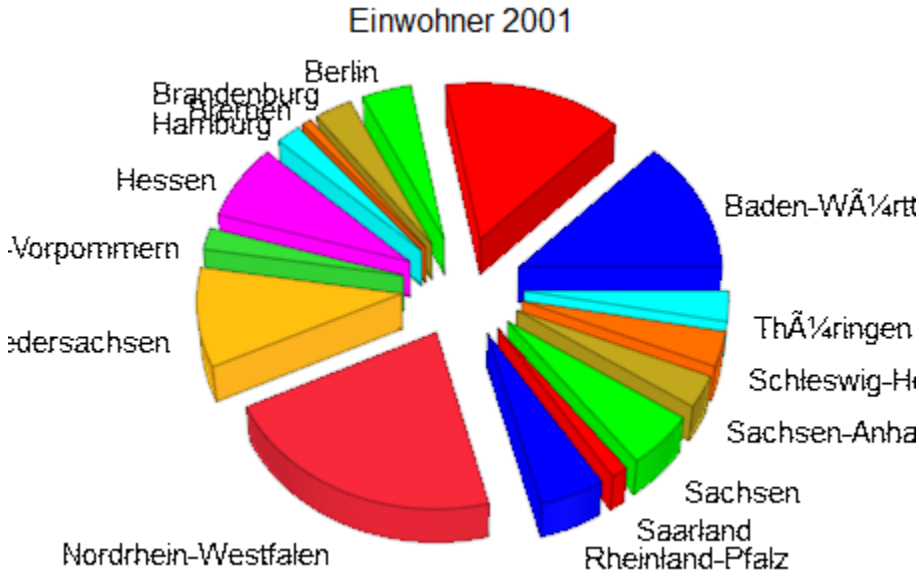


Note that pie charts with many pies are in general tricky to label nicely:

```
p := plot::Piechart3d([10601, 12330, 3388, 2593, 660, 1726,
                       6078, 1760, 7956, 18052, 4049, 1066,
                       4384, 2581, 2804, 2411],
             Titles = ["Baden-Württemberg", "Bayern", "Berlin",
                       "Brandenburg", "Bremen",
                       "Hamburg", "Hessen",
                       "Mecklenburg-Vorpommern",
                       "Niedersachsen", "Nordrhein-Westfalen",
                       "Rheinland-Pfalz", "Saarland", "Sachsen",
                       "Sachsen-Anhalt", "Schleswig-Holstein",
                       "Thüringen"]):
plot(p, Header = "Einwohner 2001")
```




```
p::Moves := [0.3]:
plot(p, Header = "Einwohner 2001")
```



See Also

MuPAD Functions

`TitleAlignment` | `TitleFont` | `TitlePosition`

TitlePosition, TitlePositionX, TitlePositionY, TitlePositionZ

Position of object titles

Value Summary

`TitlePosition`	Library wrapper for "[`TitlePositionX`, `TitlePositionY`]" (2D), "[`TitlePositionX`, `TitlePositionY`, `TitlePositionZ`]" (3D)	See below
`TitlePositionX`, `TitlePositionY`, `TitlePositionZ`	Optional	MuPAD expression

Description

`TitlePosition` sets the position where the object title is displayed.

`TitlePositionX`, `TitlePositionY`, and `TitlePositionZ` refer to the individual components of `TitlePosition`.

An object can be given a title to be displayed in the graphic with the attribute `Title`. `TitlePosition`, `TitlePositionX`, `TitlePositionY`, `TitlePositionZ` determines the position of this title.

`TitlePosition`, `TitlePositionX`, `TitlePositionY`, `TitlePositionZ` determines the anchor point of the title, which is in 3D displayed in "bill-boarding mode," which means that the text will always face the observer and will always be displayed in the same size, regardless of zooming or perspective. The alignment of the text w.r.t. the anchor point is further determined by the setting of `TitleAlignment`, cf. "Example 1" on page 24-1725.

MuPAD does not have automatic positioning of titles; to have a title properly positioned, `TitlePosition` must be set.

Examples

Example 1

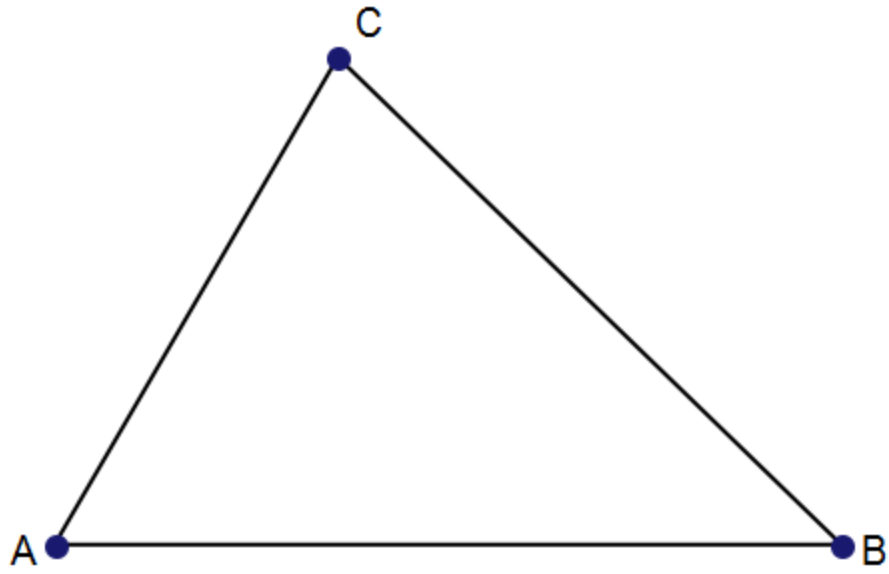
We plot three points with titles attached to them, changing the alignment. For demonstration purposes, the title positions coincide with the points:

```
plot(plot::Point2d(0, 3, Title = "Left",   TitlePosition = [0, 3],
                   TitleAlignment = Left),
     plot::Point2d(0, 2, Title = "Center", TitlePosition = [0, 2],
                   TitleAlignment = Center),
     plot::Point2d(0, 1, Title = "Right",  TitlePosition = [0, 1],
                   TitleAlignment = Right),
     Axes = None, PointSize = 2.0*unit::mm,
     ViewingBox = [-1..1, 0..4])
```

Example 2

A more realistic placement of titles is to separate them slightly from the points:

```
plot(plot::Polygon2d([[0, 0], [5, 0], [9/5, 12/5]], Closed),
     plot::Point2d([0, 0], Title = "A",
                   TitlePosition = [-0.2, -0.1]),
     plot::Point2d([5, 0], Title = "B",
                    TitlePosition = [5.2, -0.1]),
     plot::Point2d([9/5, 12/5], Title = "C",
                    TitlePosition = [2.0, 2.5]),
     Axes = None, TitleFont = [15], LineColor = RGB::Black,
     LineWidth = 0.5*unit::mm, PointSize = 3*unit::mm)
```



See Also

MuPAD Functions

`Title` | `TitleAlignment` | `TitleFont`

Bottom, Left

Positioning of a scene in a canvas

Value Summary

`Bottom`, `Left`	Optional	See below

Graphics Primitives

Objects	Default Values
`plot::Scene2d`, `plot::Scene3d`	`Bottom`, `Left`: `0`

Description

With the canvas attribute `Layout` set to `Absolute` or `Relative`, scenes in the canvas can be scaled and positioned freely.

`Bottom = b` places the bottom side of a scene at a distance *b* above the bottom side of the canvas.

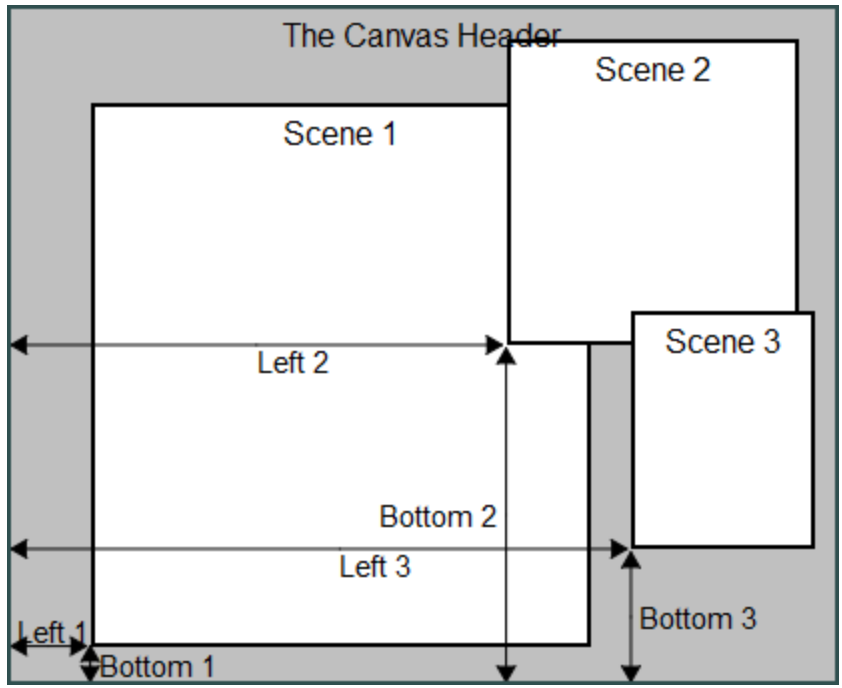
`Left = l` places the left hand side of a scene at a distance *l* to the right of the left hand side of the canvas.

The automatic layout schemes `Layout` = `Horizontal`, `Layout` = `Vertical`, and `Layout` = `Tabular` are available for a canvas that contains several scenes.

The canvas settings `Layout` = `Absolute` and `Layout` = `Relative` switch the automatic layout mode off and allow to position each scene freely via the attributes `Bottom` and `Left`, respectively. These attributes set the distances of the lower left corner of the scene to the bottom, respectively left hand side of the canvas. These values can be set separately for each scene.

Note: `Bottom` and `Left` are only respected for plots with `Layout` = `Absolute` or `Layout` = `Relative`.

The following pictures illustrates the positioning of scenes in a canvas via the scene attributes `Bottom` and `Left`:



With `Layout` = `Absolute`, the distance of the lower left corner of the scene to the lower left corner of the canvas must be specified via physical lengths with a unit, e.g., `Bottom = 2*unit::mm`, `Left = 0.1*unit::inch`. Missing units are assumed to be mm.

With `Layout` = `Relative`, the distance of the bottom side of the scene to the bottom side of the canvas must be specified as a fraction of the canvas height, i.e., as a number between 0 and 1. The distance of the left hand side of the scene to the left hand side of the canvas must be specified as a fraction of the canvas width, i.e., as a number between 0 and 1.

The lower left corner of a scene may be placed outside the canvas. The parts of a scene outside the canvas are clipped.

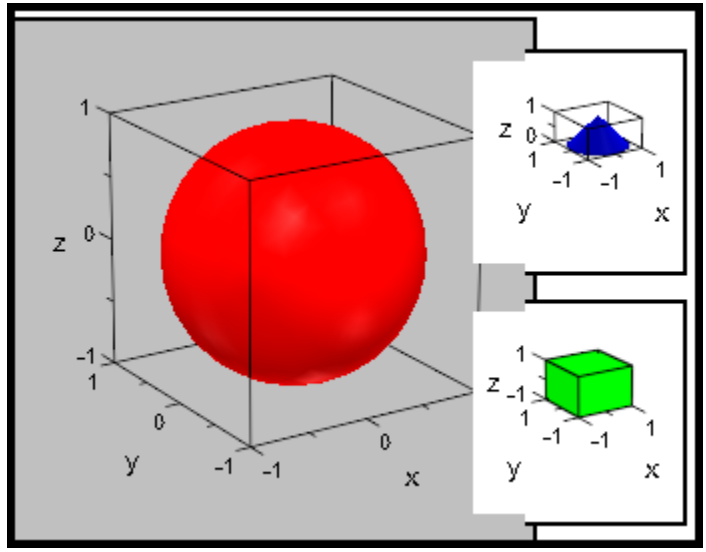
Overlapping scenes can be created. In such a situation it may be useful to create transparent scenes (without a background) via `BackgroundTransparent` = `TRUE`.

Examples

Example 1

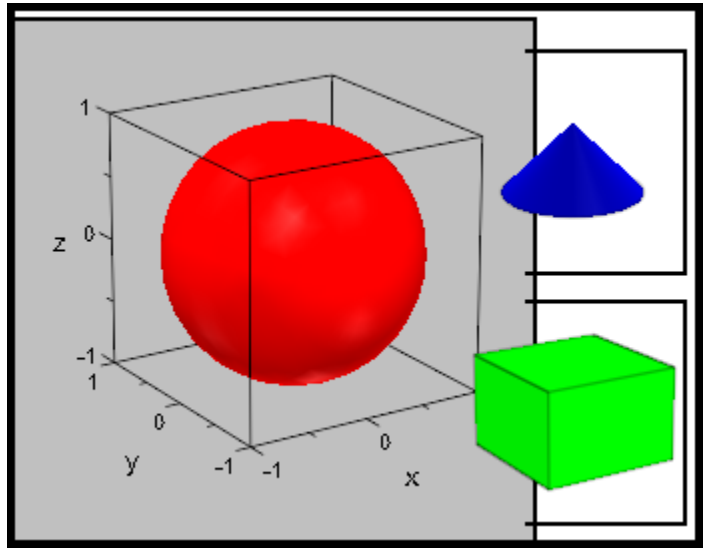
We demonstrate the layout of the canvas with `Layout = Absolute`. The scene `S1` is positioned automatically in the canvas using the default values `Bottom = 0`, `Left = 0`. The smaller scenes `S2` and `S3` are positioned explicitly via `Bottom` and `Left`:

```
S1 := plot::Scene3d(plot::Sphere(1, [0, 0, 0],
                                 Color = RGB::Red),
                    Width = 70*unit::mm, Height = 70*unit::mm,
                    BackgroundColor = RGB::Grey):
S2 := plot::Scene3d(plot::Box(-1..1, -1..1, -1..1,
                              Color = RGB::Green),
                    Width = 30*unit::mm, Height = 30*unit::mm,
                    Left = 60*unit::mm, Bottom = 3*unit::mm):
S3 := plot::Scene3d(plot::Cone(1, [0, 0, 0], [0, 0, 1],
                               Color = RGB::Blue),
                    Width = 30*unit::mm, Height = 30*unit::mm,
                    Left = 60*unit::mm, Bottom = 36*unit::mm):
plot(S1, S2, S3, Layout = Absolute,
     plot::Canvas::BorderWidth = 1.0*unit::mm,
     plot::Canvas::BorderColor = RGB::Black,
     plot::Canvas::Width = 92*unit::mm,
     plot::Canvas::Height = 72*unit::mm,
     plot::Scene3d::BorderWidth = 0.5*unit::mm,
     plot::Scene3d::BorderColor = RGB::Black):
```



We make the background of the small scenes transparent and switch the axes off:

```
S2::BackgroundTransparent := TRUE:
S2::Axes := None:
S3::BackgroundTransparent := TRUE:
S3::Axes := None:
plot(S1, S2, S3, Layout = Absolute,
     plot::Canvas::BorderWidth = 1.0*unit::mm,
     plot::Canvas::BorderColor = RGB::Black,
     plot::Canvas::Width = 92*unit::mm,
     plot::Canvas::Height = 72*unit::mm,
     plot::Scene3d::BorderWidth = 0.5*unit::mm,
     plot::Scene3d::BorderColor = RGB::Black):
```



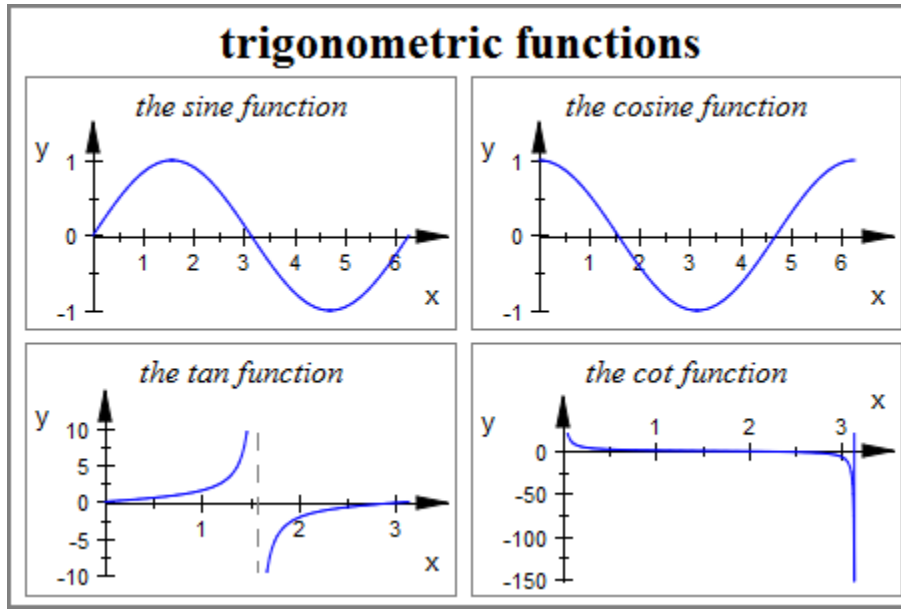

```
delete S1, S2, S3:
```

Example 2

We demonstrate the layout of the canvas with `Layout = Relative`. Apart from the scene headers and the positioning via `Bottom` and `Left`, all scene attributes are set in the `plot` call via specifications such as `plot::Scene2d::Width` etc. This distinguishes the scene attributes from the canvas attributes `Width`, `BorderWidth` etc.

```
S1 := plot::Scene2d(plot::Function2d(sin(x), x = 0..2*PI),
                    Left = 0.02, Bottom = 0.46,
                    Header = "the sine function"):
S2 := plot::Scene2d(plot::Function2d(cos(x), x = 0..2*PI),
                    Left = 0.51, Bottom = 0.46,
                    Header = "the cosine function"):
S3 := plot::Scene2d(plot::Function2d(tan(x), x = 0..PI),
                    Left = 0.02, Bottom = 0.02,
                    Header = "the tan function"):
S4 := plot::Scene2d(plot::Function2d(cot(x), x = 0..PI),
                    Left = 0.51, Bottom = 0.02,
                    Header = "the cot function"):
plot(S1, S2, S3, S4, Layout = Relative,
     Width = 120*unit::mm, Height = 80*unit::mm,
```

```
  BorderWidth = 0.5*unit::mm,
  HeaderFont = ["Times New Roman", 18, Bold],
  Header = "trigonometric functions",
  plot::Scene2d::Width = 0.475,
  plot::Scene2d::Height = 0.42,
  plot::Scene2d::BorderWidth = 0.2*unit::mm,
  plot::Scene2d::HeaderFont =
            ["Times New Roman", Italic, 12]):
```




```
delete S1, S2, S3, S4:
```

See Also

MuPAD Functions
Layout

Height, Width

Heights and widths of canvases and scenes

Value Summary

`Height`, `Width`	Inherited	Positive output size

Graphics Primitives

Objects	Default Values

`plot::Canvas`, `plot::Scene2d`, `plot::Scene3d`	`Height`: `80`
	
	`Width`: `120`

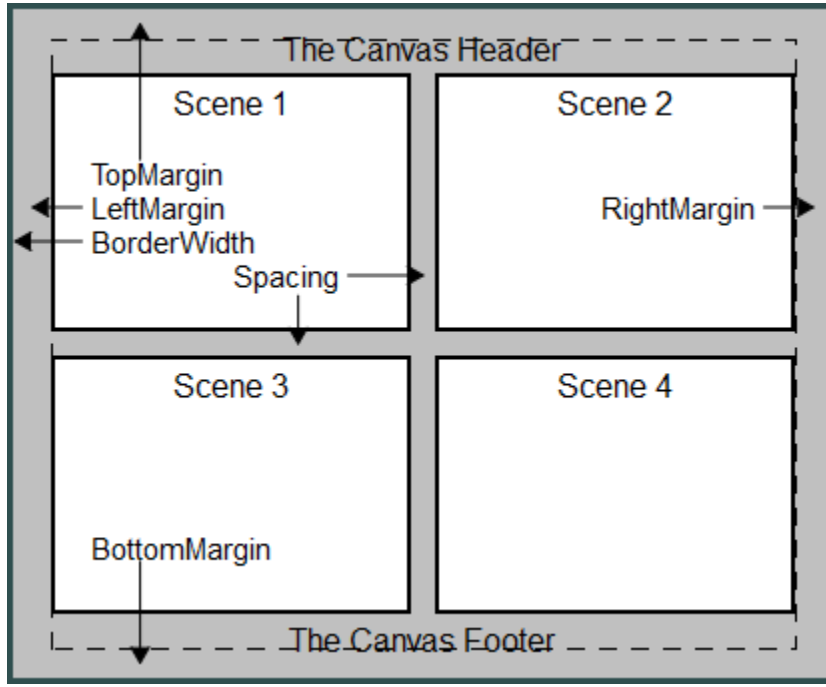
Description

`Height = h` and `Width = w` set the size of the canvas or scene to the height `h` and the width `w`.

For the canvas, the width and the height should be specified as physical lengths with a `unit`, e.g., `Width = 120*unit::mm`, `Height = 4.72*unit::inch`. Numbers without a physical unit give the size in mm.

These values specify the (approximate) physical size of the canvas, with which the graphics appears on the screen. A printout of the MuPAD graphics will have this physical size precisely.

The following picture illustrates the layout of the canvas:



The width and height of the canvas include the margin set by `Margin` and the border set by `BorderWidth`.

When only one scene is displayed in the canvas, this scene fills the canvas, i.e., the scene size coincides with the canvas size. When the canvas contains several scenes, there are various layout schemes, set by the canvas attribute `Layout`, to arrange the scenes in the canvas. Two schemes allow to set the size of the scenes indepently of the canvas size:

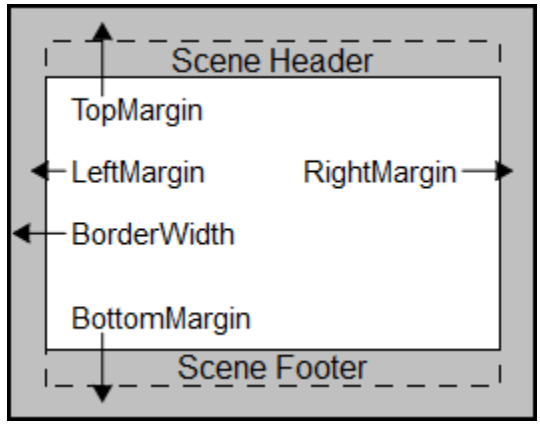
Note: For scenes, the attributes `Width` and `Height` are only used when plotting with the canvas attribute `Layout` set to `Absolute` or `Relative`.

With `Layout = Absolute`, width and height of a scene must be specified as physical lengths with a unit, e.g., `Height = 40*unit::mm`, `Width = 2.4*unit::inch` (missing units are assumed to be mm).

With `Layout = Relative`, width and height of a scene must be specified as fractions of the canvas widht and height, i.e., as numbers between 0 and 1.

The lower left corner of a scene can be moved to any position via the attributes `Bottom` and `Left`.

The following picture illustrates the layout of a scene:



The width and height of the scene include the margin set by `Margin` and the border set by `BorderWidth`.

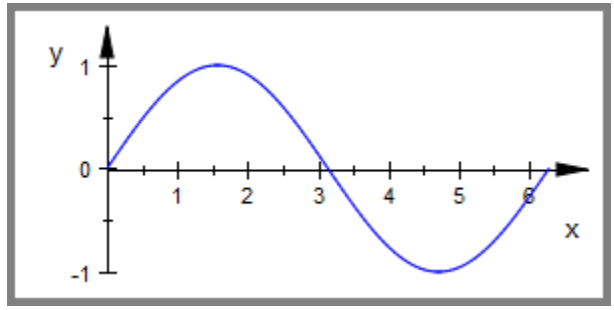
If a scene exceeds the canvas, the corresponding parts of the scene are clipped.

Examples

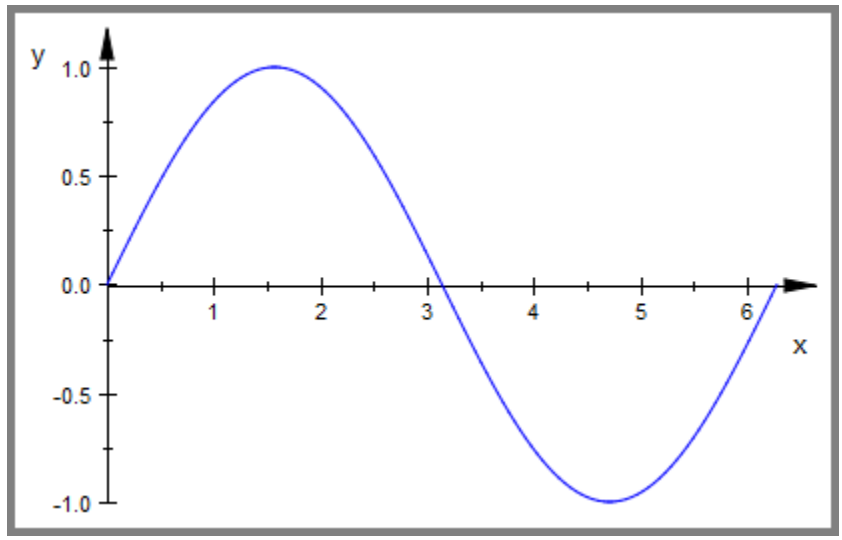
Example 1

The following calls produce plots of the physical sizes 8 ×4 cm and 11 ×7 cm, respectively.

```
f := plot::Function2d(sin(x), x = 0..2*PI):
plot(f, Width = 80*unit::mm, Height = 4*unit::cm,
     BorderWidth = 1.0*unit::mm):
```




```
plot(f, Width = 110*unit::mm, Height = 7*unit::cm,
     BorderWidth = 1.0*unit::mm):
```



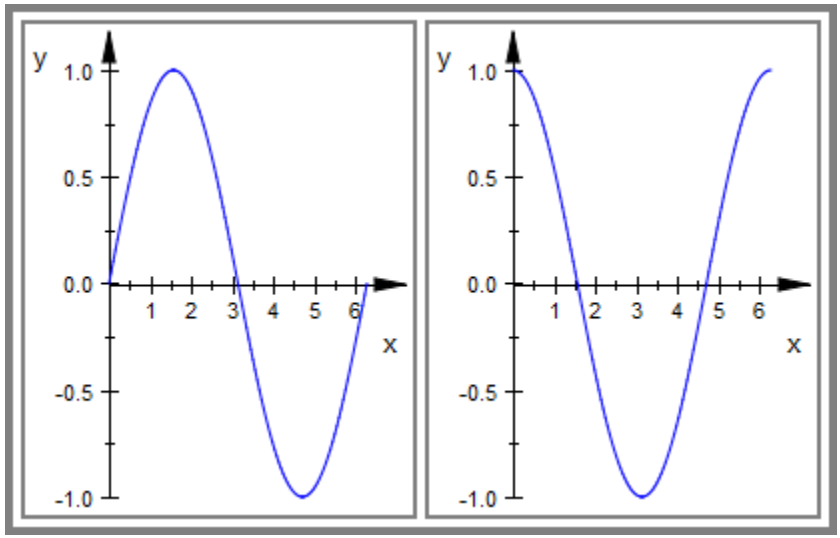

```
delete f:
```

Example 2

In the following graphics, we place two scenes in one canvas:

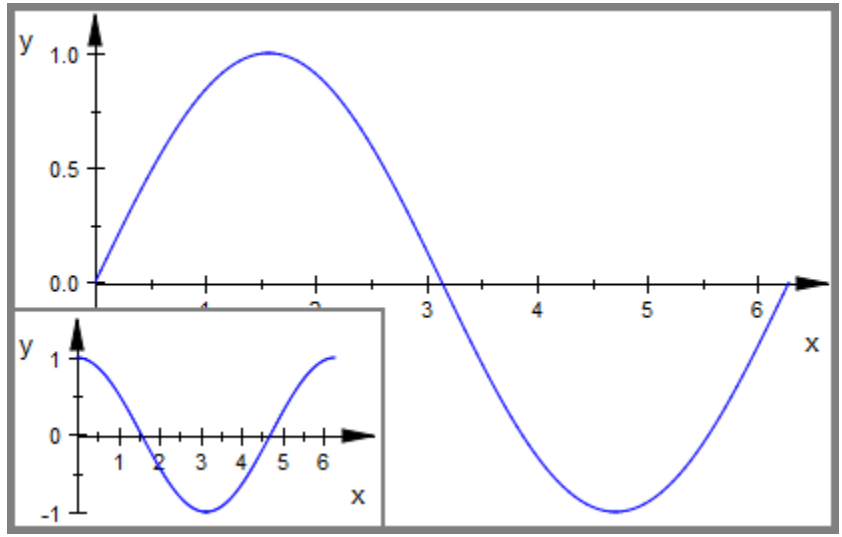
```
f1 := plot::Function2d(sin(x), x = 0..2*PI):
```

```
f2 := plot::Function2d(cos(x), x = 0..2*PI):
S1 := plot::Scene2d(f1, BorderWidth = 0.5*unit::mm,
                    Height = 7*unit::cm, Width = 11*unit::cm):
S2 := plot::Scene2d(f2, BorderWidth = 0.5*unit::mm,
                    Height = 3*unit::cm, Width = 5*unit::cm):
plot(S1, S2, Layout = Horizontal, BorderWidth = 1.0*unit::mm,
     Height = 7*unit::cm, Width = 11*unit::cm):
```



Note that with `Layout = Horizontal`, the size attributes of the scenes were ignored in the plot above. They affect the graphic when switching `Layout` to `Absolute`, either interactively in the inspector or directly in the `plot` call:

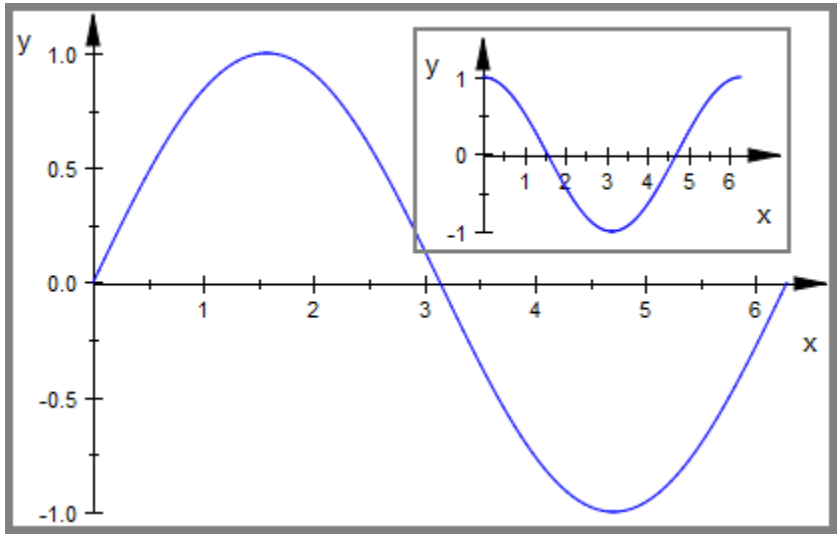
```
plot(S1, S2, Layout = Absolute, BorderWidth = 1.0*unit::mm,
     Height = 7*unit::cm, Width = 11*unit::cm):
```



Note that we did not set the attributes `Bottom` and `Left` of the scenes, so the bottom left corners of the scenes are placed in the bottom left corner of the canvas.

We make the background of the scene `S2` transparent via `BackgroundTransparent = TRUE` and shift this scene via suitable values of `Bottom` and `Left`:

```
S2::BackgroundTransparent := TRUE:
S2::Bottom := 3.7*unit::cm:
S2::Left := 5.4*unit::cm:
plot(S1, S2, Layout = Absolute, BorderWidth = 1.0*unit::mm,
     Height = 7*unit::cm, Width = 11*unit::cm):
```




```
delete f1, f2, S1, S2:
```

See Also

MuPAD Functions

BorderWidth | Bottom | Layout | Left | Margin

Layout, Rows, Columns

Arrangement/layout of several scenes in a canvas

Value Summary

`Layout`	Optional	`Absolute`, `Horizontal`, `Relative`, `Tabular`, or `Vertical`
`Columns`, `Rows`	Optional	Positive integer

Graphics Primitives

Objects	Default Values

`plot::Canvas`	`Layout`: `Tabular`
	
	`Rows`, `Columns`: `0`

Description

`Layout` determines the arrangement of several scenes in a canvas.

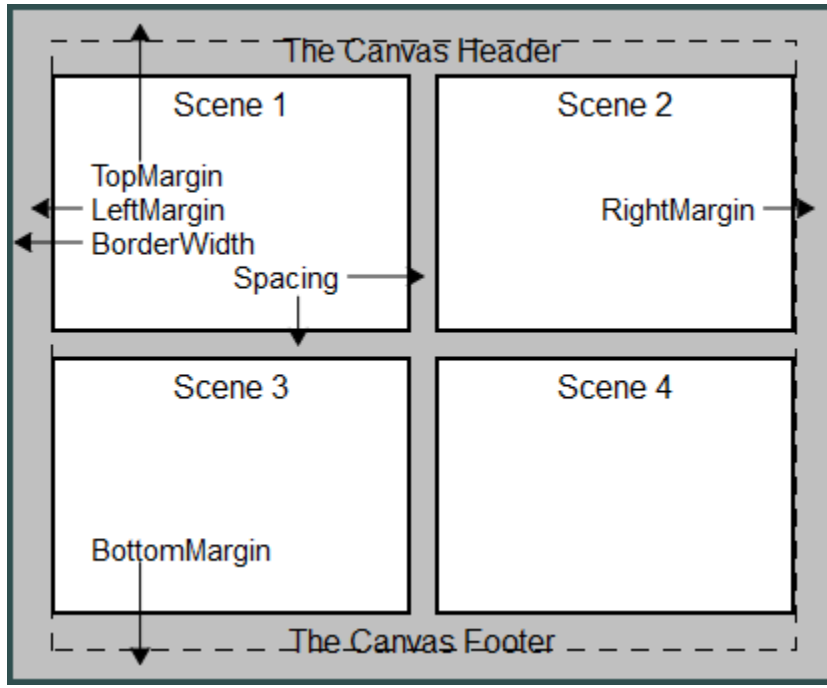
`Rows` determines the number of scene rows in a tabular arrangement of several scenes.

`Columns` determines the number of scene columns in a tabular arrangement of several scenes.

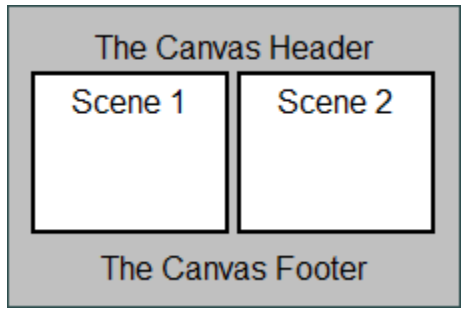
If a canvas contains more than one scene, the `Layout` attribute determines how the scenes are arranged in the canvas:

- With the default setting `Layout` = `Tabular`, a sequence of scenes in a canvas is arranged like a table with several columns and rows. The number of columns or rows may be chosen via the attributes `Columns` or `Rows`, respectively. If none of these attributes is given, the tabular layout scheme chooses some suitable values, automatically.

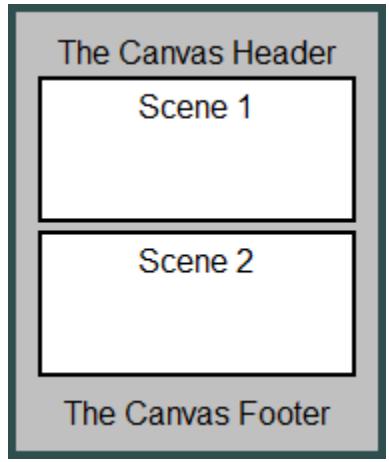
The scenes are filled into the table according to standard western reading order, filling the upper row from left to right, then proceeding to the next row etc:



- `Layout` = `Horizontal` is a shortcut for `Layout` = `Tabular`, `Rows` = `1`. The scenes are placed side by side in a single row.



- `Layout = Vertical` is a shortcut for `Layout = Tabular, Columns = 1`. The scenes a placed below each other in a single column.



The settings `Layout = Absolute` and `Layout = Relative` switch the automatic layout mode off and allow to position each scene via the scene attributes `Left` and `Bottom`. These attributes determine the position of the lower left corner of the scene and can be set separately for each scene.

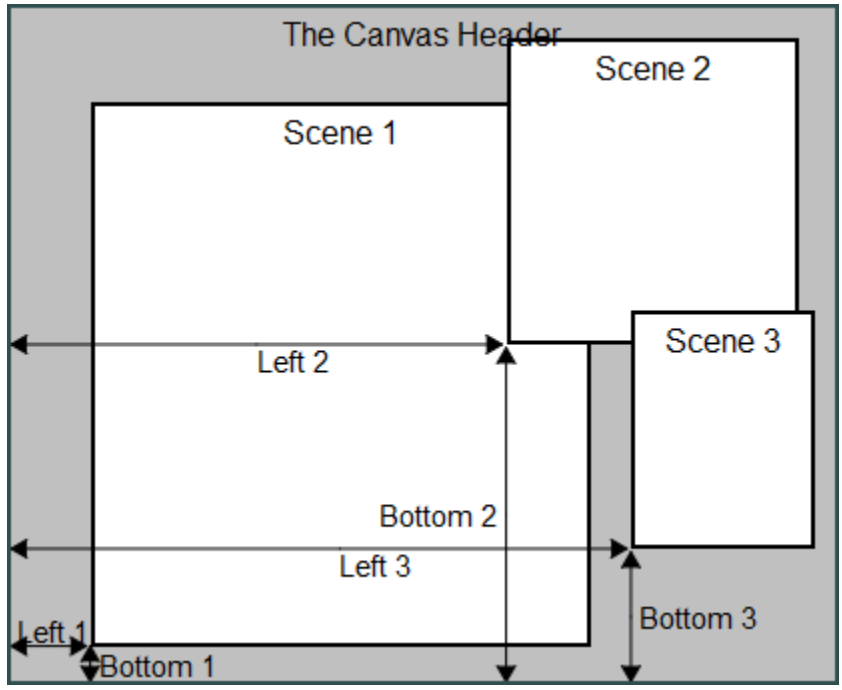
- With `Layout = Absolute`, the values for the lower left corner of the scene as well as its width and height must be specified as absolute physical lengths such as `Left = 3.0*unit::mm`, `Bottom = 4.5*unit::mm`, `Width = 10*unit::cm`, `Height = 4*unit::inch`.
- With `Layout = Relative`, these values must be specified as fractions of the canvas height and width. E.g.,

  ```
  Layout = Relative,
  Left = 0.3, Bottom = 0.2, Width = 0.5, Height = 0.5
  ```

  is equivalent to

  ```
  Layout = Absolut,
  Left = 0.3*canvaswidth, Bottom = 0.2*canvasheight,
  Width = 0.5*canvaswidth, Height = 0.5*canvasheight,
  ```

where `canvaswidth` and `canvasheight` are the physical width and height of the canvas.



With `Layout` = `Absolute` and `Layout` = `Relative` overlapping scenes can be created. In such a situation it may be useful to create transparent scenes (without a background) via `BackgroundTransparent` = `TRUE`.

Examples

Example 1

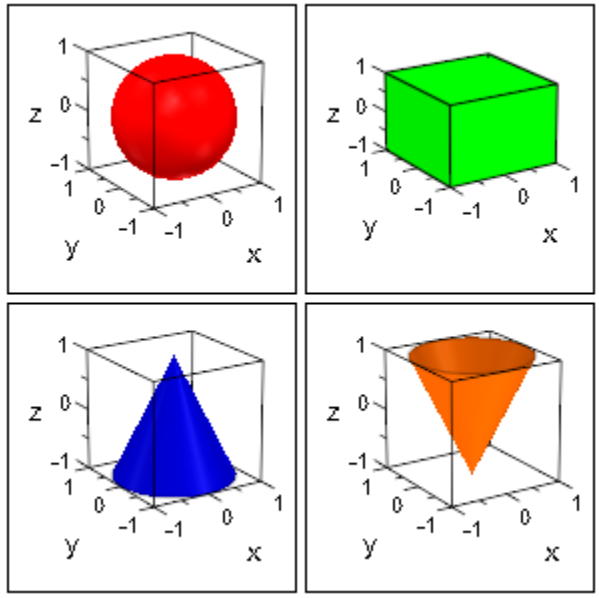
We define four scenes:

```
S1 := plot::Scene3d(plot::Sphere(1, [0, 0, 0],
                                 Color = RGB::Red),
```

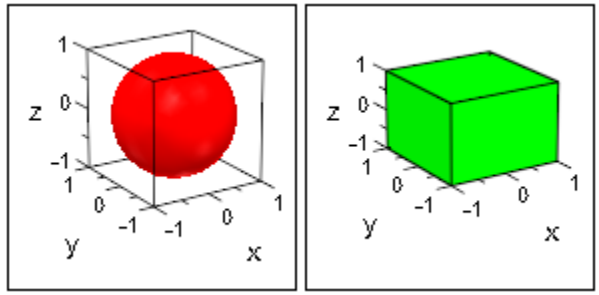
```
                    BorderWidth = 0.2*unit::mm,
                    BorderColor = RGB::Black):
S2 := plot::Scene3d(plot::Box(-1..1, -1..1, -1..1,
                                Color = RGB::Green),
                    BorderWidth = 0.2*unit::mm,
                    BorderColor = RGB::Black):
S3 := plot::Scene3d(plot::Cone(1, [0, 0, -1], [0, 0, 1],
                                 Color = RGB::Blue),
                    BorderWidth = 0.2*unit::mm,
                    BorderColor = RGB::Black):
S4 := plot::Scene3d(plot::Cone(1, [0, 0, 1], [0, 0, -1],
                                 Color = RGB::Orange),
                    BorderWidth = 0.2*unit::mm,
                    BorderColor = RGB::Black):
```

These scenes are positioned in the canvas in various ways:

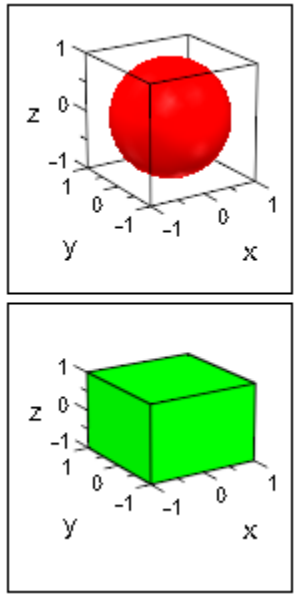
```
plot(S1, S2, S3, S4, Layout = Tabular,
     Height = 80*unit::mm, Width = 80*unit::mm):
```




```
plot(S1, S2, Layout = Horizontal,
     Height = 40*unit::mm, Width = 80*unit::mm):
```




```
plot(S1, S2, Layout = Vertical,
     Height = 80*unit::mm, Width = 40*unit::mm):
```



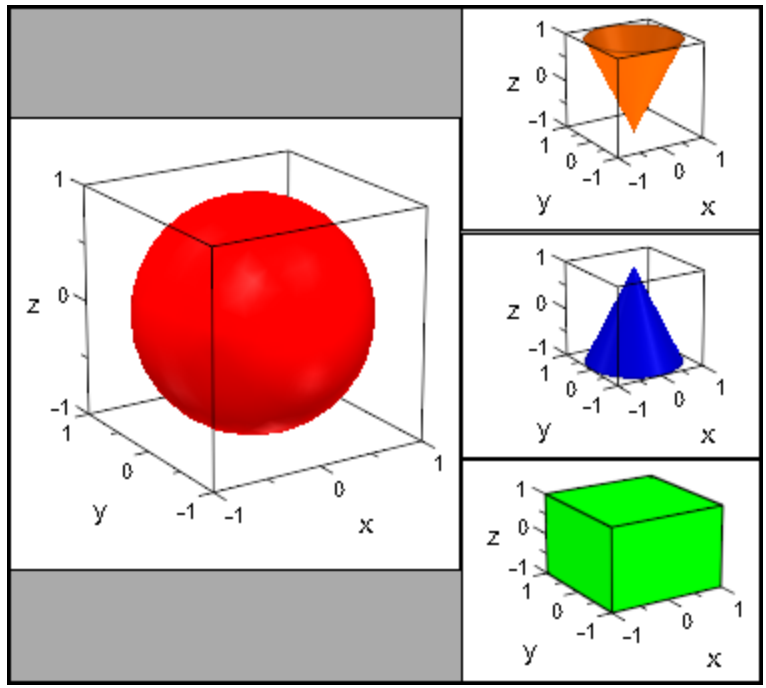
For explicit placement of the scenes, we set values for the `Left`, `Bottom`, `Width`, and `Height` attributes of the scenes:

```
S1::Left := 0:            S1::Bottom := 15*unit::mm:
S1::Width := 60*unit::mm: S1::Height:= 60*unit::mm:
S2::Left := 60*unit::mm:  S2::Bottom :=  0*unit::mm:
S2::Width := 40*unit::mm: S2::Height:= 30*unit::mm:
S3::Left := 60*unit::mm:  S3::Bottom := 30*unit::mm:
```

```
S3::Width := 40*unit::mm: S3::Height:= 30*unit::mm:
S4::Left := 60*unit::mm:  S4::Bottom := 60*unit::mm:
S4::Width := 40*unit::mm: S4::Height:= 30*unit::mm:
```

We use `Layout = Absolute`:

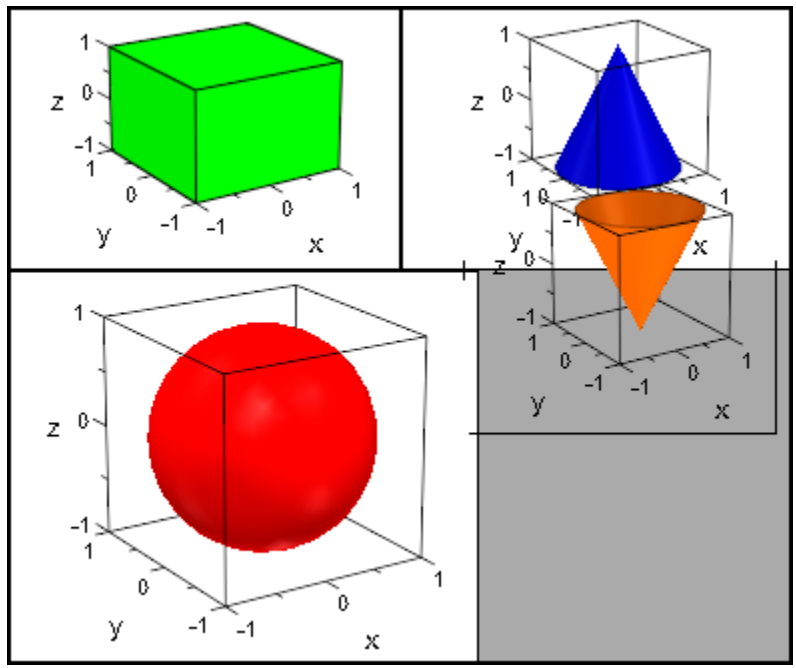
```
plot(S1, S2, S3, S4, Layout = Absolute,
     BorderWidth = 0.5*unit::mm, BorderColor = RGB::Black,
     BackgroundColor = RGB::LightGrey,
     Height = 90*unit::mm, Width = 100*unit::mm):
```



For `Layout = Relative`, the scene attributes `Left`, `Width`, `Bottom`, `Height` must be given as fractions of the canvas width and height, respectively:

```
S1::Left := 0:      S1::Width := 0.6:
S1::Bottom := 0:    S1::Height := 0.6:
S2::Left := 0:      S2::Width := 0.5:
S2::Bottom := 0.6:  S2::Height := 0.4:
S3::Left := 0.5:    S3::Width := 0.5:
S3::Bottom := 0.6:  S3::Height := 0.4:
```

```
S4::Left := 0.58:   S4::Width := 0.4:
S4::Bottom := 0.35: S4::Height := 0.4:
S4::BackgroundTransparent := TRUE:

plot(S1, S2, S3, S4, Layout = Relative,
     BorderWidth = 0.5*unit::mm, BorderColor = RGB::Black,
     BackgroundColor = RGB::LightGrey,
     Height = 87*unit::mm, Width = 104*unit::mm):
```



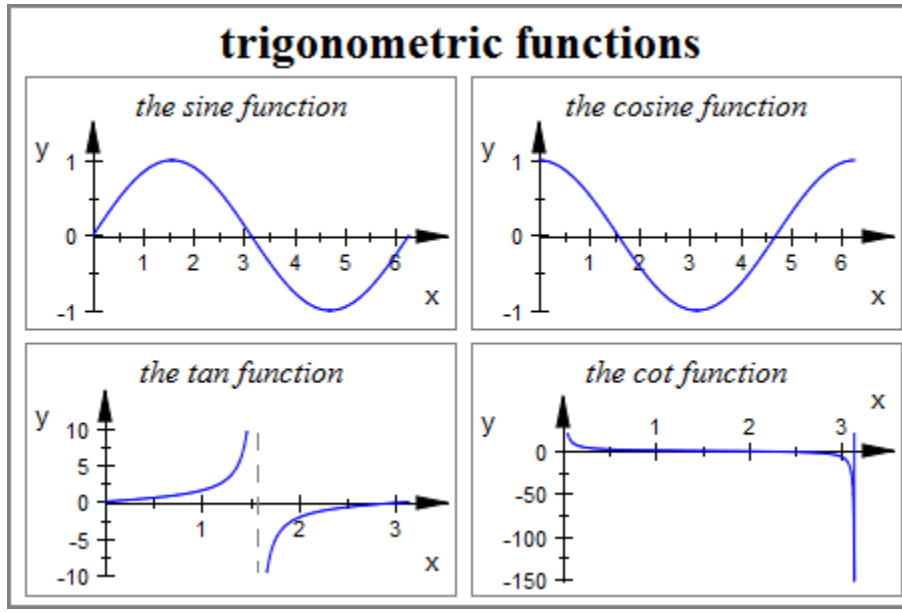

```
delete S1, S2, S3, S4:
```

Example 2

We demonstrate the layout of the canvas with `Layout` = `Relative`. Apart from the scene headers and the positioning via `Bottom` and `Left`, all scene attributes are set in the `plot` call via specifications such as `plot::Scene2d::Width` etc. This distinguishes the scene attributes from the canvas attributes `Width`, `BorderWidth` etc.

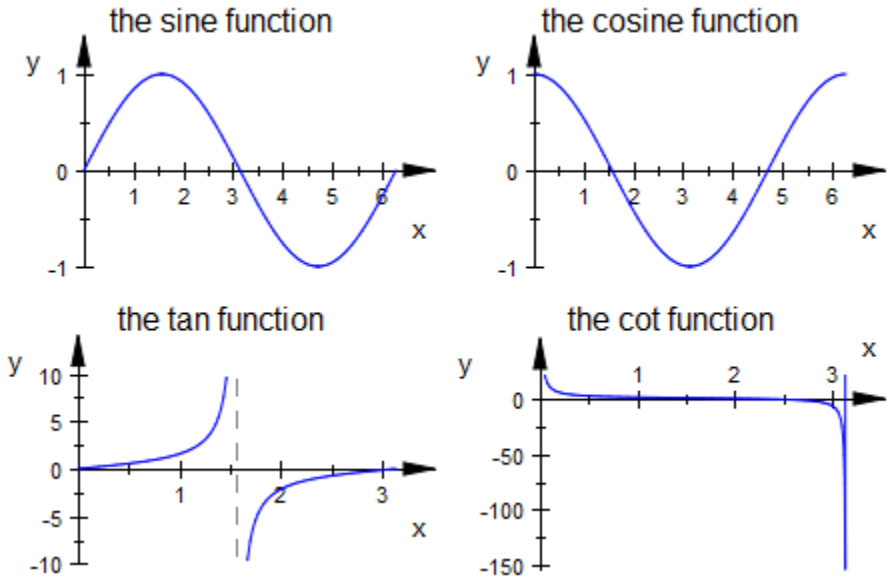
```
S1 := plot::Scene2d(plot::Function2d(sin(x), x = 0..2*PI),
                    Left = 0.02, Bottom = 0.46,
                    Header = "the sine function"):
```

```
S2 := plot::Scene2d(plot::Function2d(cos(x), x = 0..2*PI),
                    Left = 0.51, Bottom = 0.46,
                    Header = "the cosine function"):
S3 := plot::Scene2d(plot::Function2d(tan(x), x = 0..PI),
                    Left = 0.02, Bottom = 0.02,
                    Header = "the tan function"):
S4 := plot::Scene2d(plot::Function2d(cot(x), x = 0..PI),
                    Left = 0.51, Bottom = 0.02,
                    Header = "the cot function"):
plot(S1, S2, S3, S4, Layout = Relative,
     Width = 120*unit::mm, Height = 80*unit::mm,
     BorderWidth = 0.5*unit::mm,
     HeaderFont = ["Times New Roman", 18, Bold],
     Header = "trigonometric functions",
     plot::Scene2d::Width = 0.475,
     plot::Scene2d::Height = 0.42,
     plot::Scene2d::BorderWidth = 0.2*unit::mm,
     plot::Scene2d::HeaderFont =
               ["Times New Roman", Italic, 12]):
```



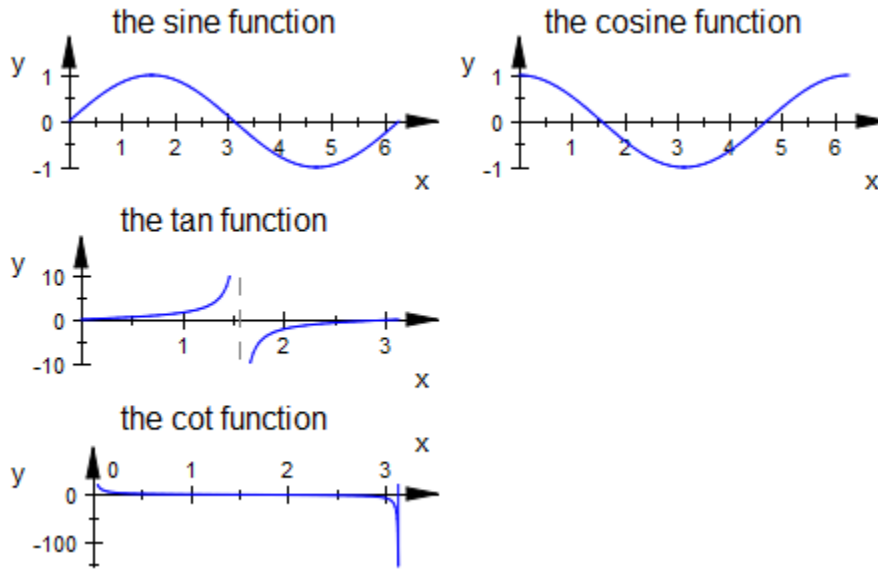
Finally, we demonstrate the attributes `Rows` and `Columns`. The automatic tabular layout ignores the explicit positioning of the scenes and chooses the following arrangement:

```
plot(S1, S2, S3, S4)
```



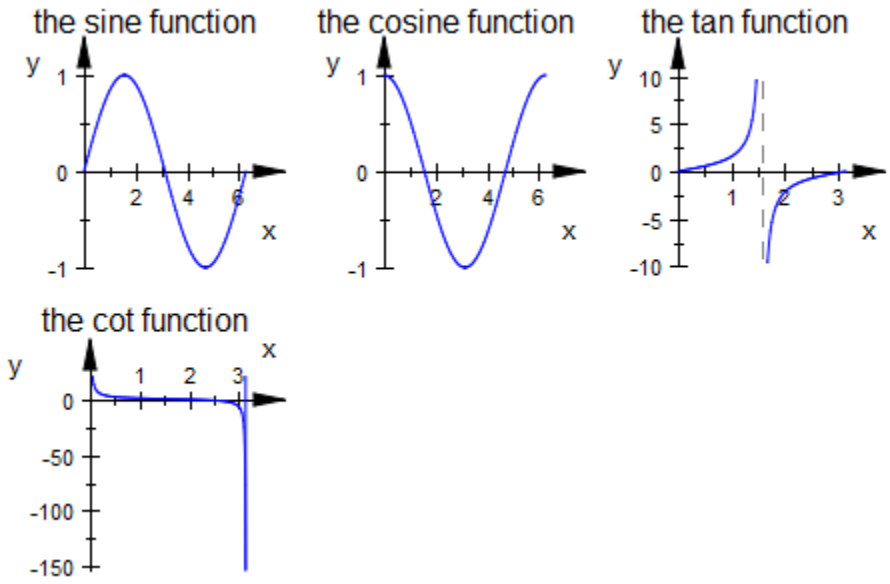
We explicitly request three rows:

```
plot(S1, S2, S3, S4, Rows = 3)
```



We explicitly request three columns:

```
plot(S1, S2, S3, S4, Columns = 3)
```

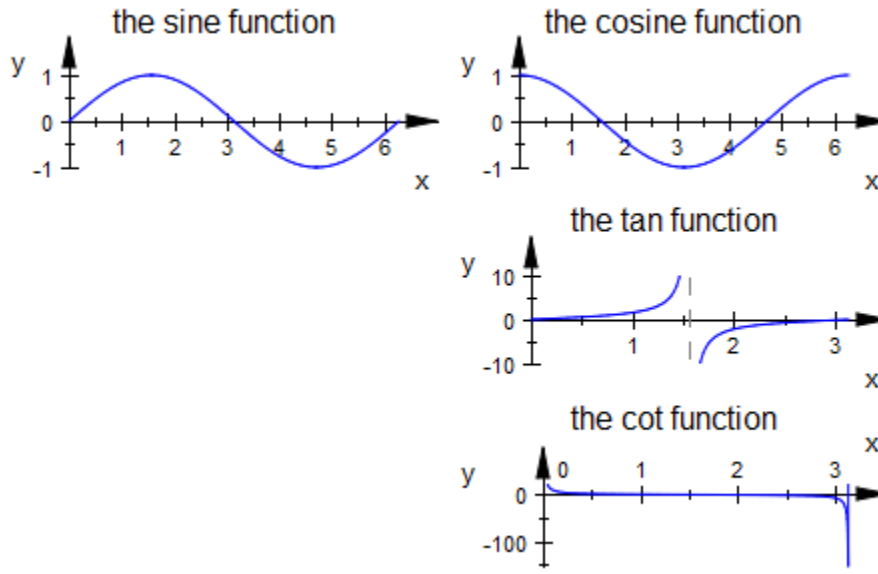


We generate an empty scene:

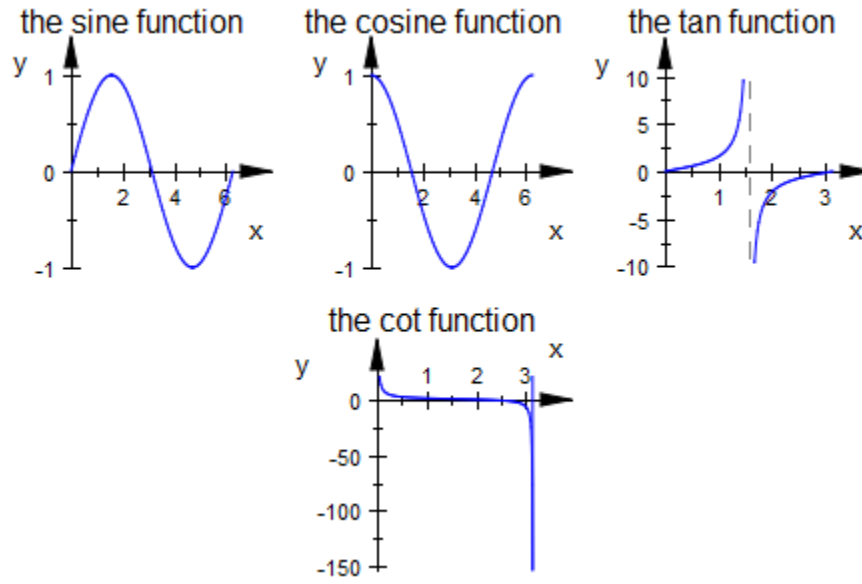
```
empty := plot::Scene2d(Axes = None):
```

The tabular layout is filled in with empty scenes:

```
plot(S1, S2, empty, S3, empty, S4, Rows = 3)
```




```
plot(S1, S2, S3, empty, S4, empty, Columns = 3)
```




```
delete S1, S2, S3, S4, empty:
```

See Also

MuPAD Functions

`BorderColor` | `BorderWidth` | `Bottom` | `BottomMargin` | `Left` | `LeftMargin` | `Margin` | `RightMargin` | `Spacing` | `TopMargin`

Margin, BottomMargin, TopMargin, LeftMargin, RightMargin

Margins around canvas and scenes

Value Summary

`Margin`	[{`BottomMargin`, `LeftMargin`, `RightMargin`, `TopMargin`}]	Non-negative output size
`BottomMargin`, `LeftMargin`, `RightMargin`, `TopMargin`	Inherited	Non-negative output size

Graphics Primitives

Objects	Default Values
`plot::Canvas`, `plot::Scene2d`, `plot::Scene3d`	`Margin`, `BottomMargin`, `TopMargin`, `LeftMargin`, `RightMargin`: `1`

Description

`Margin = d` sets a margin of size `d` around a canvas or scene. The margins at the bottom, to the left etc. can also be specified separately via `BottomMargin = d`$_1$, `LeftMargin = d`$_2$ etc.

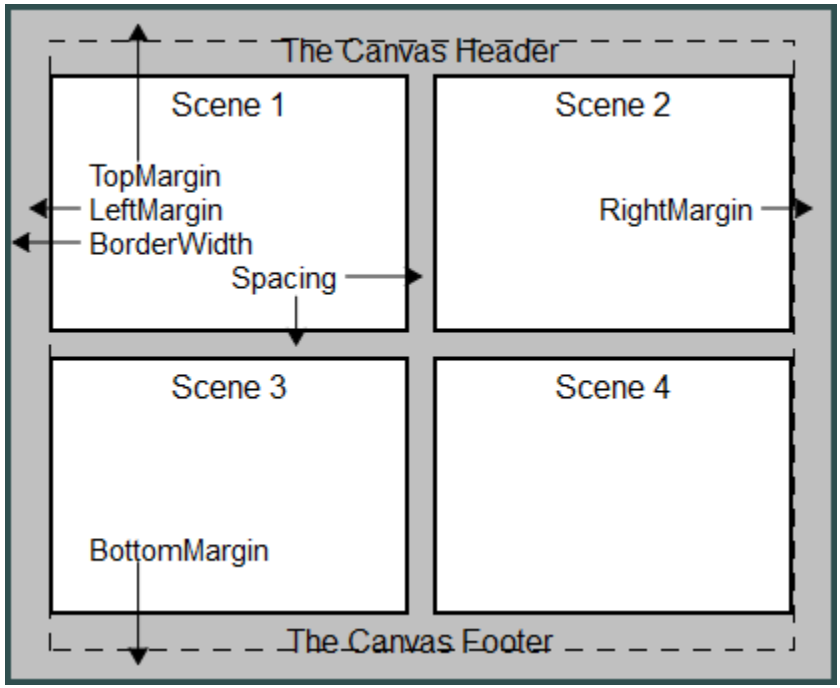
The canvas as well as the scenes have a margin that is not used for displaying graphical objects or captions. Its color coincides with the background color of the canvas or the scenes, respectively.

The size `d` of this margin is set by specifying `Margin = d` in a canvas or in a scene (of type `plot::Scene2d` or `plot::Scene3d`), respectively. Here, `d` is the physical width of the margin, e.g., `Margin = 0.5*unit::mm`.

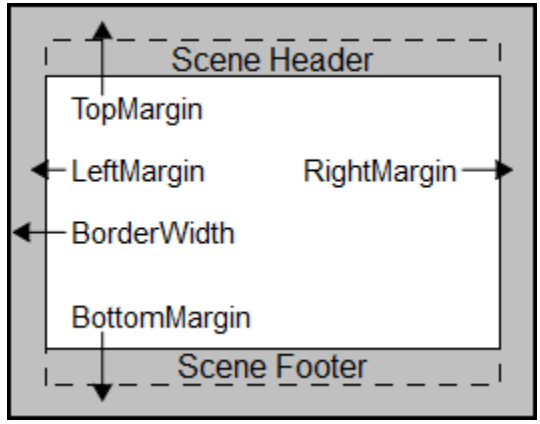
The margin sizes at the bottom, left, right, top of the canvas or the scenes can be specified separately via `BottomMargin = d1`, `LeftMargin = d2`, `RightMargin = d3`, `TopMargin = d4`.

The attribute `Margin = d` is a shortcut for `BottomMargin = d`, `LeftMargin = d`, `RightMargin = d`, `TopMargin = d`.

The following picture illustrates the layout of the canvas:



The following picture illustrates the layout of a scene:



The size of a canvas, set by the attributes `Width` and `Height`, includes the width of the margin set by `Margin`. The same holds for the scenes.

With `BackgroundTransparent = TRUE`, transparent scenes (without a background) can be created. The margin becomes transparent as well.

The margins do not react to `Layout = Relative`. One always has to specify the margin width as absolut physical lengths such as `0.5*unit::mm`.

Scenes do *not* inherit margin widths from the enclosing canvas. You can set margin widths for all scenes simultaneously by specifying them in `plot::setDefault` as `plot::Scene2d::Margin` or `plot::Scene3d::Margin`, respectively. Cf. “Example 2” on page 24-1757.

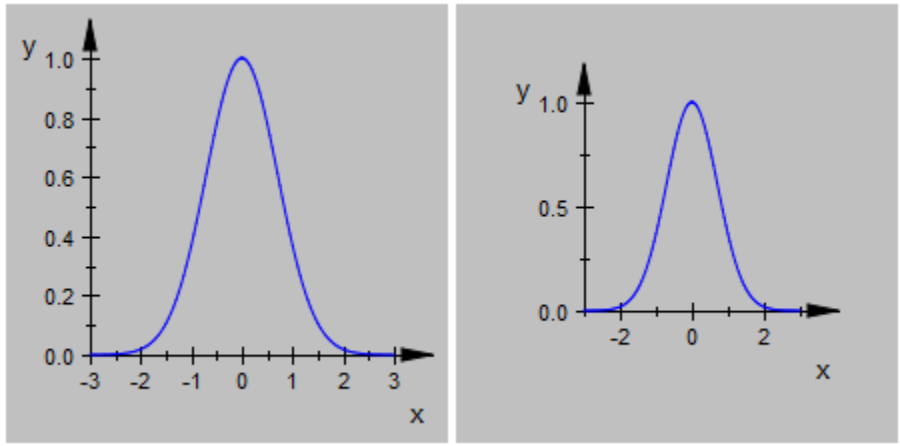
Examples

Example 1

The following two scenes display the same function graph using different margins:

```
f := plot::Function2d(exp(-x^2), x = -3..3):
plot(plot::Scene2d(f, Margin = 2*unit::mm,
                   BackgroundColor = RGB::Grey),
     plot::Scene2d(f, Margin = 8*unit::mm,
                   BackgroundColor = RGB::Grey),
```

```
    Layout = Horizontal, Axes = Frame,
    Width = 120*unit::mm, Height = 60*unit::mm):
```




```
delete f:
```

Example 2

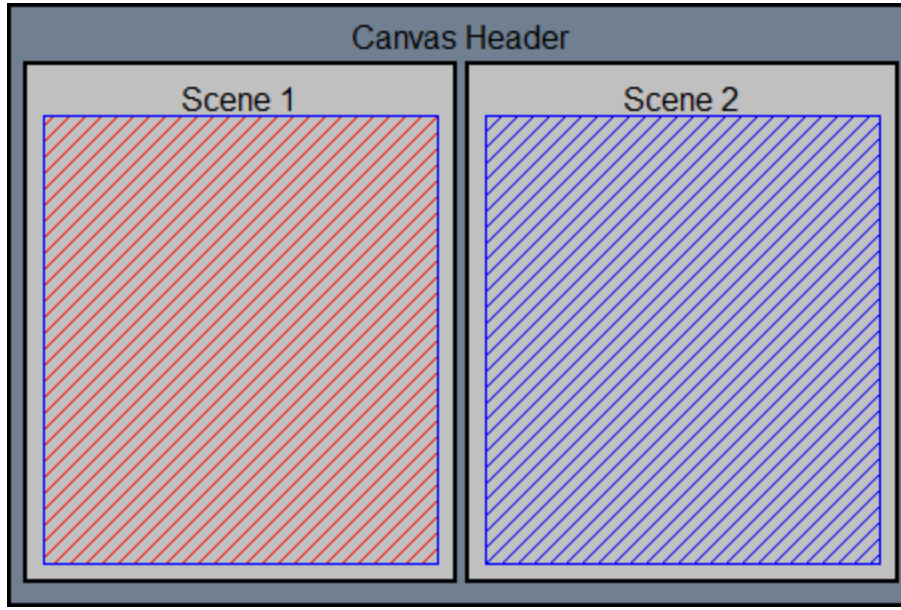
We use `plot::setDefault` to define new default values for the layout and style parameters `BorderWidth`, `BorderColor`, `Margin`, and `BackgroundColor`:

```
plot::setDefault(
  plot::Canvas::BorderWidth = 0.5*unit::mm,
  plot::Canvas::BorderColor = RGB::Black,
  plot::Canvas::Margin = 1.5*unit::mm,
  plot::Canvas::BackgroundColor = RGB::SlateGrey,
  plot::Scene2d::BorderWidth = 0.5*unit::mm,
  plot::Scene2d::BorderColor = RGB::Black,
  plot::Scene2d::Margin = 2*unit::mm,
  plot::Scene2d::BackgroundColor = RGB::Grey
):
```

The following canvas contains two scenes. This plot uses the new defaults:

```
plot(plot::Scene2d(plot::Rectangle(-1..1, -1..1,
                     Filled = TRUE, FillColor = RGB::Red,
                     Header = "Scene 1")),
     plot::Scene2d(plot::Rectangle(-1..1, -1..1,
```

```
                    Filled = TRUE, FillColor = RGB::Blue,
                    Header = "Scene 2")),
Layout = Horizontal, Axes = None,
Header = "Canvas Header"):
```



See Also

MuPAD Functions

BackgroundColor | BackgroundColor2 | BackgroundStyle | BackgroundTransparent | BorderColor | BorderWidth | Bottom | Left

OutputUnits

Physical length unit used by the inspector

Value Summary

Optional	`unit::cm`, `unit::dm`, `unit::inch`, `unit::km`, `unit::m`, `unit::mm`, or `unit::pt`

Graphics Primitives

Objects	OutputUnits Default Values
`plot::Canvas`	`unit::mm`

Description

Various length parameters in the MuPAD graphics such as the width and the height of the canvas, the length of tick marks, the width of lines, the size of points etc. may be specified as physical lengths with a length unit. The inspector allows to display a physical length in the physical unit set by `OutputUnits`.

For example, when specifying the canvas size by the attributes `Width = 120*unit::mm`, `Height = 80*unit::mm`, the MuPAD graphics will appear on the screen in a canvas of 120 ×80 mm (approximately). A printout of the MuPAD graphics will have this physical size precisely.

One may also specify these lengths as pure numbers such as `Width = 120`, `Height = 80`. In this case, the physical length unit is given in mm.

In the “object inspector” of the MuPAD Graphics Tool (see the section Viewer, Browser, and Inspector: Interactive Manipulation of this document), lengths are displayed as numbers without unit. The actual physical length is given by these numbers times the physical length unit given by `OutputUnits`.

Note: Note that the specification `Width = 10, OutputUnit = unit::inch` does not mean `Width = 10*unit::inch`, but `Width = 10*unit::mm`, displayed as `0.3937...` inches.

It is recommended to specify output sizes always as products of the numerical values times the unit.

Changing the value of `OutputUnits` does not change the physical lengths! When changing `OutputUnits = unit::mm` to `OutputUnits = unit::inch`, say, the numbers in the object inspector such as `Width = 120` (corresponding to a canvas size of 120 mm in the real world) change automatically to `Width = 4.7244...` (corresponding to the same canvas size 120 mm = 4.7244... inches).

If you want to change the physical length, you need to change the number in the input region of `Width` in the object inspector.

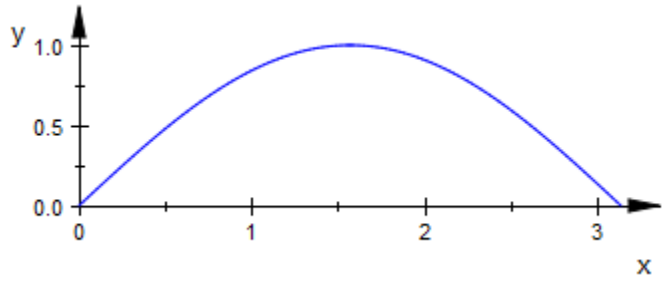
Switching between different output units via `OutputUnits` is convenient if physical conditions such as the real world size of a printout have to be met. Depending on your nationality, you will have a preference for inches or millimeters.

Examples

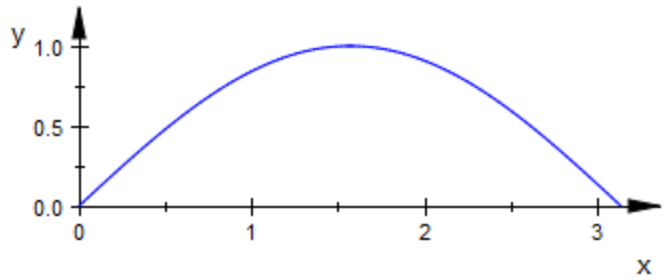
Example 1

The following calls all produce graphical output of the same physical size:

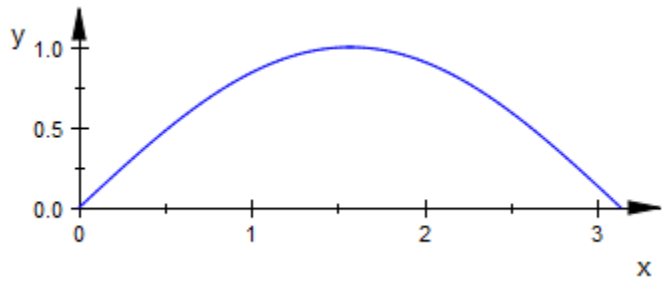
```
f := plot::Function2d(sin(x), x = 0..PI):
plot(f, Width = 90*unit::mm, Height = 40*unit::mm):
```



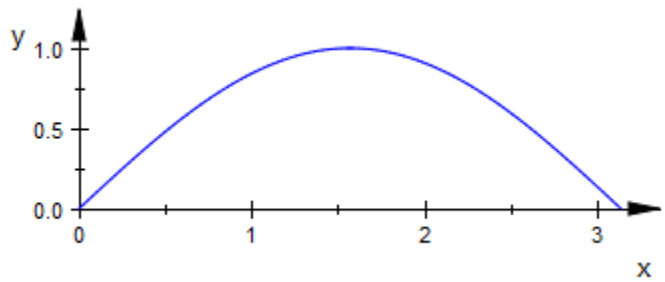
```
plot(f, Width = 90, Height = 40, OutputUnits = unit::mm):
```



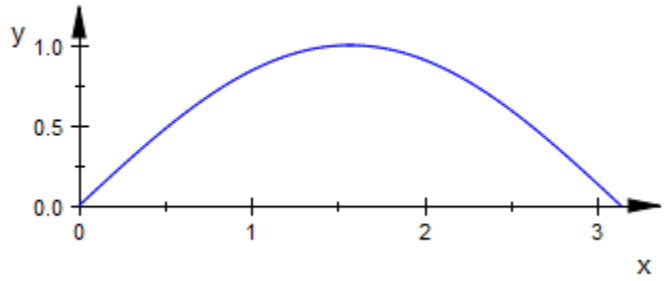

```
plot(f, Width = 90, Height = 40, OutputUnits = unit::inch):
```




```
plot(f, Width = 3.544*unit::inch, Height = 40*unit::mm):
```

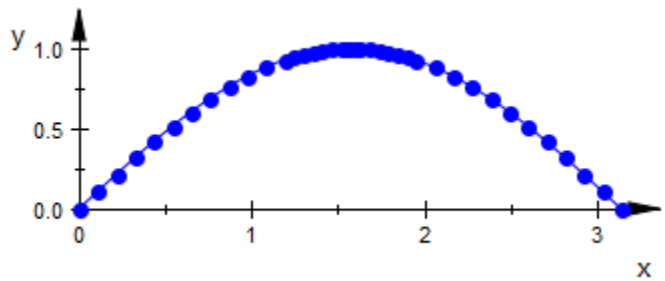



```
plot(f, Width = 3.544*unit::inch, Height = 1.575*unit::inch):
```



In the following plot command, the size of graphical points is specified in millimeters. The specification of `OutputUnits` = `unit::inch` does not change the physical point size of 2 mm = 0.07874... inch. It just means that the value of the point size is displayed as 0.07874 in the object inspector of the MuPAD Graphics Tool, not as 2:

```
plot(plot::Function2d(sin(x), x = 0..PI, Mesh = 30),
     PointsVisible = TRUE, PointSize = 2*unit::mm,
     Width = 90*unit::mm, Height = 40*unit::mm,
     OutputUnits = unit::inch):
```




```
delete f:
```

Example 2

The conversion between the output sizes can be computed via MuPAD:

```
120.0*unit::mm = unit::convert(120.0*unit::mm, unit::inch),
4.7244*unit::inch = unit::convert(4.7244*unit::inch, unit::pt)
```

$$120.0\ \text{mm} = 4.724409449\ \text{inch},\quad 4.7244\ \text{inch} = 341.4323914\ \text{pt}$$

See Also

MuPAD Functions

`AxesLineWidth` | `Bottom` | `BottomMargin` | `GridLineWidth` | `Height` | `Left` | `LeftMargin` | `LineWidth` | `PointSize` | `RightMargin` | `Spacing` | `SubgridLineWidth` | `TicksLength` | `TipLength` | `TopMargin` | `TubeDiameter` | `VerticalAsymptotesWidth` | `Width`

Spacing

Space between scenes

Value Summary

Optional	Non-negative output size

Graphics Primitives

Objects	Spacing Default Values
`plot::Canvas`	`1.0`

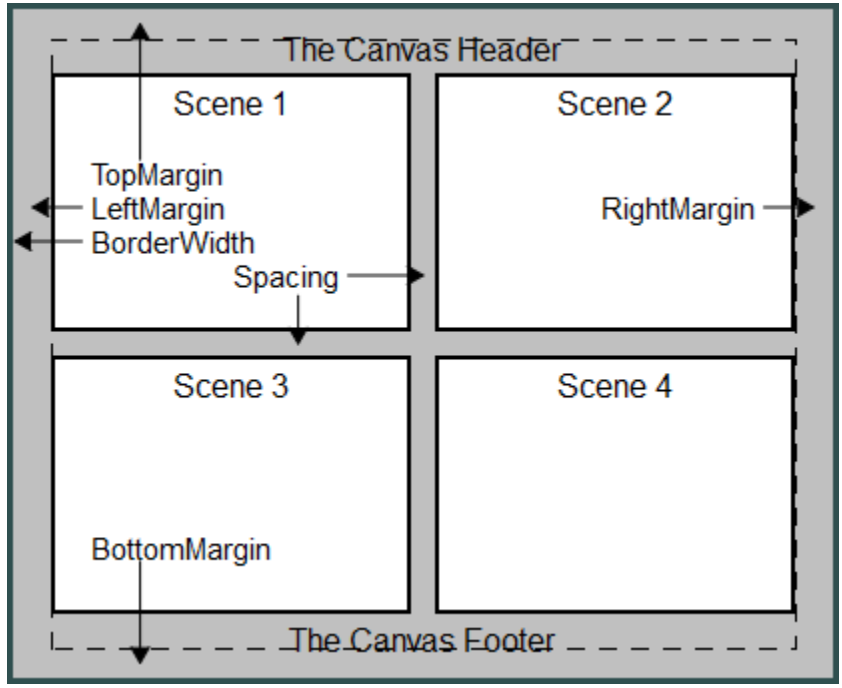
Description

`Spacing = d` sets a gap of size *d* between neighboring scenes in a canvas.

If a canvas contains several scenes, an automatic layout of the canvas may be requested by `Layout = Horizontal`, `Layout = Tabular`, or `Layout = Vertical`. In these modes, the scenes are separated by a gap that is set by `Spacing = d`. Here, `d` is the physical width of the gap, e.g., `Spacing = 0.5*unit::mm`.

The `Spacing` attribute has an effect only in conjunction with the automatic layout modes `Layout = Horizontal`, `Layout = Tabular`, or `Layout = Vertical`, respectively.

The following picture illustrates the layout of the canvas:



Examples

Example 1

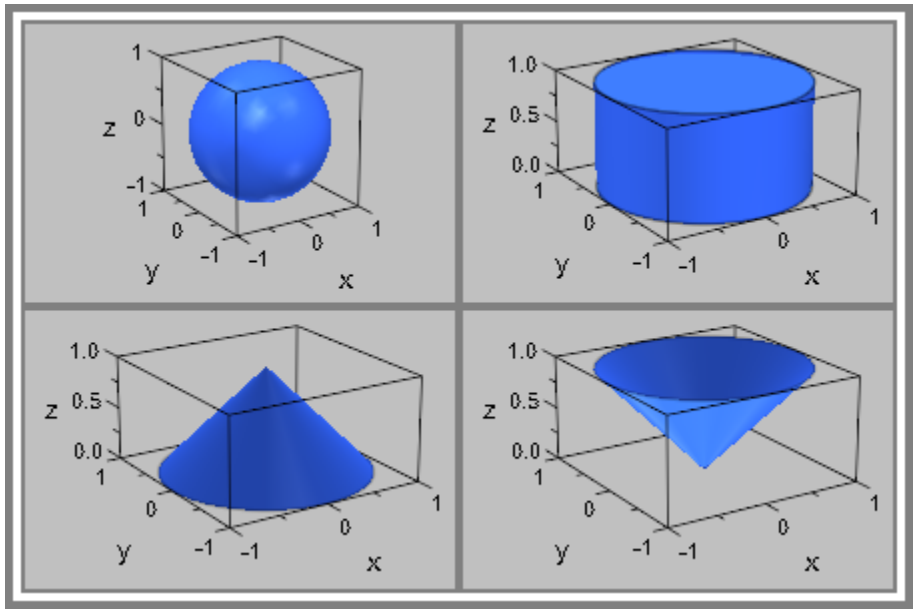
We define four scenes:

```
S1 := plot::Scene3d(plot::Sphere(1, [0, 0, 0]),
                    BackgroundColor = RGB::Grey,
                    BorderWidth = 0.5*unit::mm):
S2 := plot::Scene3d(plot::Cylinder(1, [0, 0, 0], [0, 0, 1]),
                    BackgroundColor = RGB::Grey,
                    BorderWidth = 0.5*unit::mm):
S3 := plot::Scene3d(plot::Cone(1, [0, 0, 0], [0, 0, 1]),
                    BackgroundColor = RGB::Grey,
                    BorderWidth = 0.5*unit::mm):
S4 := plot::Scene3d(plot::Cone(1, [0, 0, 1], [0, 0, 0]),
                    BackgroundColor = RGB::Grey,
```

```
                    BorderWidth = 0.5*unit::mm):
```

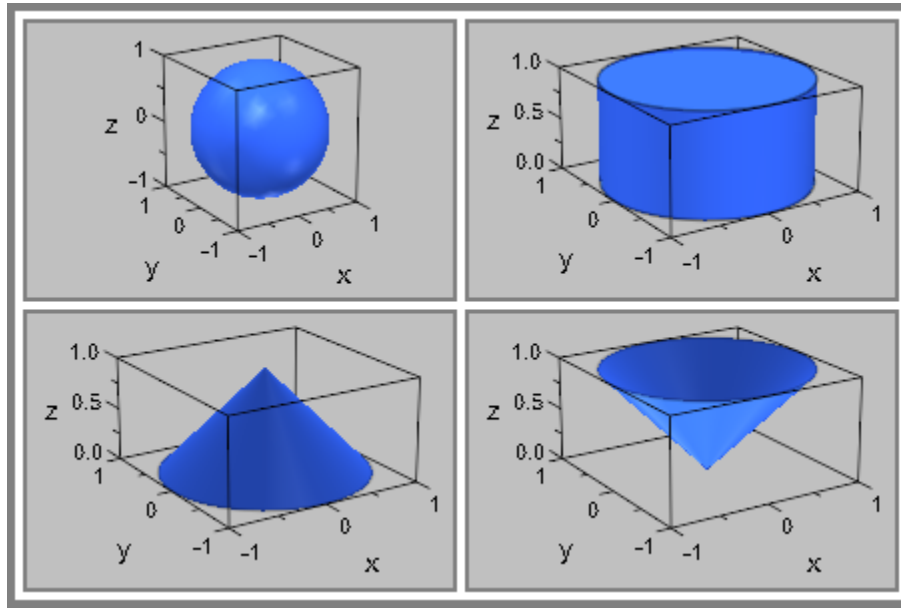
These scenes are positioned in the canvas with no gap between them (`Spacing = 0`). By default, the automatic layout mode `Layout = Tabular` is used:

```
plot(S1, S2, S3, S4, Spacing = 0,
     BorderWidth = 1.0*unit::mm):
```



We introduce a gap of 1 mm:

```
plot(S1, S2, S3, S4, Spacing = 1.0*unit::mm,
     BorderWidth = 1.0*unit::mm):
```




```
delete S1, S2, S3, S4:
```

See Also

MuPAD Functions

Bottom | BottomMargin | Layout | Left | LeftMargin | Margin | RightMargin | TopMargin

AbsoluteError, RelativeError

Maximal absolute discretization error

Value Summary

`AbsoluteError`, `RelativeError`	Optional	MuPAD expression

Graphics Primitives

Objects	Default Values
`plot::Ode2d`, `plot::Ode3d`	
`plot::Streamlines2d`	`RelativeError`: `1/100000`

Description

`AbsoluteError = atol` sets the tolerance `atol` for the maximal absolute discretization error in the numerical solution of ODEs.

`RelativeError = rtol` sets the tolerance `rtol` for the maximal relative discretization error.

Internally, `plot::Ode2d` and `plot::Ode3d` call the routine `numeric::odesolve` for solving the given ODE numerically. The attributes `AbsoluteError`, `RelativeError` are forwarded to `numeric::odesolve`. See the corresponding help page for further details.

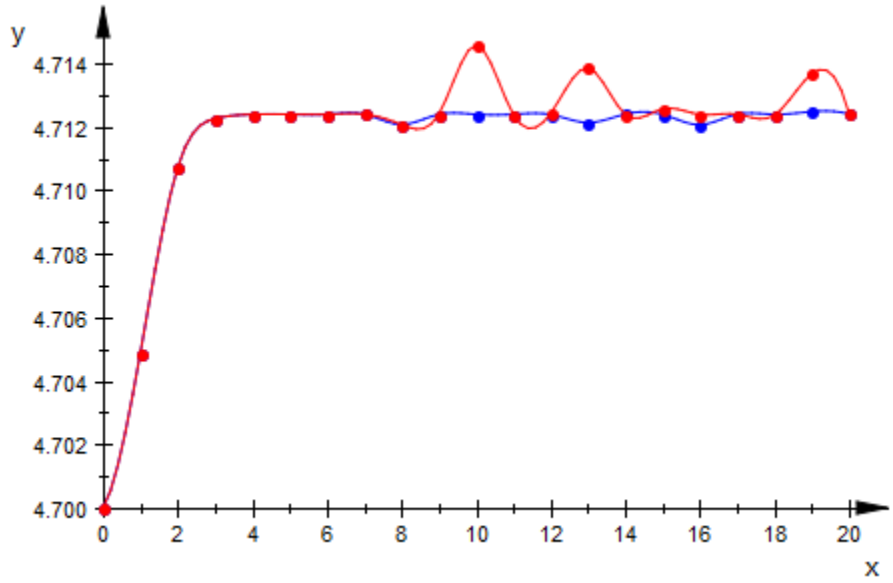
Examples

Example 1

We consider the initial value problem $y'(t) = -t\cos(y(t))$, $y(0) = 4.7$. The ODE is solved numerically with different tolerances for the relative discretization error. The 'oscillating'

behaviour of the red solution curve is a numerical artifact. The blue solution curve, computed with a smaller tolerance, is more precise:

```
f:= (t, Y) -> [-t*cos(Y[1])]:
Y0 := [4.7]:
plot(plot::Ode2d(f, [i $ i = 0..20], Y0, Color = RGB::Blue,
                 RelativeError = 0.0001),
     plot::Ode2d(f, [i $ i = 0..20], Y0, Color = RGB::Red,
                 RelativeError = 0.001))
```




```
delete f, Y0:
```

See Also

MuPAD Functions

`InitialConditions` | `ODEMethod` | `Projectors` | `Stepsize` | `TimeMesh`

AdaptiveMesh

Adaptive sampling

Value Summary

Inherited	Non-negative integer

Graphics Primitives

Objects	AdaptiveMesh Default Values
`plot::Function2d`	2
`plot::Conformal`, `plot::Curve2d`, `plot::Curve3d`, `plot::Cylindrical`, `plot::Function3d`, `plot::Implicit3d`, `plot::Polar`, `plot::Spherical`, `plot::Surface`, `plot::Sweep`, `plot::XRotate`, `plot::ZRotate`	0
`plot::Rootlocus`	4

Description

`AdaptiveMesh = n` controls the adaptive sampling in the numerical evaluation of functions, curves and surfaces. With $n = 0$, adaptive sampling is disabled. With $n > 0$, adaptive sampling is enabled.

The "depth" n of the adaptive sampling should be a *small* integer such as 0, 1, 2, or 3.

Continuous graphical objects such as function graphs, parameterized curves and surfaces are approximated by a discrete mesh of numerical points.

This mesh may be controlled by the user via the attributes `Mesh`, `Submesh`, and `AdaptiveMesh`. (Depending on the object, the `Mesh` attribute splits into more specific versions such us `UMesh` and `VMesh` for curve and surface plots, or `XMesh`, `YMesh`, `ZMesh` for function and implicit plots.)

First, the object is evaluated numerically on an equidistant "initial mesh" set via the attribute `Mesh` (or the more specific versions mentioned above).

With `AdaptiveMesh = 0`, the numerical data over the initial mesh are used to render the object without any further adaptive refinement.

With `AdaptiveMesh = n`, $n > 0$, further numerical data are computed before the renderer is called. In particular, the data of neighboring points on the initial mesh are investigated. If a point is not reasonably represented by a straight line connecting the neighboring points, the corresponding intervals of the initial mesh are sub-divided recursively. The adaptive mechanism descends into the sub-intervals of the initial mesh if consecutive line segments of the discretized plot object deviate from a straight line by a "bend angle" of more than 10 degrees. The intervals involved in such a situation are split into halves, recursively.

The value of n should be a *small* integer that determines the recursive depth of the adaptive refinement. In each direction, up to $2^n - 1$ additional points are placed between the points of the initial mesh.

If the object looks smooth on the initial mesh set via the attribute `Mesh` or its more detailed variants, the adaptive mechanism does *not* descend into the intervals of the initial mesh. If there are fine structures hidden inside these intervals, specifying `AdaptiveMesh = n` with $n > 0$ will *not* help to improve the plot. In such a case, the initial mesh should be refined via the appropriate attribute for the initial mesh.

On the other hand, if the initial mesh is fine enough to indicate finer internal structures via the "max bend angle" criterion, it is often more efficient to use `AdaptiveMesh = n` than to refine the initial mesh, because the adaptive mechanism refines only those parts of the object that do need refinement. This effect can be seen in "Example 3" on page 24-1777.

Note: Note that increasing the recursive depth n by 1 may increase the run time by a factor of 2 for line objects (2D function graphs and curves) and by a factor of 4 for surface objects (3D function graphs and surfaces). In most cases, a *small* value such as $n \in \{1, 2, 3\}$ suffices to obtain a reasonably smooth plot object.

Note: Note that the adaptive algorithm for surface objects in 3D is *very expensive!* As an alternative to values $n > 0$ in `AdaptiveMesh = n`, you may experiment with `AdaptiveMesh = 0`, `Submesh = [2^n- 1, 2^n- 1]` in 3D function graphs or

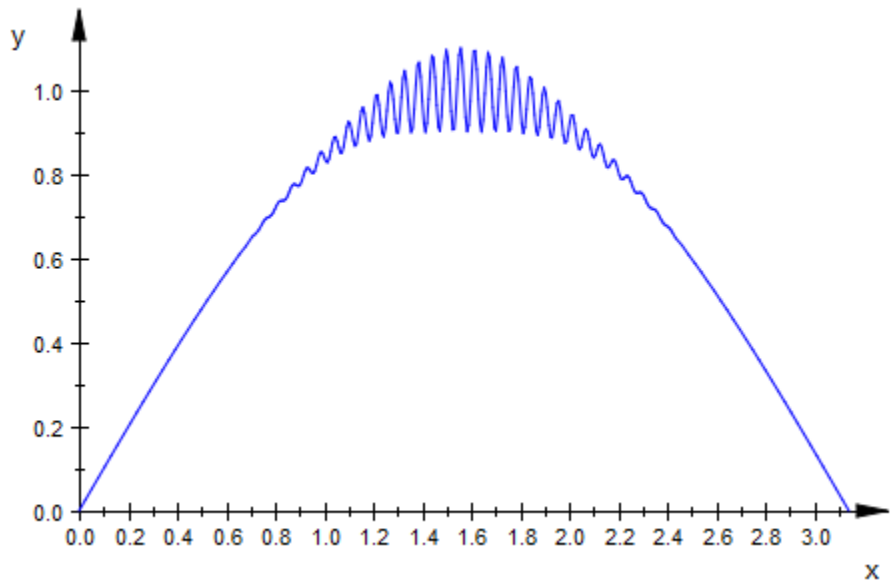
surfaces. The granularity of the "initial mesh" generated with these attribute values is approximately of the same size as the adaptive mesh generated with `AdaptiveMesh = n`, `Submesh = [0, 0]`. The non-adaptive evaluation on the refined regular mesh may still be more efficient than the evaluation on the (irregular) non-adaptive mesh.

Examples

Example 1

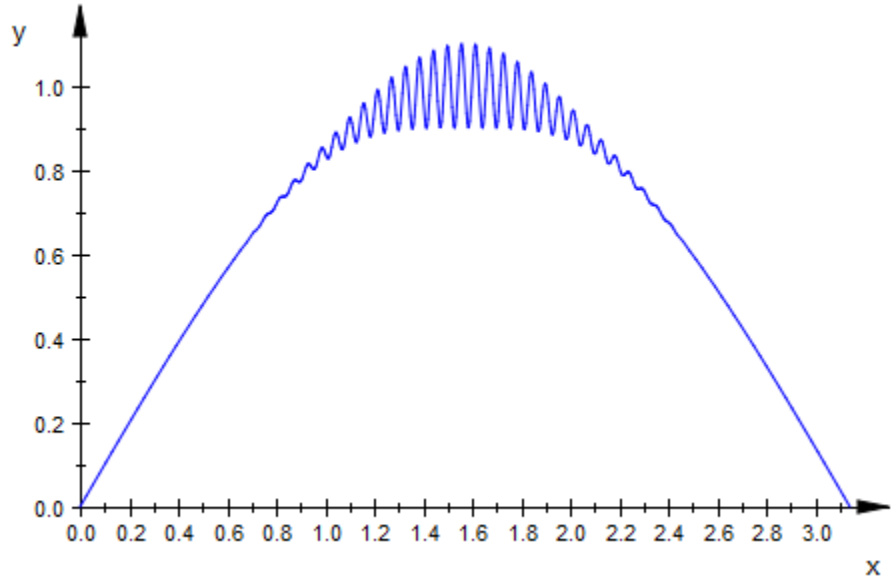
The following function plot contains areas of high variation. Without a specification of `AdaptiveMesh`, the default mode `AdaptiveMesh = 0` is used and we clearly see artifacts caused by the evaluation on a discrete mesh:

```
plot(plot::Function2d(
     sin(x) + exp(-5*(x - PI/2)^2)*sin(110*x)/10, x = 0..PI)):
```



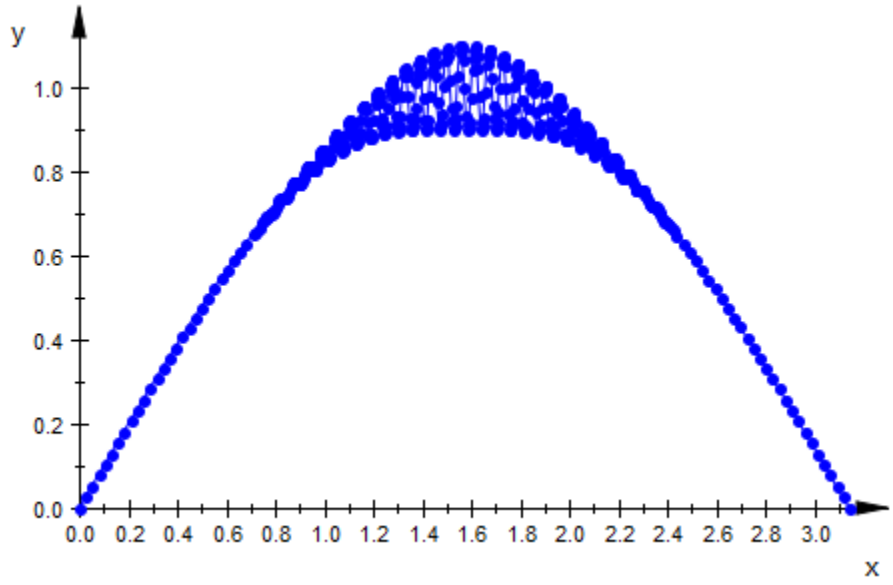
We activate the adaptive refinement with a high level of 3:

```
plot(plot::Function2d(
     sin(x) + exp(-5*(x - PI/2)^2)*sin(110*x)/10, x = 0..PI,
     AdaptiveMesh = 3)):
```



We set the attribute `PointsVisible = TRUE` so that the points of the adaptive mesh become visible:

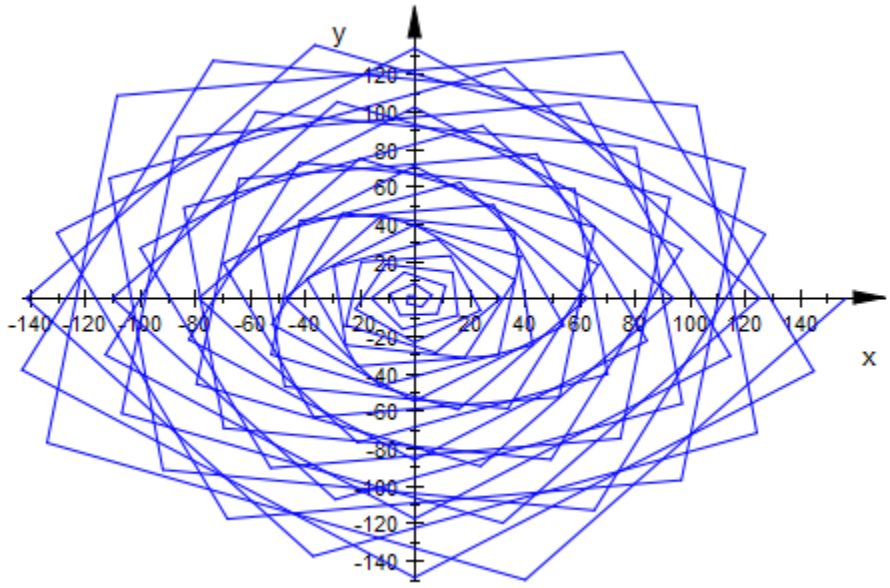
```
plot(plot::Function2d(
     sin(x) + exp(-5*(x - PI/2)^2)*sin(110*x)/10, x = 0..PI,
     AdaptiveMesh = 3, PointsVisible = TRUE)):
```



Example 2

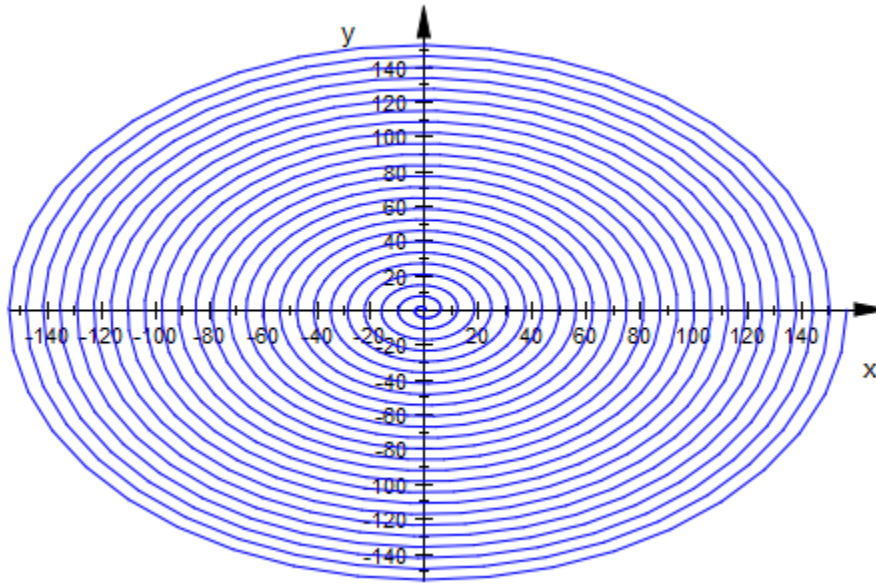
The default value of `Mesh` does not provide a sufficient resolution for the following spiral:

```
plot(plot::Curve2d([x*cos(x), x*sin(x)], x = 0..50*PI)):
```



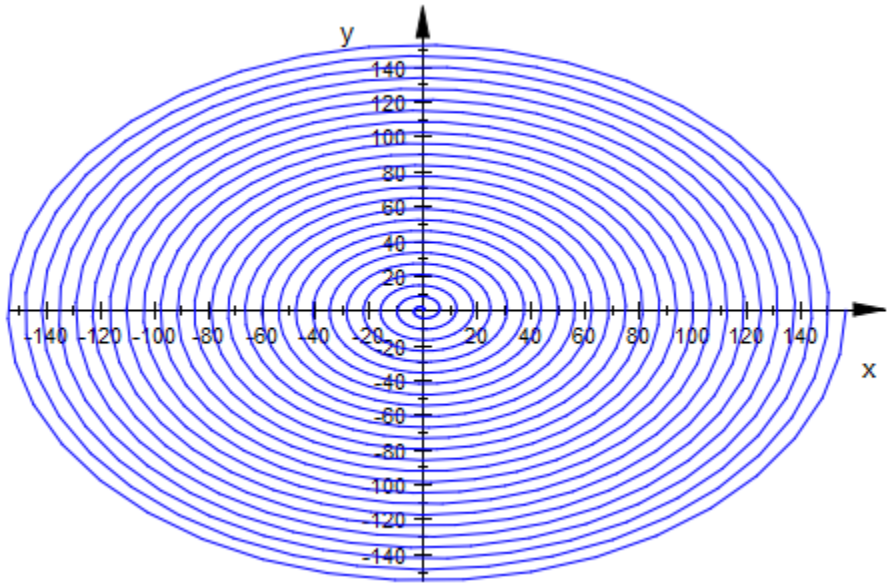
Increasing the `Mesh` value improves the plot:

```
plot(plot::Curve2d([x*cos(x), x*sin(x)], x = 0..50*PI,
                   Mesh = 1000)):
```



Alternatively, adaptive plotting can be used:

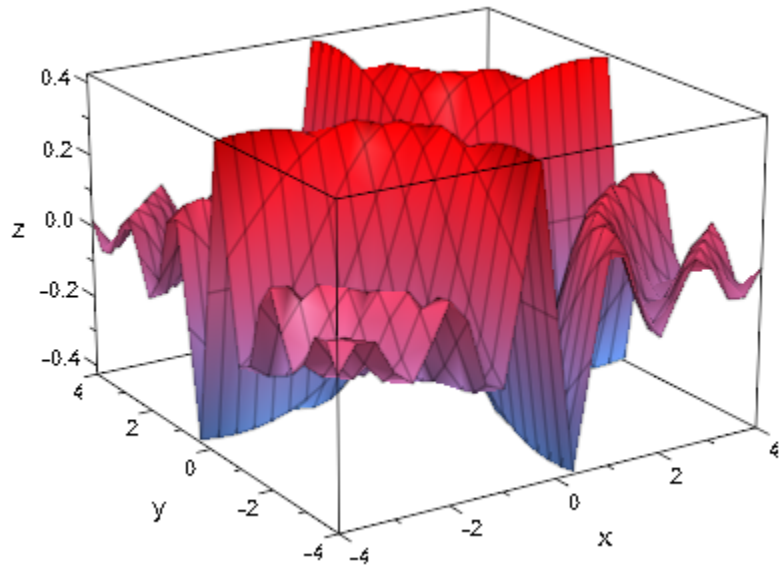
```
plot(plot::Curve2d([x*cos(x), x*sin(x)], x = 0..50*PI,
                   AdaptiveMesh = 3)):
```



Example 3

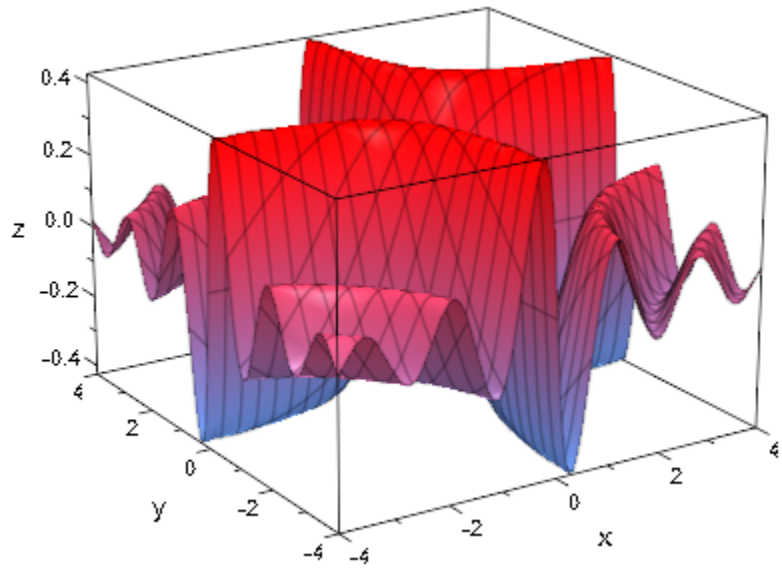
In 3D the typical artifacts caused by the rectilinear initial mesh are “dents” on surface features that are not parallel to a parameter axis. Without a specification of `AdaptiveMesh`, the default mode `AdaptiveMesh = 0` is used:

```
f := plot::Function3d(sin(x*y)/(abs(x*y) + 1),
                      x = -4 .. 4, y = -4 .. 4):
plot(f):
```



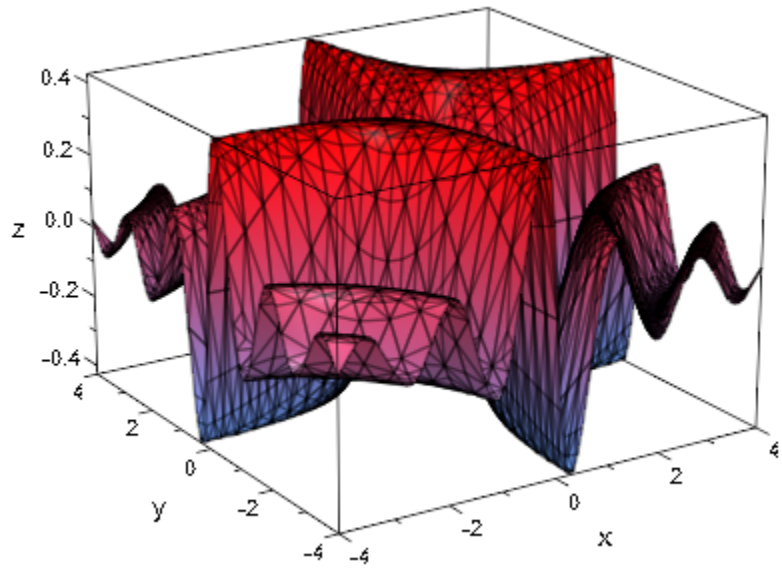
Activating the adaptive refinement, we get a much more accurate plot. However, the computation takes *much longer*:

```
plot(f, AdaptiveMesh = 2):
```



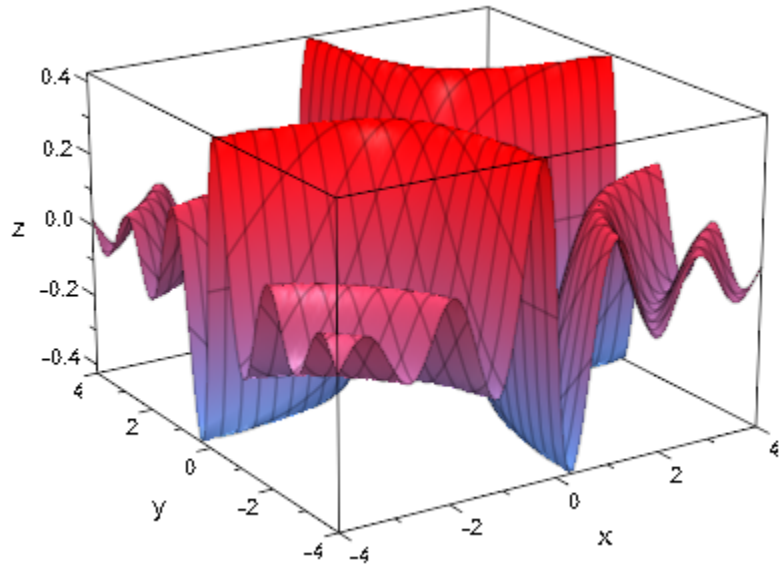
To see how local the refinement is, we set the attribute `MeshVisible = TRUE` so that the internal triangulation of the adaptive mesh becomes visible:

```
plot(f, AdaptiveMesh = 2, MeshVisible = TRUE):
```



We use a non-adaptive evaluation, but refine the regular mesh by setting `Submesh` values 2^n - 1 that correspond to the adaptive depth n = 2 used above. The result is of a similar quality as before:

```
plot(plot::Function3d(sin(x*y)/(abs(x*y) + 1),
                      x = -4 .. 4, y = -4 .. 4,
                      Submesh = [3, 3])):
```




```
delete f:
```

See Also

MuPAD Functions

`Mesh` | `MeshVisible` | `Submesh` | `UMesh` | `USubmesh` | `VMesh` | `VSubmesh` | `XMesh` | `XSubmesh` | `YMesh` | `YSubmesh` | `ZMesh`

DiscontinuitySearch

Semi-symbolic search for discontinuities

Value Summary

Inherited	FALSE, or TRUE

Graphics Primitives

Objects	DiscontinuitySearch Default Values
plot::Curve2d, plot::Curve3d, plot::Function2d, plot::Polar, plot::Sweep	TRUE

Description

DiscontinuitySearch = TRUE versus DiscontinuitySearch = FALSE determines whether a graphical object is checked (semi-)symbolically for discontinuities and singularities.

Certain graphical objects such as function graphs or parametrized curves may have singularities. This may create graphical artifacts such as spurious lines between numerical sample points that enclose a singularity. With DiscontinuitySearch = TRUE, the object is pre-processed to find potential singularities. If singular points are found, the object is split into several disjoint sub-objects (“branches”), each of which is smooth.

Note: DiscontinuitySearch is only available for line objects (2D function graphs and parametrized curves in 2D and 3D). It is not available for surface objects such as 3D function graphs and parametrized surfaces!

Discontinuities will only be detected if they are caused by system functions that are implemented as a function environment with an appropriate "realDiscont" or "numericDiscont" slot.

The search for discontinuities uses interval arithmetic. If special functions are involved that do not support this kind of arithmetic, the search will not succeed.

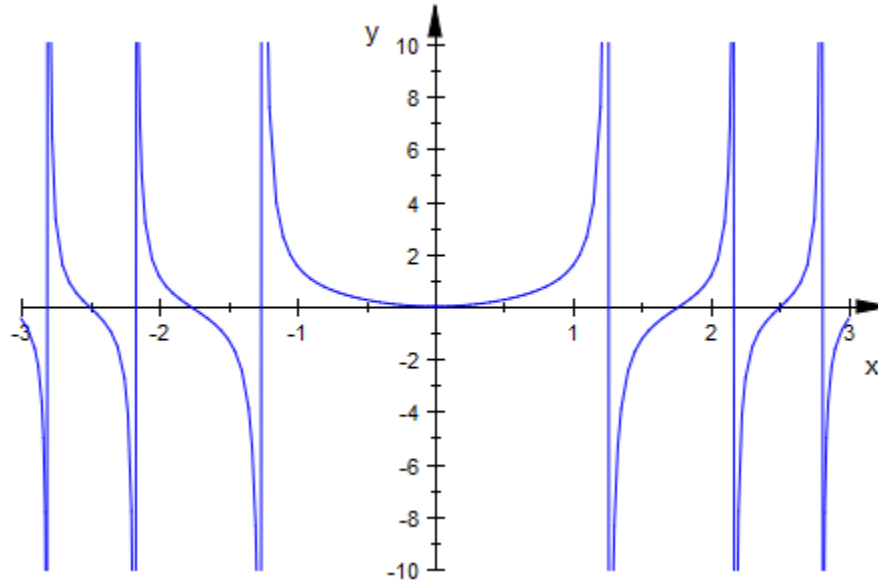
For efficiency reasons, it is recommended to disable the search for discontinuities with `DiscontinuitySearch` = `FALSE` when it is known that the graphical object is continuous.

Examples

Example 1

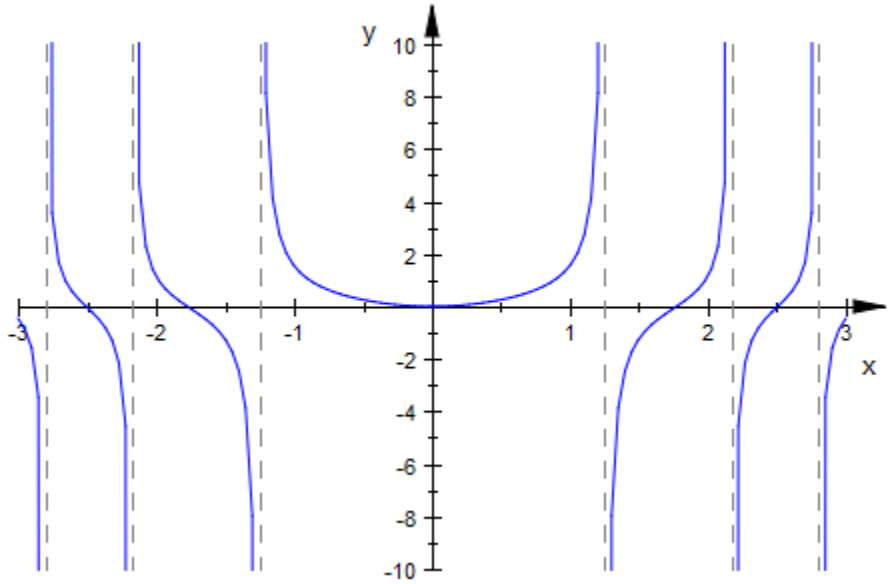
The following plot contains first order poles. When the discontinuity search is disabled, spurious vertical lines occur connecting sample points to the left of a pole with neighbouring sample points to the right of the pole. Further, the neighbourhood of the poles is poorly sampled:

```
plot(plot::Function2d(tan(x^2), x=-3..3,
                      ViewingBoxYRange = -10..10,
                      DiscontinuitySearch = FALSE)):
```



Without specification of `DiscontinuitySearch`, the default setting `DiscontinuitySearch = TRUE` is used. The spurious lines disappear. With the default `VerticalAsymptotesVisible = TRUE`, they are replaced by dashed vertical asymptotes indicating the poles. Also note that the numerical sampling near the poles is better, because the existence of the singularities and their positions is known before the numerical evaluation of the function graph starts:

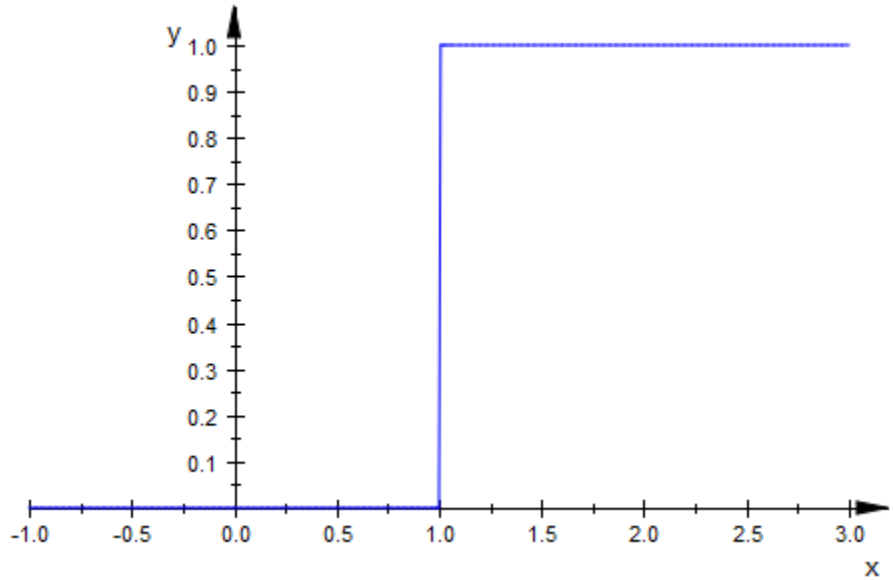
```
plot(plot::Function2d(tan(x^2), x=-3..3,
                      ViewingBoxYRange = -10..10)):
```



Example 2

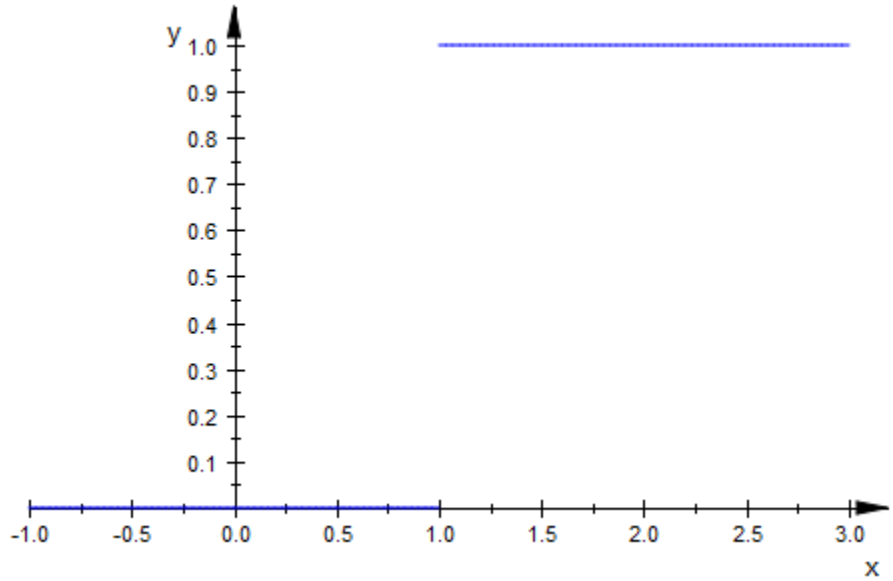
The Heaviside function has a jump discontinuity. Without a discontinuity search, a spurious line connecting the left and the right limit points of the jump appears:

```
plot(plot::Function2d(heaviside(x-1), x = -1..3,
                      DiscontinuitySearch = FALSE)):
```



This spurious line disappears with the default setting `DiscontinuitySearch` = `TRUE`:

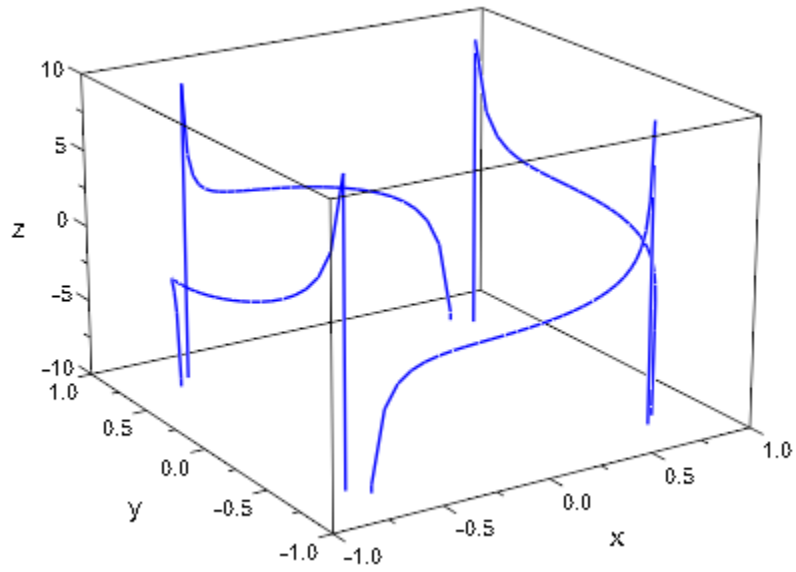
```
plot(plot::Function2d(heaviside(x-1), x = -1..3),
                      VerticalAsymptotesVisible = FALSE):
```



Example 3

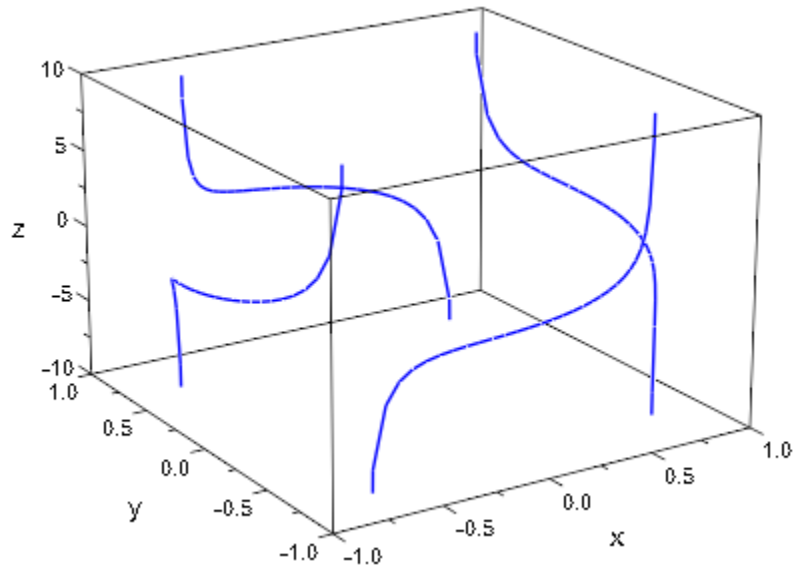
Without a discontinuity search, the poles of the following singular 3D curve are poorly presented graphically:

```
plot(plot::Curve3d([cos(u), sin(u), tan(2*u)],  u = 0..2*PI,
                   ViewingBox = [-1..1, -1..1, -10..10],
                   DiscontinuitySearch = FALSE)):
```



The default setting `DiscontinuitySearch = TRUE` produces a better graphical presentation:

```
plot(plot::Curve3d([cos(u), sin(u), tan(2*u)],  u = 0..2*PI,
                   ViewingBox = [-1..1, -1..1, -10..10])):
```


10
5
0
-5
-10
z
1.0
0.5
0.0
-0.5
-1.0
y
-1.0
-0.5
0.0
0.5
1.0
x

Mesh, Submesh

Number of sample points

Value Summary

`Mesh`	Library wrapper for "`UMesh`, `VMesh`, `XMesh`, `YMesh`, and `ZMesh`"	See below
`Submesh`	Library wrapper for "`USubmesh`, `VSubmesh`, `XSubmesh`, and `YSubmesh`"	See below

Graphics Primitives

Objects	Default Values
`plot::Cylindrical`, `plot::Function3d`, `plot::Spherical`, `plot::Surface`, `plot::XRotate`, `plot::ZRotate`	`Mesh`: [25, 25] `Submesh`: [0, 0]
`plot::Rootlocus`	`Mesh`: 51
`plot::Sweep`	`Mesh`: 25 `Submesh`: 4
`plot::Curve2d`, `plot::Curve3d`, `plot::Function2d`, `plot::Polar`	`Mesh`: 121 `Submesh`: 0
`plot::Conformal`	`Mesh`: [11, 11] `Submesh`: [0, 0]
`plot::Plane`	`Mesh`: [15, 15]
`plot::Implicit2d`, `plot::Raster`, `plot::VectorField2d`	`Mesh`: [11, 11]

Objects	Default Values
plot::VectorField3d	Mesh: [7, 7, 7]
plot::Implicit3d	Mesh: [11, 11, 11]
plot::Inequality	Mesh: [256, 256]
plot::Density	Mesh: [25, 25]
plot::Tube	Mesh: [60, 11] Submesh: [0, 1]
plot::Matrixplot	Submesh: [2, 2]
plot::Ode2d, plot::Ode3d	Submesh: 4
plot::Listplot	Submesh: 6

Description

The attributes Mesh and Submesh determine the number of sample points used for the numerical approximation of plot objects.

Many plot objects have to be evaluated numerically on a discrete mesh. Depending on the object type, there are type specific attributes such as XMesh (for 2D function graphs), UMesh, VMesh (for parametrized surfaces), XMesh, YMesh, ZMesh (for implicit plots in 3D) etc. setting the number of sample points of the numerical mesh.

The Mesh attribute unifies these more specific attributes and can be set for all objects that use a discrete numerical mesh. Depending on the object, the values for Mesh must be integer numbers or lists of such numbers. The more specific attributes are set automatically when Mesh values are specified.

E.g., in a 2D function plot of type plot::Function2d, Mesh = 200 is equivalent to XMesh = 200. In a 3D surface plot of type plot::Surface, Mesh = [40, 50] is equivalent to UMesh = 40, VMesh = 50.

In the “object inspector” of the interactive graphics tool (see section Viewer, Browser, and Inspector: Interactive Manipulation in this document), only the type specific attributes are visible, not the Mesh attribute.

Roughly speaking, high Mesh values yield smooth plots but cost run time.

With the attribute `Submesh = m`, additional *m* equidistant sample points are inserted between each pair of adjacent sample points set by the `Mesh` attribute. This smoothens the object.

Like `Mesh`, the attribute `Submesh` unifies type specific attributes such as `XSubmesh`, `USubmesh` etc. Depending on the object, the values of `Submesh` have to be integers or lists of integers.

There is a semantical difference between the "major" mesh points set by `Mesh` and the "minor" mesh points inserted by `Submesh`. There are coordinate lines associated with the (regular) numerical mesh. See `XLinesVisible`, `ULinesVisible` etc. The coordinates lines are available only for the mesh given by the "major" mesh points, whereas `Submesh` does not influence the number of coordinate lines. Thus, increased `Mesh` values yield a smoother plot with more coordinate lines, whereas `Submesh` can be used to smoothen the plot without adding further coordinate lines.

Apart from this effect, the pair `Mesh = n`, `Submesh = m` corresponds to the combination `Mesh = (n - 1) (m + 1) + 1`, `Submesh = 0`.

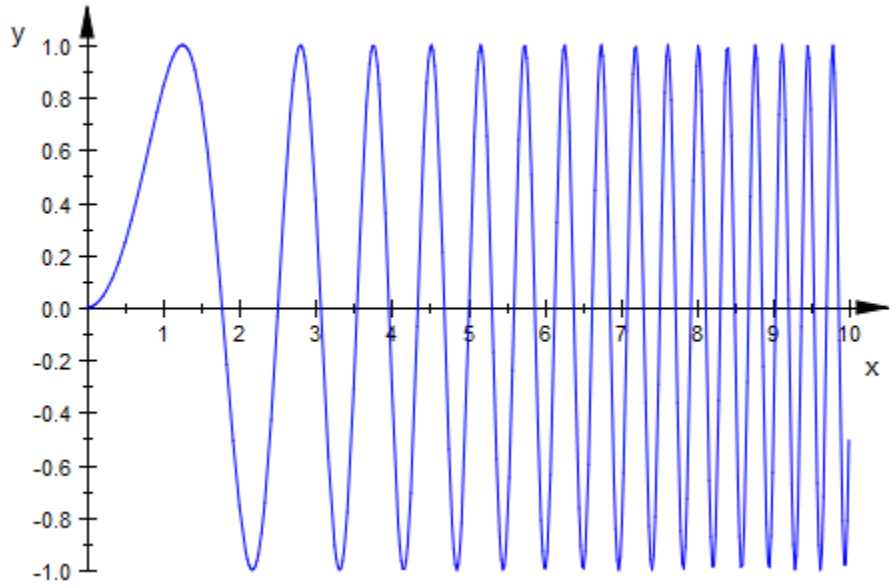
If adaptive sampling is enabled, further non-equidistant sample points are chosen automatically between the equidistant points of the `initial mesh' set via the `Mesh` and `Submesh` attributes.

Examples

Example 1

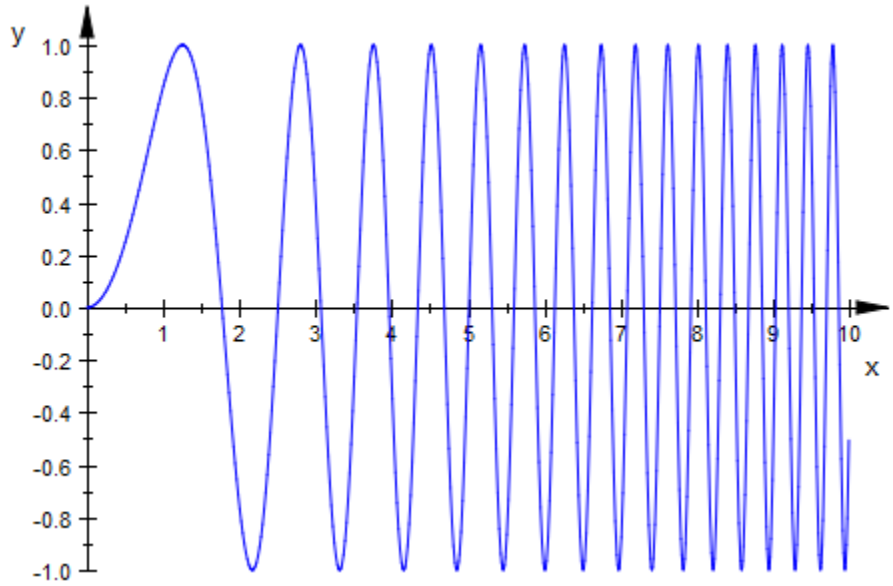
In the following plot, the default value of `Mesh` does not suffice to produce a sufficiently exact picture:

```
plot(plot::Function2d(sin(x^2), x = 0..10)):
```



A mesh with more sample points yields a higher resolution graphics:

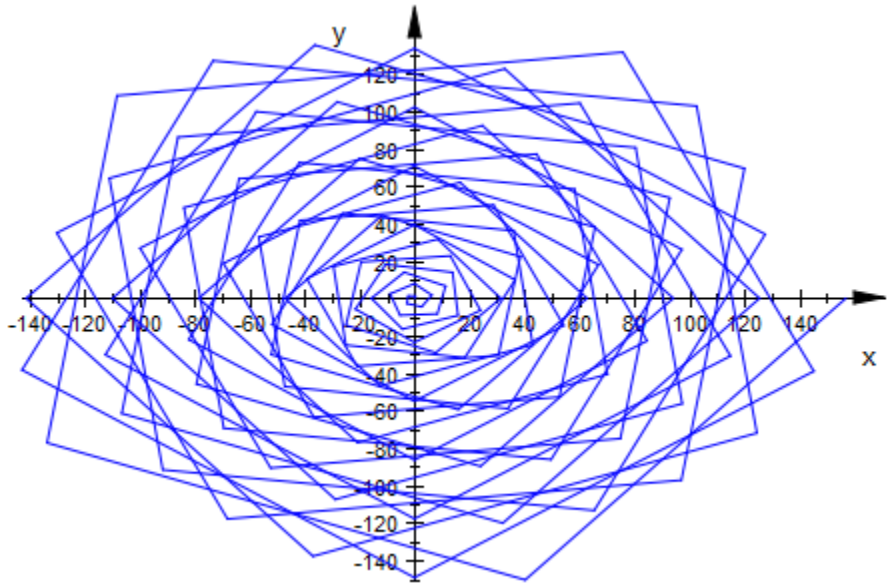
```
plot(plot::Function2d(sin(x^2), x = 0..10, Mesh = 500)):
```



Example 2

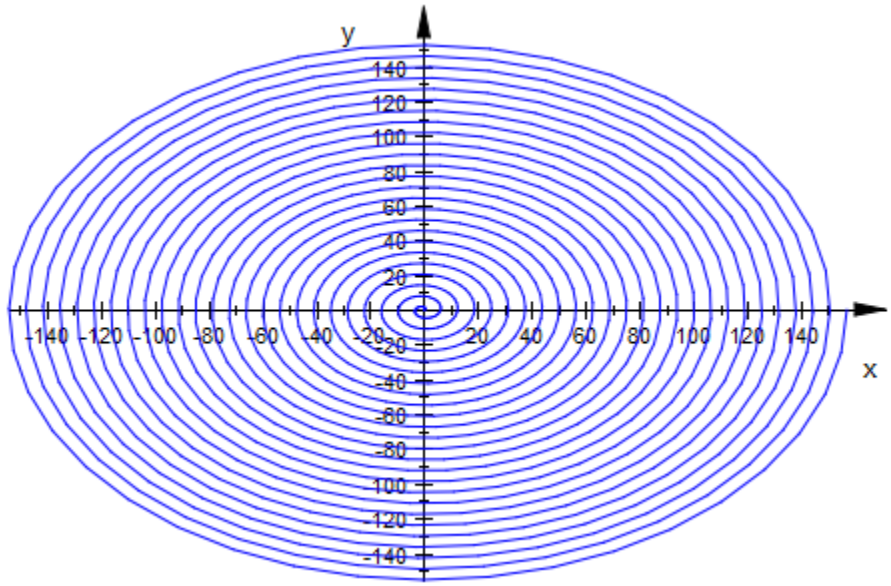
The default value of `Mesh` does not provide a sufficient resolution for the following spiral:

```
plot(plot::Curve2d([x*cos(x), x*sin(x)], x = 0..50*PI)):
```



The spiral winds around the origin 25 times. We wish to have approximately 40 sample points per revolution, so we need to use a total of 1000 sample points:

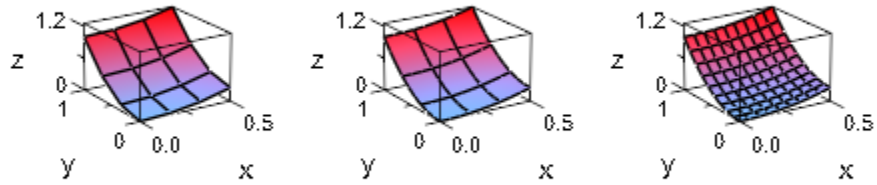
```
plot(plot::Curve2d([x*cos(x), x*sin(x)], x = 0..50*PI,
                   Mesh = 1000)):
```



Example 3

Note the difference between increased `Mesh` values and additional sample points inserted via `Submesh`. `Submesh` does not introduce additional coordinate lines:

```
S1 := plot::Scene3d(plot::Function3d(
        x^2 + y^2, x = 0..1/2, y = 0..1, Mesh = [4, 4])):
S2 := plot::Scene3d(plot::Function3d(
        x^2 + y^2, x = 0..1/2, y = 0..1, Mesh = [4, 4],
       Submesh = [2, 2])):
S3 := plot::Scene3d(plot::Function3d(
        x^2 + y^2, x = 0..1/2, y = 0..1, Mesh = [10, 10])):
plot(S1, S2, S3, Layout = Horizontal,
     Height = 5*unit::cm, Width = 12*unit::cm,
     LineColor = RGB::Black):
```




```
delete S1, S2, S3:
```

See Also

MuPAD Functions

AdaptiveMesh | UMesh | USubmesh | VMesh | VSubmesh | XMesh | YMesh | ZMesh

MinimumDistance

Space between stream lines

Value Summary

Optional	MuPAD expression

Graphics Primitives

Objects	MinimumDistance Default Values
`plot::Streamlines2d`	

Description

`MinimumDistance` determines how closely spaced the stream lines generated by a `plot::Streamlines2d` object are.

`plot::Streamlines2d` displays orbits (stream lines) of ODEs which are at least m and at most $2\,m$ units apart from one another, if `MinimumDistance` has been set to m.

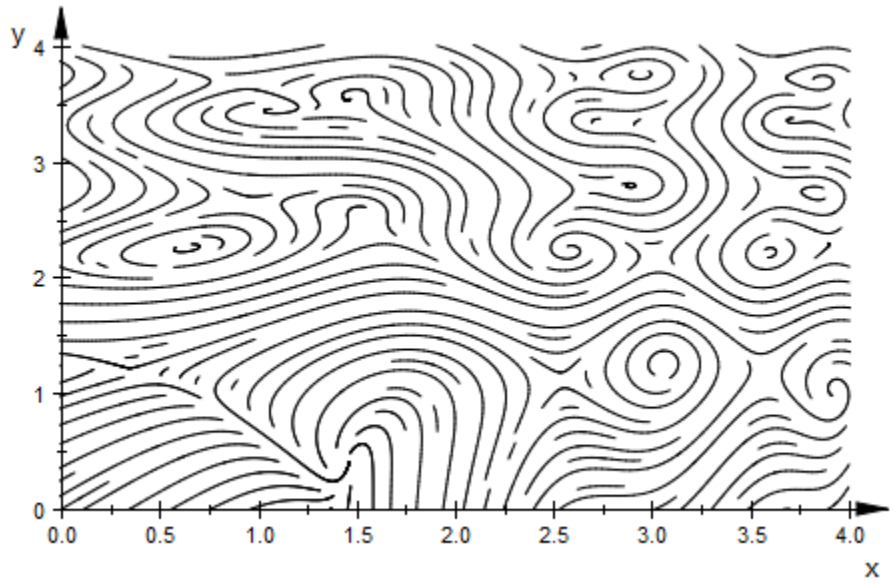
The distance of stream lines is taken as the Euclidean distance, measured in coordinate units. If `MinimumDistance` is not set, it defaults to 0.02 times the maximum extent in either direction.

Examples

Example 1

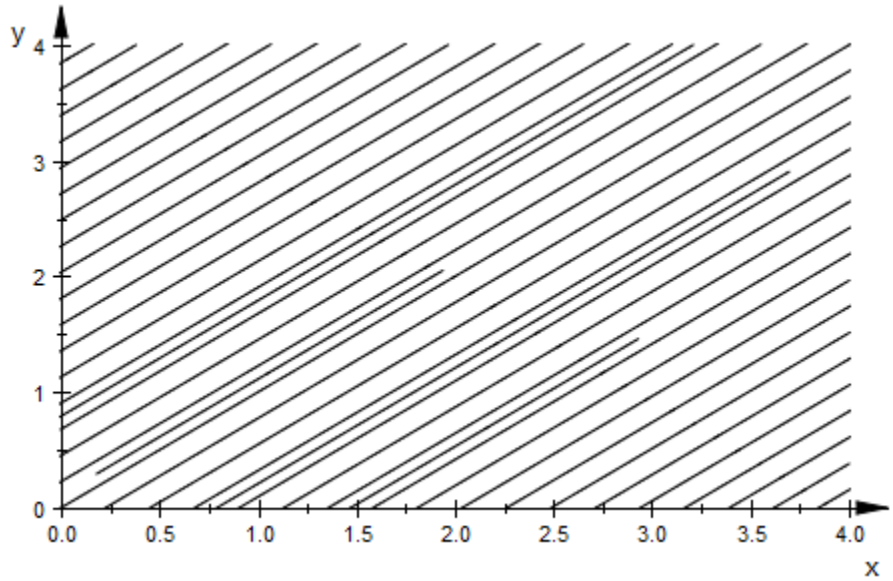
The default setting is adequate for many stream lines plots:

```
plot(plot::Streamlines2d([sin(x)^2-cos(y^2),
                          sin(x^2)-cos(y)^2],
                         x=0..4, y=0..4))
```

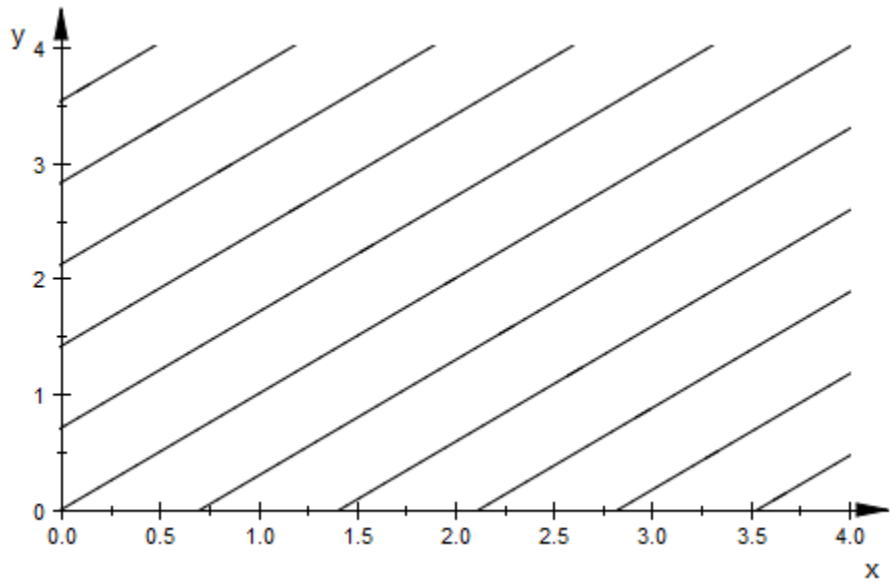


To display simple ODEs, you may wish to reduce the number of stream lines:

```
plot(plot::Streamlines2d([1, 1], x=0..4, y=0..4))
```




```
plot(plot::Streamlines2d([1, 1], x=0..4, y=0..4, MinimumDistance=0.25))
```


y
4
3
2
1
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
x

ODEMethod, Stepsize

Numerical scheme used for solving the ODE

Value Summary

`ODEMethod`, `Stepsize`	Optional	MuPAD expression

Graphics Primitives

Objects	Default Values
`plot::Ode2d`, `plot::Ode3d`	`ODEMethod`: `DOPRI78`
`plot::Streamlines2d`	`ODEMethod`: `ABM4`

Description

`ODEMethod = method` determines the numerical scheme for solving the ODE. The parameter `method` is a name such as `EULER1`, `RK4`, `RKF78` etc.

`Stepsize = h` sets a constant step size h that is used to compute the numerical solution.

Internally, `plot::Ode2d`, `plot::Ode3d`, and `plot::Streamlines2d` call the routine `numeric::odesolve` for solving the given ODE numerically. The method set by the attribute `ODEMethod = method` and/or the step size set by `Stepsize = h` are forwarded to `numeric::odesolve`. See the corresponding help page for a complete list of all methods available and for further details on the step size.

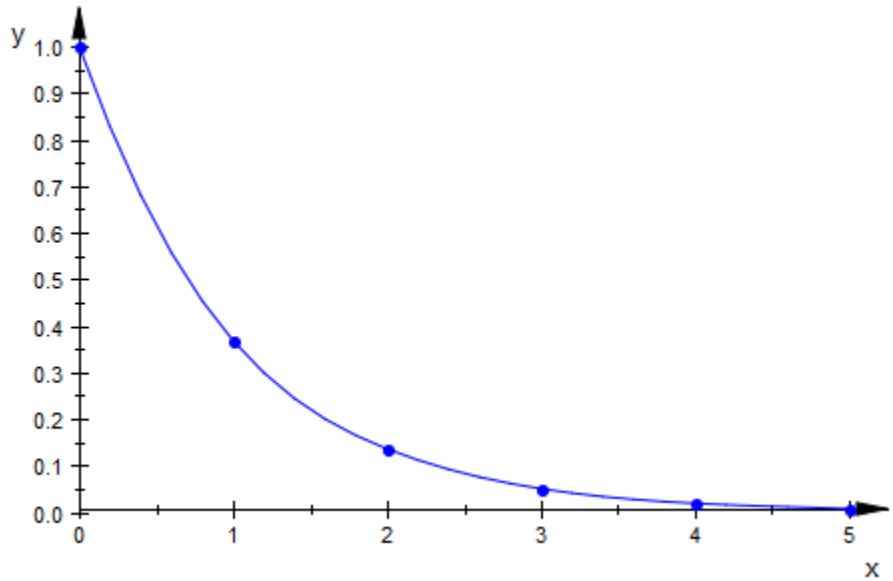
The setting `ODEMethod = ABM4` is an exception to the above: It is only available with `plot::Streamlines2d` and makes the plot use a fast Adams-Bashforth-Moulton predictor corrector integrator of fourth order with fixed step size. It ignores the settings of `RelativeError` and `AbsoluteError`.

Examples

Example 1

We solve the initial value problem $y'(t) = -y(t)$, $y(0) = 1$ numerically by the classical 4th order Runge-Kutta scheme `RK4` using a constant step size 0.1:

```
f := (t, Y) -> [-Y[1]]:
Y0 := [1]:
timemesh:= [0, 1, 2, 3, 4, 5]:
plot(plot::Ode2d(f, timemesh, Y0, ODEMethod = RK4,
                 Stepsize = 0.1))
```

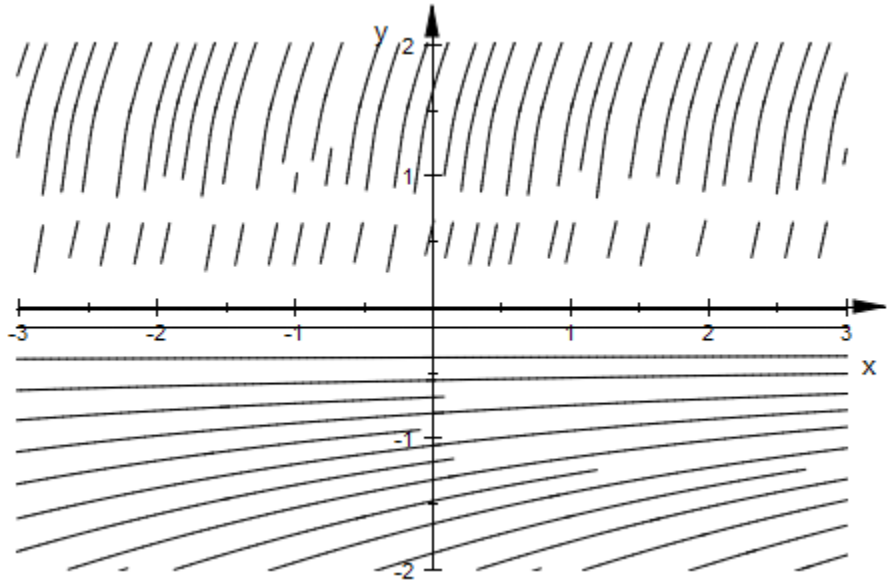



```
delete f, Y0, timemesh:
```

Example 2

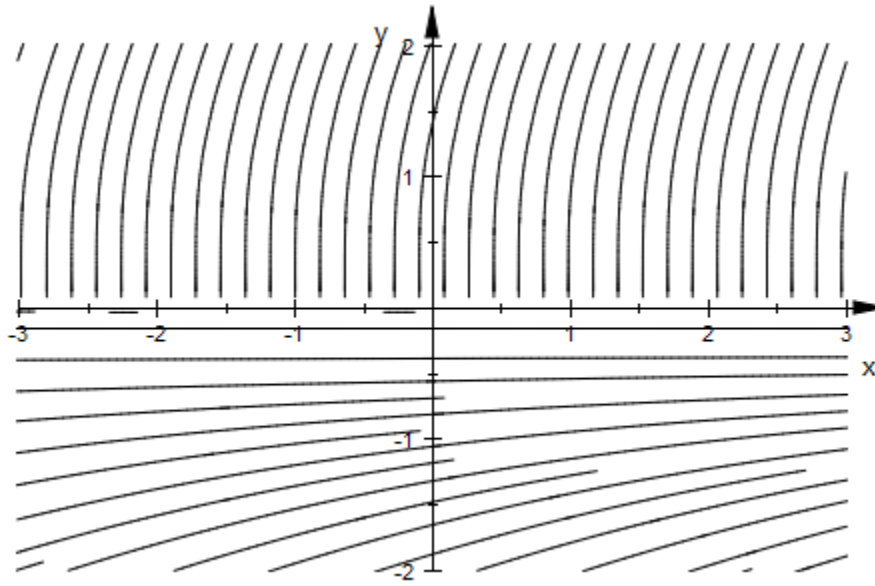
With the default settings, `plot::Streamlines2d` is not able to plot the vector field $\left[1, 3^{2/y}\right]$ (which is not Lipschitz continous) in a satisfying way:

```
plot(plot::Streamlines2d([1, surd(3,y)^2],
                         x=-3..3, y=-2..2))
```



By using a different numerical integrator, the problems can be overcome (at the cost of longer computation):

```
plot(plot::Streamlines2d([1, surd(3,y)^2],
                         x=-3..3, y=-2..2,
                         ODEMethod=RKF43,
                         RelativeError=1e-3))
```



See Also

MuPAD Functions

`AbsoluteError` | `InitialConditions` | `ODEMethod` | `Projectors` | `RelativeError`

UMesh, VMesh, USubmesh, VSubmesh

Number of sample points

Value Summary

`UMesh`, `USubmesh`, `VMesh`, `VSubmesh`	Inherited	Positive integer

Graphics Primitives

Objects	Default Values
`plot::Curve2d`, `plot::Curve3d`, `plot::Polar`	`UMesh`: 121 `USubmesh`: 0
`plot::Cylindrical`, `plot::Spherical`, `plot::Surface`, `plot::XRotate`, `plot::ZRotate`	`UMesh`, `VMesh`: 25 `USubmesh`, `VSubmesh`: 0
`plot::Rootlocus`	`UMesh`: 51
`plot::Sweep`	`UMesh`: 25 `USubmesh`: 4
`plot::Plane`	`UMesh`, `VMesh`: 15
`plot::Tube`	`UMesh`: 60 `VMesh`: 11 `USubmesh`: 0 `VSubmesh`: 1
`plot::Ode2d`, `plot::Ode3d`	`USubmesh`: 4

Description

The attributes `UMesh` etc. determine the number of sample points used for the numerical approximation of parameterized plot objects such as curves and surfaces.

Many plot objects have to be evaluated numerically on a discrete mesh. The attributes described on this help page serve for setting the number of sample points of the numerical mesh.

For curves in 2D and 3D given by a parametrization $x(u)$, $y(u)$ and, possibly, $z(u)$ with the curve parameter u, the attribute `UMesh = n` creates a numerical mesh of n equidistant u values. The attribute `USubmesh = m` inserts additional m mesh points between each pair of adjacent points set by `UMesh`.

The combinations `UMesh = n`, `USubmesh = m` and `UMesh = (m + 1) (n - 1) + 1`, `USubmesh = 0` are equivalent.

Specifying `Mesh`, `Submesh` has the same effekt as specifying `UMesh`, `USubmesh`.

The sample points of a curve can be made visible by setting `PointsVisible = TRUE`.

Surface objects in 3D are parameterized by coordinate functions $x(u, v)$, $y(u, v)$, $z(u, v)$ of two surface parameters u, v.

The attribute `UMesh = nu` sets the number n_u of sample points for the first surface parameter. The attribute `VMesh = nv` sets the number nv of sample points for the second surface parameter. The parametrization is evaluated on a regular mesh of $n_u \times n_v$ values of the surface parameters u, v.

With the `USubmesh`, `VSubmesh` attributes, additional equidistant sample points can be inserted between each pair of adjacent sample points set by the `UMesh`, `VMesh` attributes.

With `ULinesVisible = TRUE` and `VLinesVisible = TRUE`, respectively, the parameter lines of the regular mesh set by the attributes `UMesh`, `VMesh` are displayed on the surface. Additonal points inserted via `USubmesh`, `VSubmesh` do *not* create additional parameter lines.

You can also specify `UMesh = nu`, `VMesh = nv`, `USubmesh = mu`, `VSubmesh = mv` in the shorter form `Mesh = [nu, nv]`, `Submesh = [mu, mv]`.

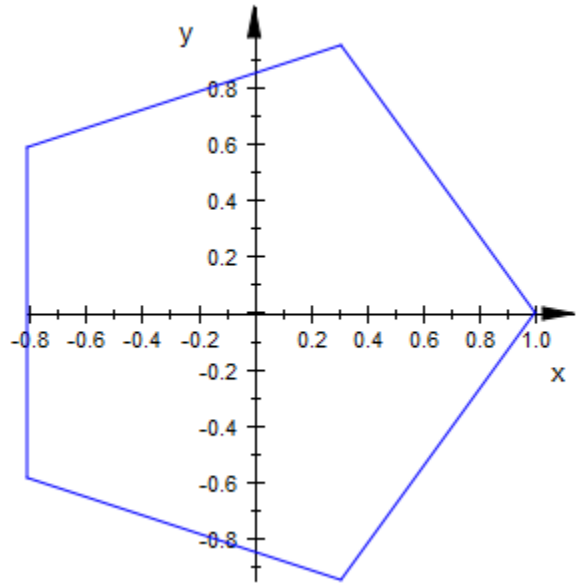
If adaptive sampling is enabled, further non-equidistant sample points are chosen automatically between the equidistant points of the `initial mesh' set via the `UMesh`, `USubmesh`, `VMesh`, `VSubmesh` attributes.

Examples

Example 1

It is possible to use low settings of mesh parameters to achieve special effects. As an example, we draw a parametrization of a circle with just six evaluation points:

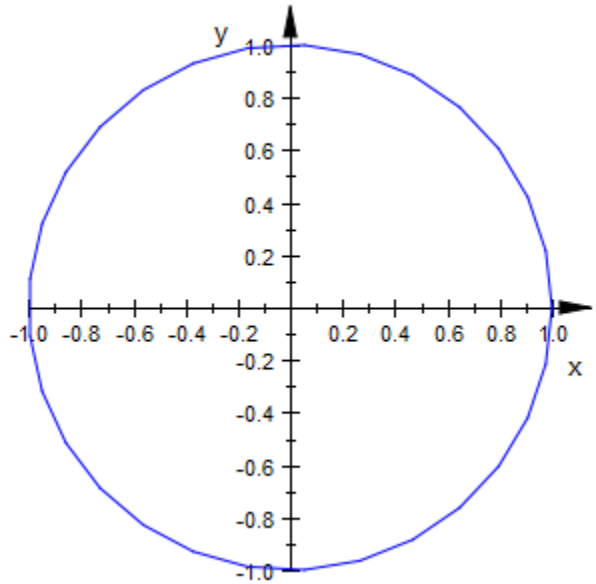
```
plot(plot::Curve2d([cos(t), sin(t)], t = 0..2*PI, UMesh = 6,
     Scaling = Constrained))
```



The reason we get a pentagon here and not a hexagon is that the first and the last evaluation points coincide: six points in a line means five line segments.

With `UMesh = 30`, the circle looks like a circle:

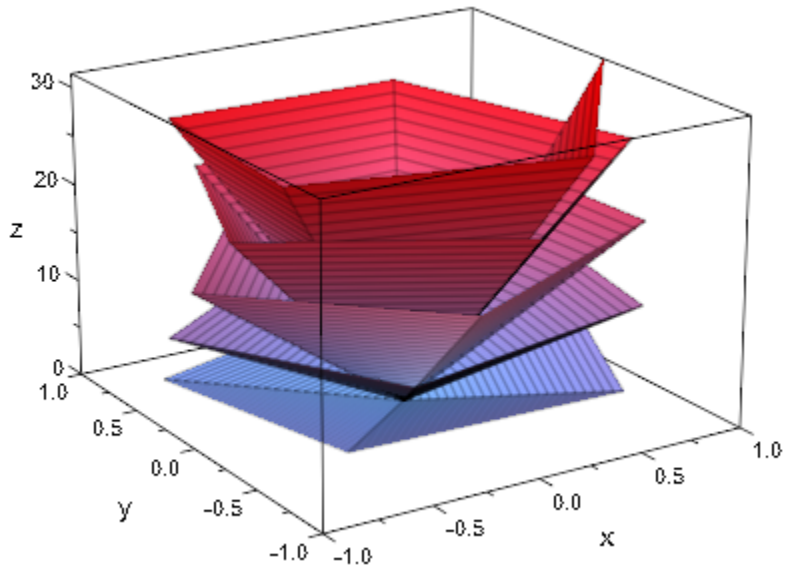
```
plot(plot::Curve2d([cos(t), sin(t)], t = 0..2*PI, UMesh = 30,
     Scaling = Constrained))
```



Example 2

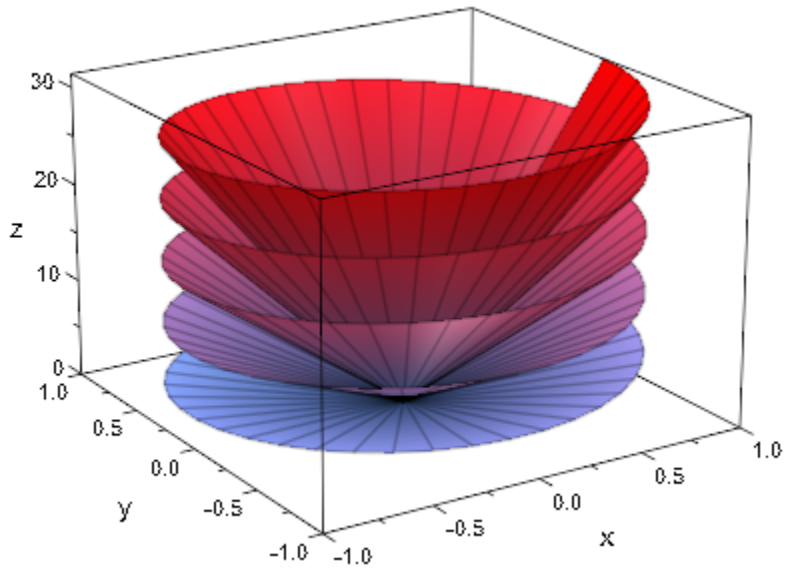
The default values of `UMesh`, `VMesh` do not provide a sufficient resolution for the following graphics:

```
plot(plot::Surface([r*cos(phi), r*sin(phi), r*phi],
                   r = 0.. 1, phi = 0..10*PI)):
```



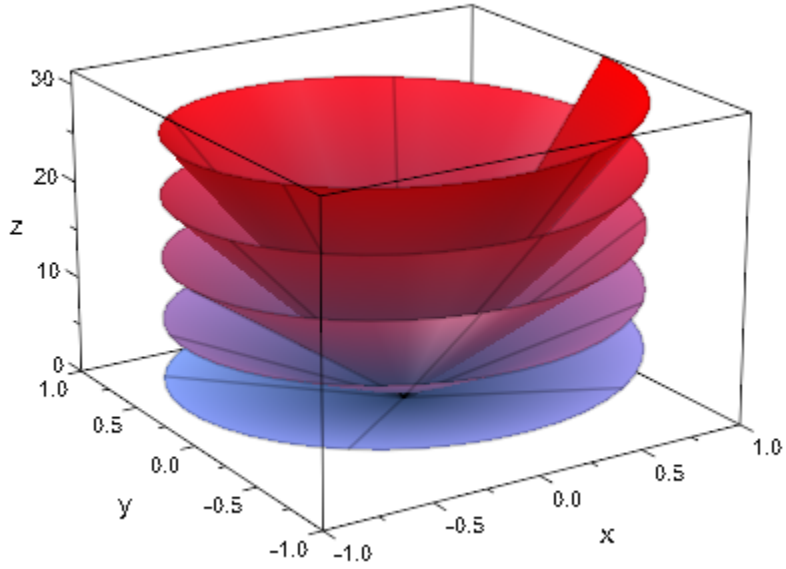
The spiral winds around the *z*-axis 5 times. We wish to have approximately 40 sample points per revolution, so we need to use a total of 200 sample points with respect to the angle parameter `phi`. The coordinate lines related to the radial parameter `r` are straight lines, so a very low resolution in this direction suffices:

```
plot(plot::Surface([r*cos(phi), r*sin(phi), r*phi],
                   r = 0.. 1, phi = 0..10*PI,
                   UMesh = 2, VMesh = 200)):
```



When refining the mesh via `VSubmesh`, no additional parameter lines are created:

```
plot(plot::Surface([r*cos(phi), r*sin(phi), r*phi],
                   r = 0.. 1, phi = 0..10*PI,
                   UMesh = 2, VMesh = 25, VSubmesh = 8)):
```



See Also

MuPAD Functions

`AdaptiveMesh` | `Mesh` | `Submesh` | `XMesh` | `YMesh` | `ZMesh`

XMesh, XSubmesh, YMesh, YSubmesh, ZMesh

Number of sample points

Value Summary

`XMesh`, `XSubmesh`, `YMesh`, `YSubmesh`, `ZMesh`	Inherited	Positive integer

Graphics Primitives

Objects	Default Values
`plot::Function2d`	`XMesh`: 121 `XSubmesh`: 0
`plot::Function3d`	`XMesh`, `YMesh`: 25 `XSubmesh`, `YSubmesh`: 0
`plot::Implicit2d`, `plot::Raster`, `plot::VectorField2d`	`XMesh`, `YMesh`: 11
`plot::Implicit3d`	`XMesh`, `YMesh`, `ZMesh`: 11
`plot::VectorField3d`	`XMesh`, `YMesh`, `ZMesh`: 7
`plot::Conformal`	`XMesh`, `YMesh`: 11 `XSubmesh`, `YSubmesh`: 0
`plot::Inequality`	`XMesh`, `YMesh`: 256
`plot::Density`	`XMesh`, `YMesh`: 25
`plot::Matrixplot`	`XSubmesh`, `YSubmesh`: 2
`plot::Listplot`	`XSubmesh`: 6

Description

The attributes `XMesh` etc. determine the number of sample points used for the numerical approximation of plot objects such as function graphs, implicit plots etc.

Various object types use numerical function evaluations on a discrete equidistant mesh. `XMesh`, `YMesh`, and for `plot::Implicit3d` also `ZMesh` set the number of points of this mesh. An exception of this are parameterized curves and surfaces, which use the attributes `UMesh`, `USubmesh`, `VMesh`, and `VSubmesh`.

For most of the object types listed above, the interpretation of the integers set by these attributes is as follows: In each of `XRange`, `YRange`, `ZRange`, the corresponding number of points is spread out equidistantly. For `XMesh = 2` and `XRange = 0..1`, for example, evaluation takes place for $x = 0$ and $x = 1$. For `XMesh = 3`, a further mesh point at $x = \frac{1}{2}$ is used.

The exception to this rule is `plot::Implicit2d`: Here, the values of `XMesh` and `YMesh` determine the density of the grid *usd for finding components* and increasing their values helps exactly in those cases where components (i.e., lines) are missing from the plot. Decreasing `XMesh` and `YMesh` in a 2D implicit plot will not make the curves appear rougher; it may result in curves missing.

For types reacting to `AdaptiveMesh` and for `plot::Implicit3d`, this mesh is used to find *initial* values that can be refined further. See the documentation of the specific types and of `AdaptiveMesh` for details.

In general, a finer mesh (higher values) leads to a longer computation, while a coarser mesh may cause details being missed.

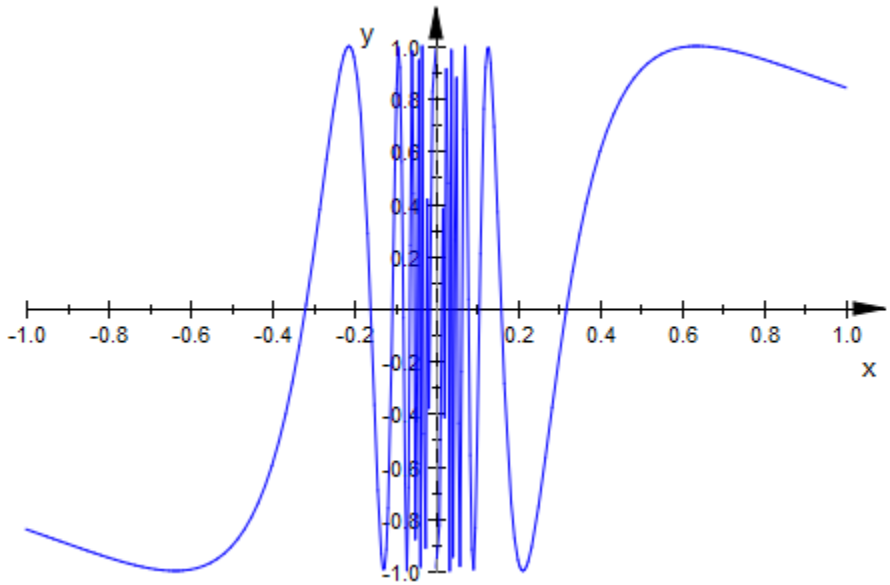
One may specify `XMesh` = n_x, `YMesh` = n_y, `XSubmesh` = m_x, `YSubmesh` = m_y also in the shorter Form `Mesh` = `[`n_x`, `n_y`]`, `Submesh` = `[`m_x`, `m_y`]`.

Examples

Example 1

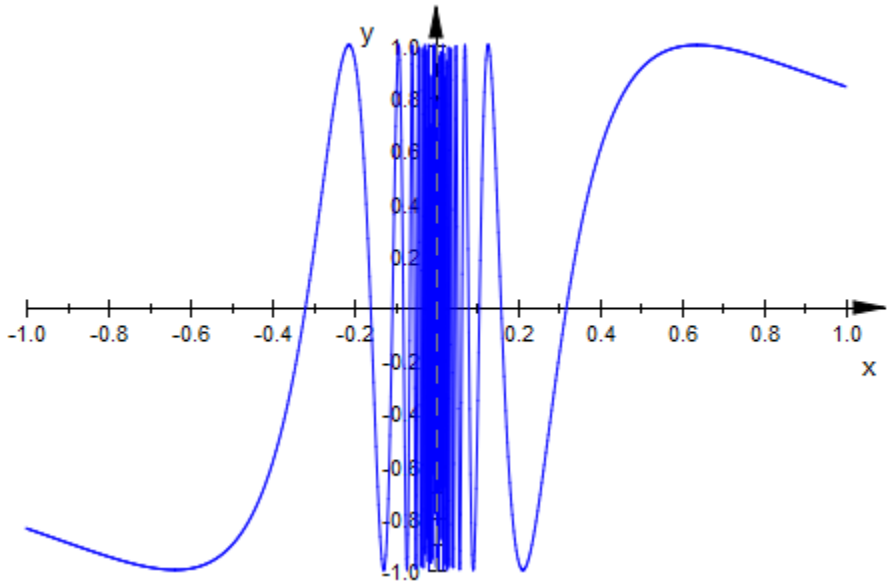
The notorious function $\sin\left(\frac{1}{x}\right)$ oscillates wildly near the origin. The standard mesh values do not suffice to resolve the behavior of the function near the critical point:

```
plot(plot::Function2d(sin(1/x), x = -1 .. 1))
```



We get a better result with an increased value of `XMesh`:

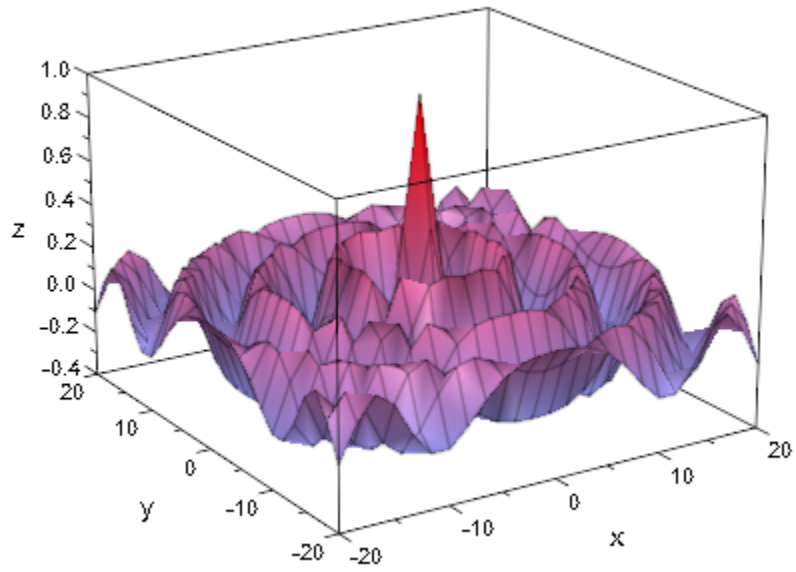
```
plot(plot::Function2d(sin(1/x), x = -1 .. 1), XMesh = 1000)
```



Example 2

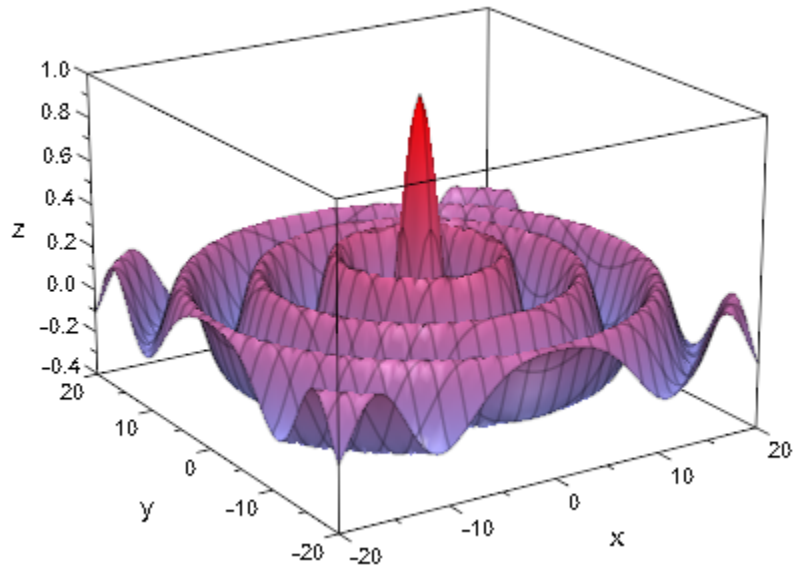
In the following plot, the default values of `XMesh`, `YMesh` do not suffice to produce a sufficiently smooth function graph:

```
plot(plot::Function3d(besselJ(0, sqrt(x^2 + y^2)),
                      x = -20 .. 20, y = -20 .. 20)):
```



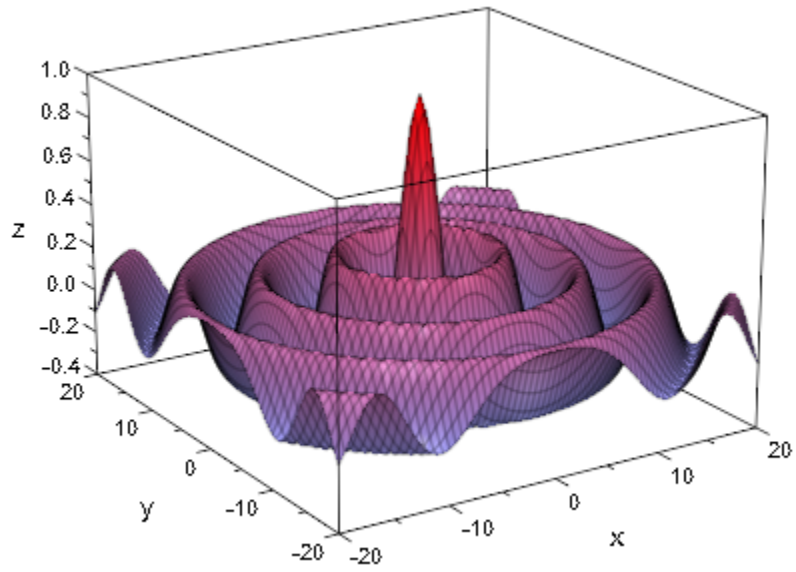
Increasing the default values `XSubmesh = 0`, `YSubmesh = 0` yields a higher resolution plot. Note that this does not influence the number of mesh lines that are displayed:

```
plot(plot::Function3d(besselJ(0, sqrt(x^2 + y^2)),
                      x = -20 .. 20, y = -20 .. 20,
                      XSubmesh = 2, YSubmesh = 2)):
```



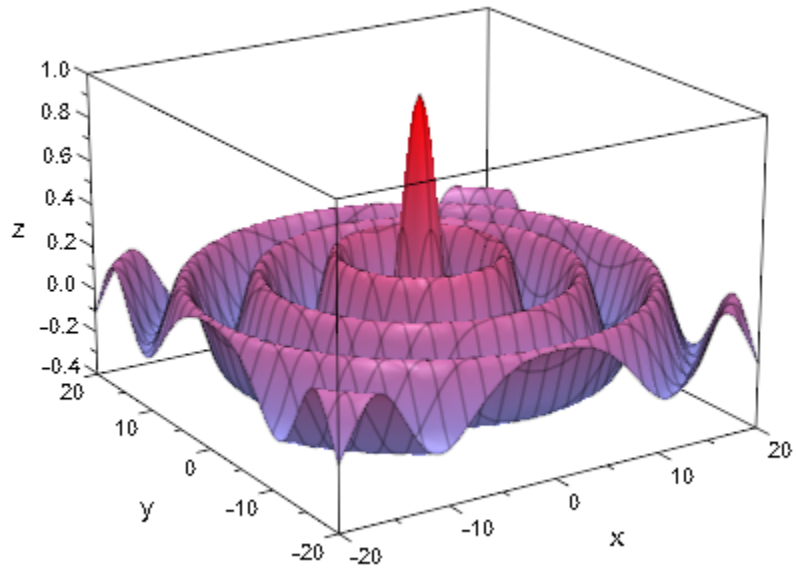
Alternatively, we increase the values of `XMesh`, `YMesh` and use the default values `XSubmesh = 0`, `YSubmesh = 0`. This, however, increases the number of mesh lines that are displayed:

```
plot(plot::Function3d(besselJ(0, sqrt(x^2 + y^2)),
                      x = -20 .. 20, y = -20 .. 20,
                      XMesh = 73, YMesh = 73)):
```



Yet another possibility is to use the default values of `XMesh`, `YMesh`, `XSubmesh`, `YSubmesh` and activate the adaptive mechanism to smoothen the critical regions of the plot. However, this plot consists almost completely of critical regions and the adaptive mechanism will therefore be slower than a direct calculation with a finer mesh that leads to almost the same result:

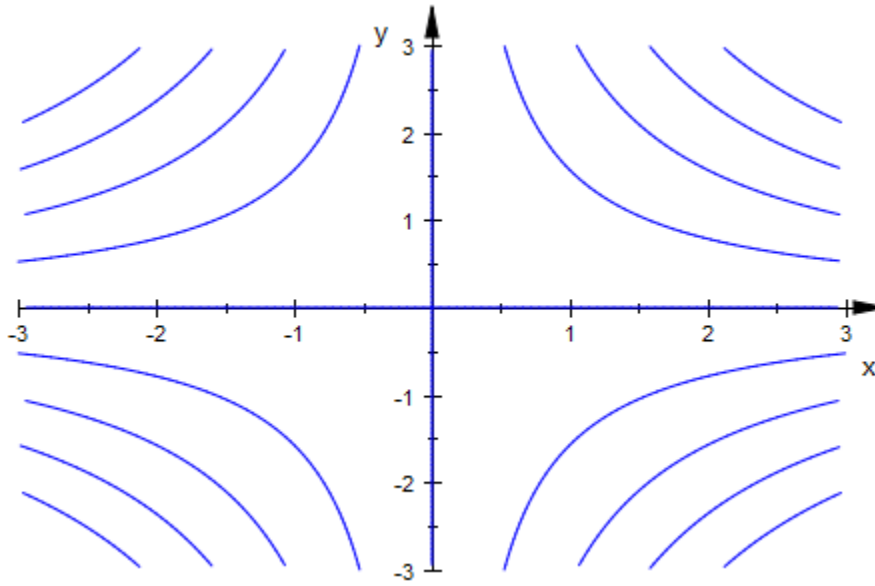
```
plot(plot::Function3d(besselJ(0, sqrt(x^2 + y^2)),
                      x = -20 .. 20, y = -20 .. 20,
                      AdaptiveMesh = 2)):
```



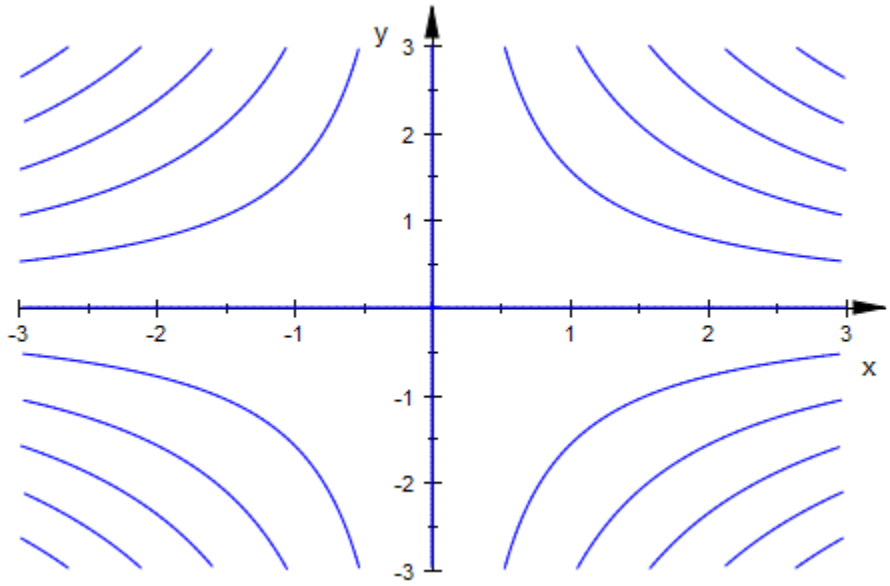
Example 3

For two-dimensional implicit plots, `XMesh` and `YMesh` determine the mesh of "seed points" that are used to find components (see the documentation of `plot::Implicit2d` for more details). In effect, this means that if some components are missing from a plot, the values of these attributes should be increased:

```
plot(plot::Implicit2d(sin(2*x*y), x = -3..3, y = -3..3))
```




```
plot(plot::Implicit2d(sin(2*x*y), x = -3..3, y = -3..3,
                      XMesh = 20, YMesh = 20))
```



See Also

MuPAD Functions

`AdaptiveMesh` | `Mesh` | `Submesh` | `UMesh` | `USubmesh` | `VMesh` | `VSubmesh`

CameraCoordinates

Position of light sources relative to the camera?

Value Summary

Inherited	FALSE, or TRUE

Graphics Primitives

Objects	CameraCoordinates Default Values
`plot::DistantLight`, `plot::PointLight`, `plot::SpotLight`	FALSE

Description

With `CameraCoordinates = FALSE`, the coordinates defining the position of a light are interpreted as model coordinates in 3 space. Thus, the lights are positioned relative to the objects in the scene. They do not move when the camera moves.

With `CameraCoordinates = TRUE`, these coordinates are interpreted as "camera coordinates". Thus, the lights are attached to the camera and move automatically with the camera when it is moved.

A vector (*x*, *y*, *z*) in "camera coordinates" has to be interpreted as follows:

The *x*-coordinate refers to the horizontal axis of the picture that you see in the finder of the camera. Positive *x* values are to your right hand side, negative *x* values to your left hand side.

The *y*-coordinate refers to the vertical axis of the picture that you see in the finder. Positive *y* values are above you, negative *y* values below you.

The *z*-coordinate refers to the position along the optical axis of the camera. Positive *z* values refer to points in front of you, negative *z* values to points behind you.

In camera coordinates, the camera position is (0, 0, 0).

For example, a point light positioned at the point (0, 1, 0) in camera coordinates is a "head lamp" fixed at a distance of 1 above the camera.

A 3D plot may contain several cameras. Changing the state of `CameraCoordinates` for a light affects its relation to *all* cameras of a scene. With `CameraCoordinates` = `TRUE`, the position of the light in 3 space changes, when a new camera is chosen interactively.

Examples

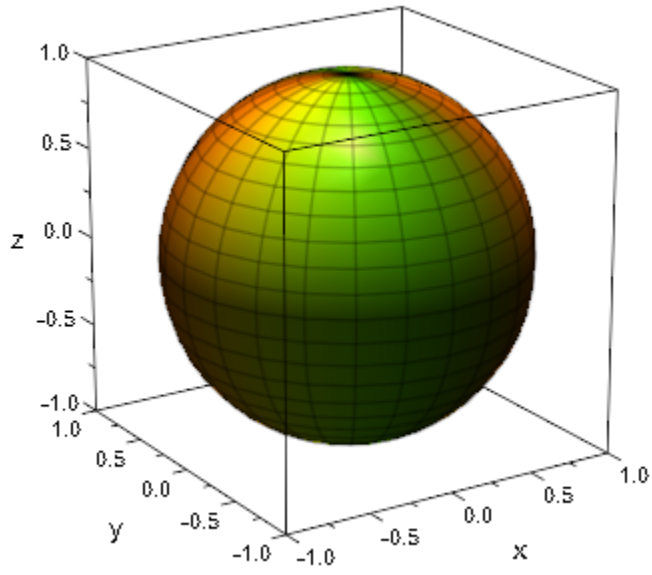
Example 1

We define a sphere:

```
sphere := plot::Spherical(
    [1, u, v], u = 0..2*PI, v = 0..PI,
    FillColorType = Functional,
    FillColorFunction =
        proc(u, v) begin
          [(2 + cos(2*u))/3, (2 + sin(2*u))/3, 0]
        end_proc):
```

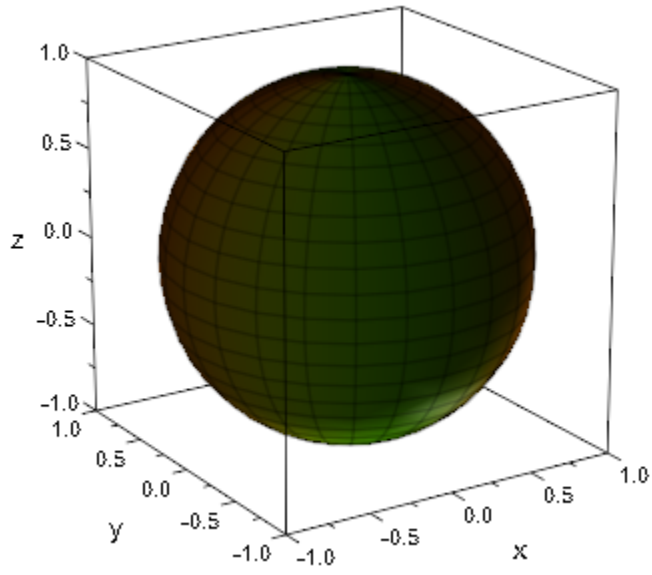
We define sunlight shining from above:

```
sunlight := plot::DistantLight([0, 0, 1], [0, 0, 0]):
plot(sphere, sunlight):
```



Next, the sunlight is defined relative to the camera. In camera coordinates, “above the camera” is given by the `Position[0, 1, 0]`. Because the camera points downwards, we set the direction of the sunlight “behind” the camera as well by choosing the direction `[0, 1, -1.5]` w.r.t. the camera:

```
sunlight := plot::DistantLight([0, 1, -1.5], [0, 0, 0],
                               CameraCoordinates = TRUE):
plot(sphere, sunlight):
```




```
delete sphere, sunlight, pointlight:
```

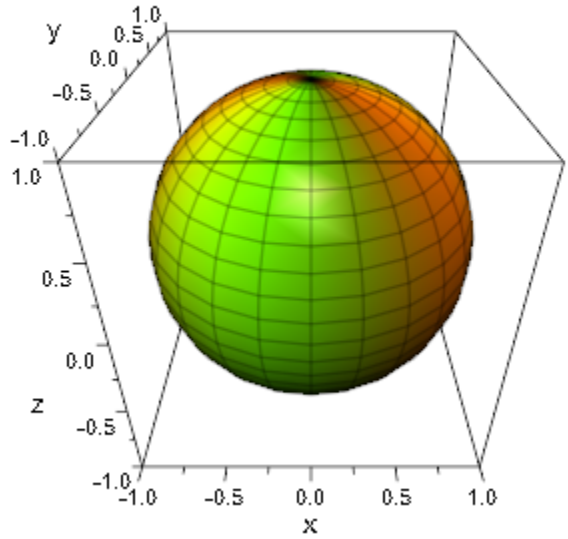
Example 2

We define the same sphere as in the previous example:

```
sphere := plot::Spherical(
    [1, u, v], u = 0..2*PI, v = 0..PI,
    FillColorType = Functional,
    FillColorFunction =
        proc(u, v) begin
          [(2 + cos(2*u))/3, (2 + sin(2*u))/3, 0]
        end_proc):
```

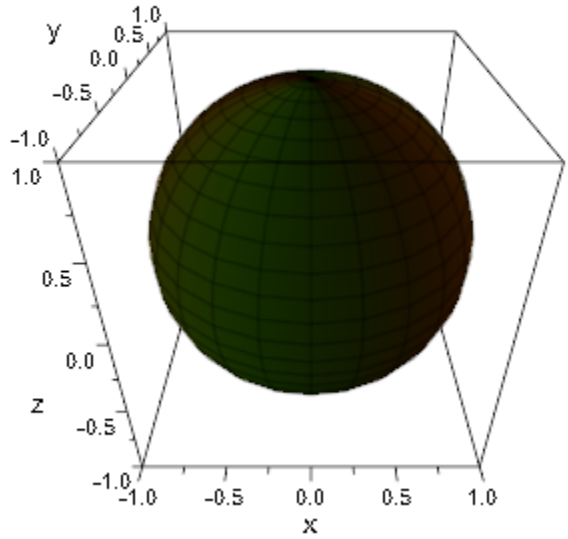
We define an animated camera. First, we use sunlight fixed in space:

```
camera := plot::Camera([-3*sin(a), -3*cos(a), 2],
                       [0, 0, 0], 0.3*PI, a = 0..2*PI):
sunlight:= plot::DistantLight([0, -2, 3], [0, 0, 0]):
plot(sphere, camera, sunlight):
```



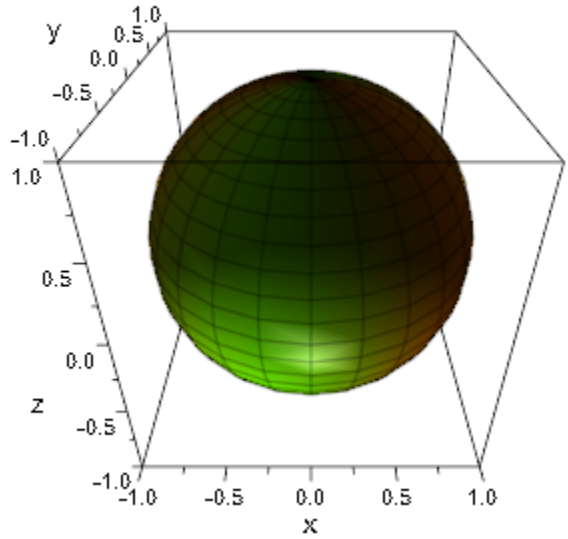
Next, we use sunlight moving with the camera:

```
sunlight:= plot::DistantLight([0, 3, -2], [0, 0, 0],
                              CameraCoordinates = TRUE):
plot(sphere, camera, sunlight):
```



We define a point light that is fixed to some point above the camera:

```
pointlight := plot::PointLight([0, 1, 0],
                               CameraCoordinates = TRUE):
plot(sphere, camera, pointlight):
```




```
delete sphere, camera, sunlight, pointlight:
```

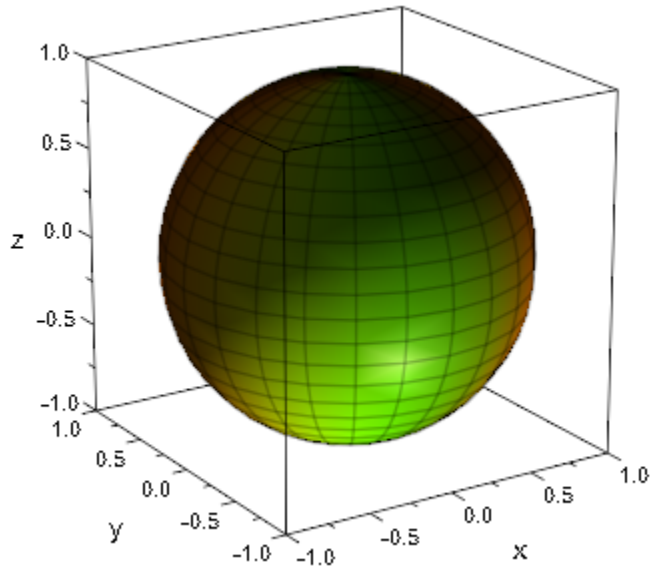
Example 3

We define the same sphere as in the previous examples:

```
sphere := plot::Spherical(
    [1, u, v], u = 0..2*PI, v = 0..PI,
    FillColorType = Functional,
    FillColorFunction =
        proc(u, v) begin
          [(2 + cos(2*u))/3, (2 + sin(2*u))/3, 0]
        end_proc):
```

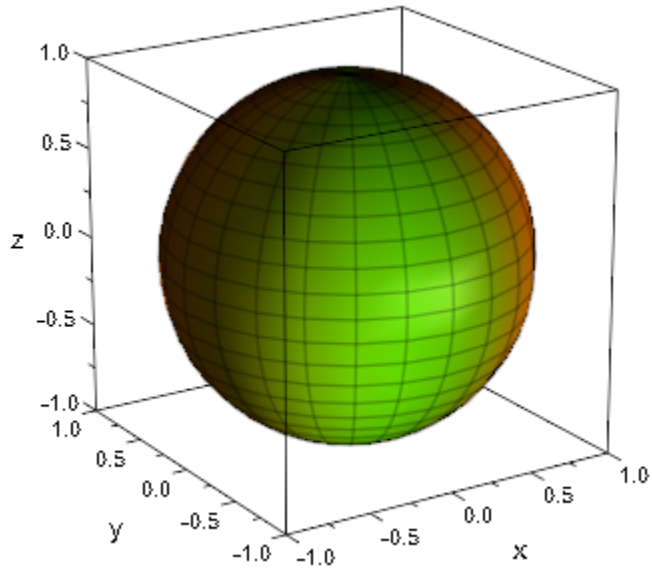
We define an animated point light that is positioned below the camera initially. It moves to some point above the camera during the animation:

```
pointlight := plot::PointLight([0, 10*a, 0], a = -1..1,
                               CameraCoordinates = TRUE):
plot(sphere, pointlight):
```



We define an animated point light that is positioned to the left of the camera initially. It moves to the right of the camera:

```
pointlight := plot::PointLight([10*a, 0, 0], a = -1 .. 1,
                               CameraCoordinates = TRUE):
plot(sphere, pointlight):
```




```
delete sphere, pointlight:
```

See Also

MuPAD Functions

`LightColor` | `LightIntensity` | `Position` | `SpotAngle` | `Target`

CameraDirection, CameraDirectionX, CameraDirectionY, CameraDirectionZ

Direction of the automatic camera

Value Summary

`CameraDirection`	Library wrapper for "[`CameraDirectionX`, `CameraDirectionY`, `CameraDirectionZ`]" (3D)	See below
`CameraDirectionX`, `CameraDirectionY`, `CameraDirectionZ`		See below

Graphics Primitives

Objects	Default Values
`plot::Scene3d`	

Description

`CameraDirection` controls the direction where the automatically set camera is positioned.

`CameraDirectionX` etc. refer to the single coordinates of this direction.

When creating a 3D scene, an "automatic camera" is used. It is placed somewhere along the ray starting at the center of the scene (or the center of an explicitly requested `ViewingBox`, respectively) with the direction given by `CameraDirection`.

The distance to the scene is chosen automatically such that the graphical scene or `ViewingBox` fills the drawing area optimally.

The `CameraDirection` value is a list or vector of numbers.

Note: This vector represents a direction, where the camera is found when starting at the center of the scene or viewing box. It is not the position of the camera!

The length of the `CameraDirection` does not matter, only its direction. The length should not be zero, though.

`CameraDirection` = `[0, 0, 1]` (looking straight down onto the *x*-*y* plane along the *z*-axis) does not yield a well defined camera view. This direction is automatically replaced by a direction that is close to, but not exactly equal to the *z* direction and orients the scene similiar to a 2D plot.

The `CameraDirection` attributes cannot be animated.

The automatic camera is designed to produce a picture of the entire scene or viewing box, filling the drawing area optimally. `CameraDirection` is the only means to control it.

If only parts of a scene shall be visible, or if the camera is not to aim at the center of the scene, or if large perspective distortions are desired, or if the camera position is to be animated, one has to define one's own camera of type `plot::Camera`. It can be placed at an arbitrary `Position` with an arbitrary `FocalPoint` and can have an arbitrary `ViewingAngle`. Further, it can be animated (allowing to realize a flight through a 3D scene).

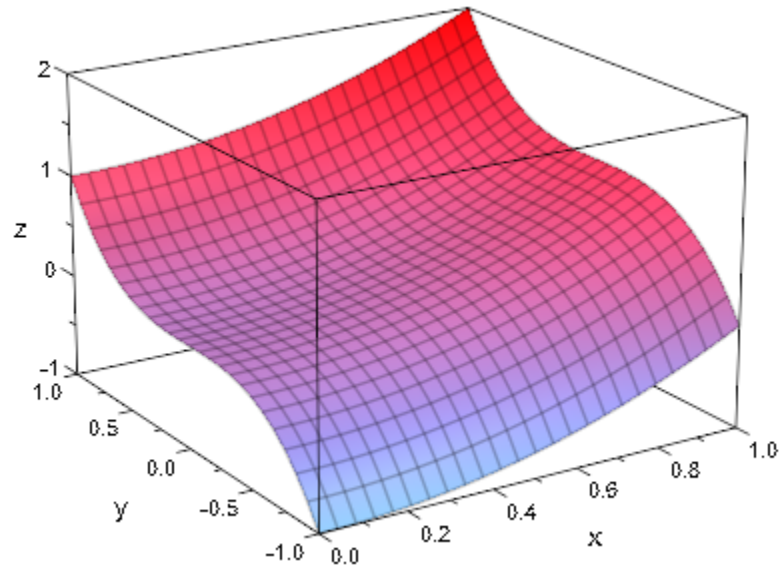
When such a camera object is inserted in a graphical scene, the automatic camera is switched off and the user defined camera takes over, automatically. It uses its own perspective parameters and ignores the attribute `CameraDirection`.

Examples

Example 1

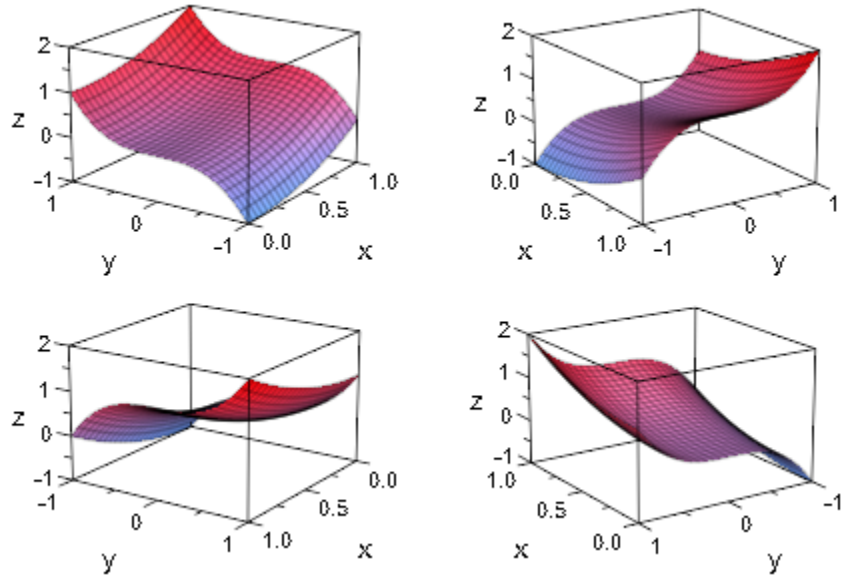
We look at a function with the default direction of the automatic camera:

```
f := plot::Function3d(x^2 + y^3, x = 0..1, y = -1 ..1):
plot(f):
```



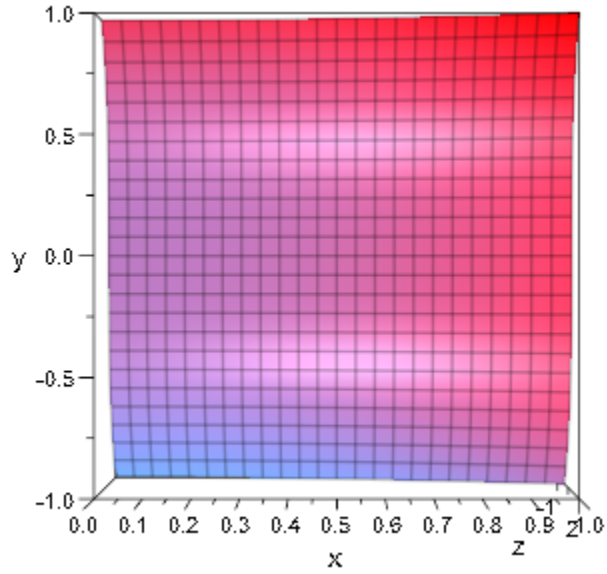
We look from different directions:

```
S1 := plot::Scene3d(f, CameraDirection = [-3,-4, 5]):
S2 := plot::Scene3d(f, CameraDirection = [ 3,-4, 5]):
S3 := plot::Scene3d(f, CameraDirection = [ 3, 4, 5]):
S4 := plot::Scene3d(f, CameraDirection = [-3, 4, 5]):
plot(S1, S2, S3, S4)
```



We look straight down onto the *x*-*y* plane along the *z*-axis:

```
plot(f, CameraDirection = [0, 0, 1])
```




```
delete f, S1, S2, S3:
```

See Also

MuPAD Functions

`FocalPoint` | `OrthogonalProjection` | `Position` | `ViewingAngle`

FocalPoint, FocalPointX, FocalPointY, FocalPointZ

Focal point of a camera

Value Summary

`FocalPoint`	Library wrapper for "[`FocalPointX`, `FocalPointY`, `FocalPointZ`]" (3D)	List of 2 or 3 expressions, depending on the dimension
`FocalPointX`, `FocalPointY`, `FocalPointZ`	Optional	MuPAD expression

Graphics Primitives

Objects	Default Values
`plot::Camera`	

Description

The attribute `FocalPoint` refers to the point a camera taking pictures of a 3D scene is aimed at. Its value is a list or vector of coordinates (numerical values or symbolic expressions of an animation parameter).

`FocalPointX` etc. refer to the single coordinates `x` etc.

The optical axis of the camera is given by the vector from the camera `Position` to its `FocalPoint`.

When creating a camera by

```
camera := plot::Camera(camera_position, focal_point, opening_angle),
```

the focal point is the second argument. Internally, this point is stored as the attribute `FocalPoint` and can be accessed and changed as `camera::FocalPoint`.

The focal point attributes can be animated.

Of course, the focal point should be set such that the camera points into the direction of the objects that are to be rendered. Typically, for a camera positioned outside the graphical scene, a good focal point is the center of the scene.

Examples

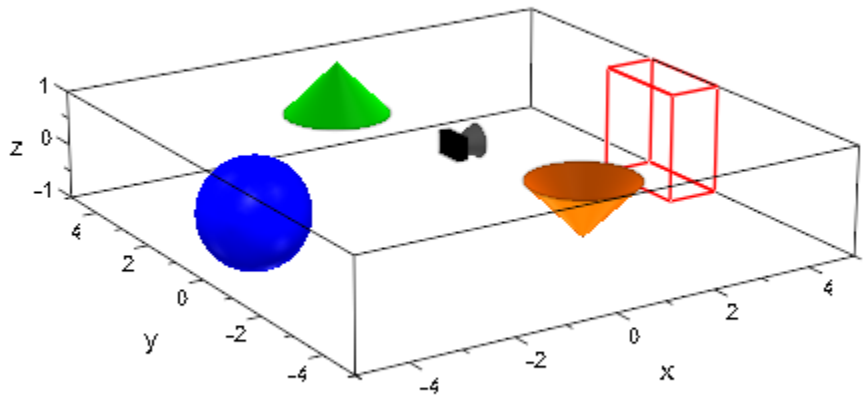
Example 1

We define a scene consisting of 4 geometric objects:

```
b := plot::Box(4..5, -1..1, -1..1, Filled = FALSE,
               LineColor = RGB::Red):
c1 := plot::Cone(1, [0, 4, 0], [0, 4, 1], Color = RGB::Green):
s := plot::Sphere(1, [-4, 0, 0], Color = RGB::Blue):
c2 := plot::Cone(1, [0, -4, 1], [0, -4, 0],
                 Color = RGB::Orange):
```

We use a small black object to mark the point (0, 0, 0.5), where we wish to place an animated camera:

```
cameraposition := plot::Group3d(
      plot::Box(-0.1..0.1, -0.3..0.3, 0.3..0.7,
                Color = RGB::Black),
      plot::Cone(0.1, [0, 0, 0.5], 0.3, [0.5, 0, 0.5],
                 Color = RGB::DarkGrey)):
plot(b, c1, s, c2, cameraposition)
```



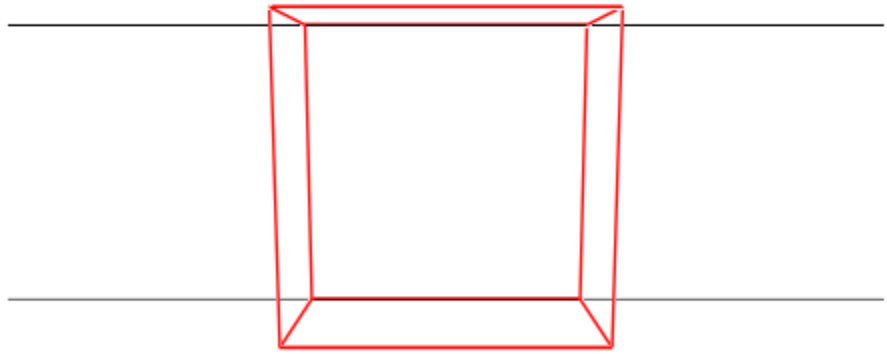
When defining the camera, the second argument is the `FocalPoint`. In this case, it is animated: The camera is to turn around the *z*-axis.

```
camera := plot::Camera([0, 0, 0.5], [4*cos(a), 4*sin(a), 0],
                       PI/4, a = 0..2*PI):
camera::FocalPoint
```

$$[4\cos(a),\ 4\sin(a),\ 0]$$

We insert the animated camera:

```
plot(b, c1, s, c2, camera)
```

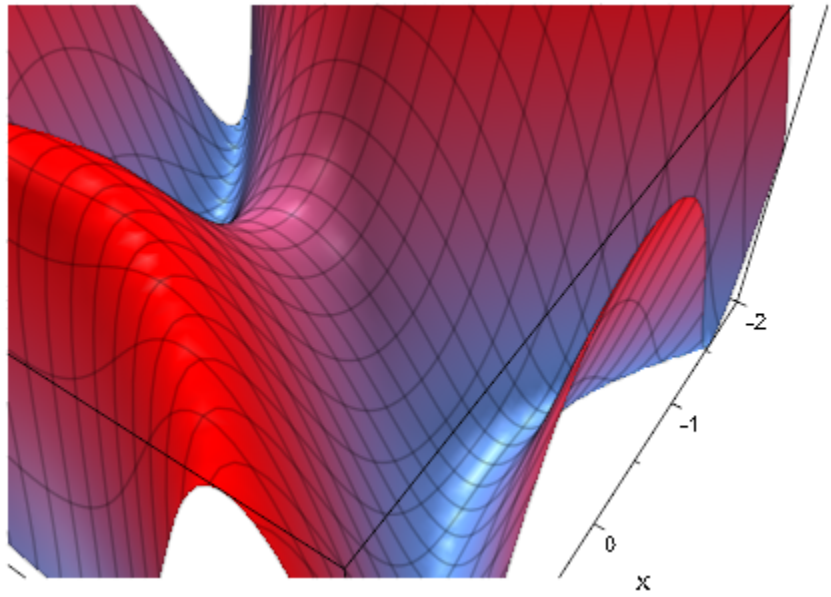


```
delete b, c1, s, c2, cameraposition, camera:
```

Example 2

We define a function and a camera with an animated focal point:

```
f := plot::Function3d(sin(x^2-y^2), x = -2..2, y = -2..2,
                      Submesh = [2, 2]):
camera := plot::Camera([3, 3, 3], [sin(a), cos(a), 0],
                       PI/6, a = 0..2*PI):
plot(f, camera)
```




```
delete f, camera:
```

See Also

MuPAD Functions

`CameraDirection` | `OrthogonalProjection` | `Position` | `ViewingAngle`

LightColor

Color of light

Value Summary

Inherited	Color

Description

`LightColor` sets the color of user-defined light sources such as `plot::AmbientLight`, `plot::DistantLight` etc.

The value of `LightColor` must be an `RGB` or `RGBa` color, i.e., a list of three or four numerical values between 0 and 1. (The fourth value is the opacity entry of an `RGBa` color. It is accepted but does not have any effect on the light color.)

The `RGB` library provides many predefined colors such as `RGB::Blue` etc. See Section of this document for more information on colors.

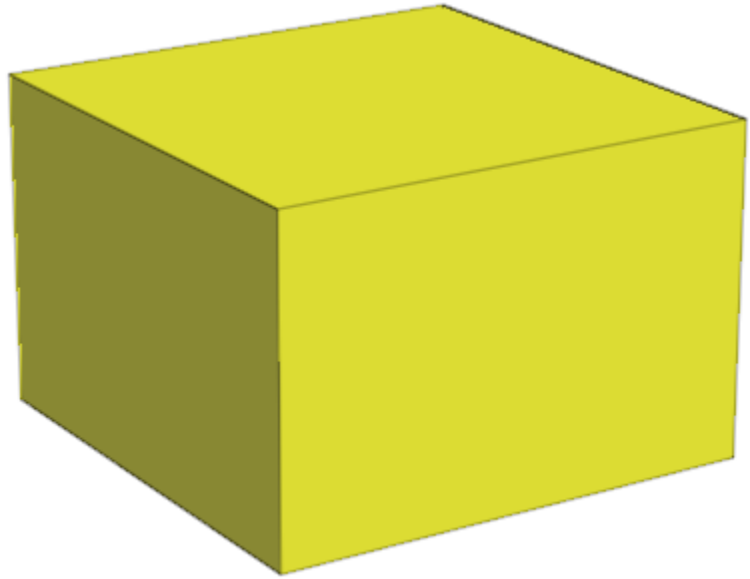
`LightColor` cannot be animated.

Examples

Example 1

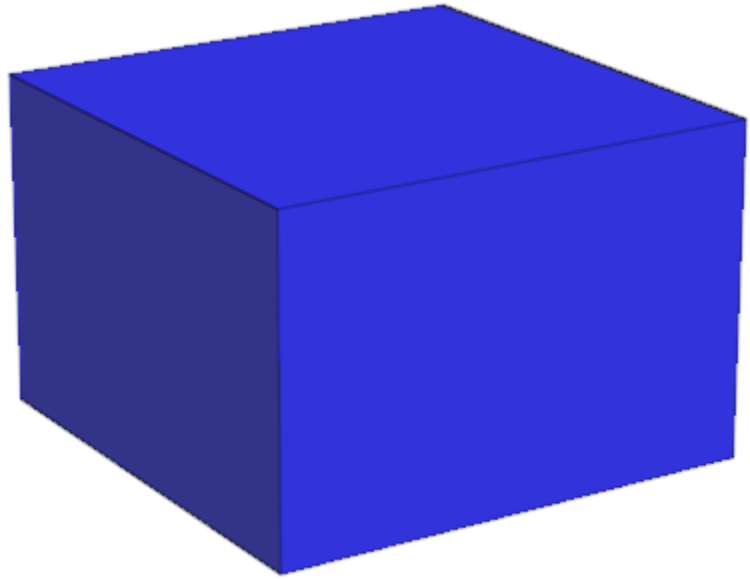
We define a white box and illuminate it by a yellow distant light:

```
b := plot::Box(-1..1, -1..1, -1..1, Color = RGB::White):
light := plot::DistantLight([-1, -2, 3], [0, 0, 0],
                            Color = RGB::Yellow):
plot(b, light, Axes = None)
```



We change the color of the light source:

```
light::LightColor := RGB::Blue:
plot(b, light, Axes = None)
```



```
delete b, light:
```

See Also

MuPAD Functions

CameraCoordinates | LightIntensity | Position | SpotAngle | Target

Lighting

Light schemes for 3D graphics

Value Summary

Inherited	`Automatic`, `Explicit`, or `None`

Graphics Primitives

Objects	Lighting Default Values
`plot::Scene3d`	`Automatic`

Description

`Lighting` determines the light scheme used to illuminate a 3D scene.

With the default `Lighting = Automatic`, several light sources are set automatically to illuminate a 3D scene. Firstly, there is ambient white light of type `plot::AmbientLight`:

- Light 0: `LightIntensity = 0.25, LightColor = RGB::White`

In addition, there are 6 directed lights of type `plot::DistantLight` with `LightColor = RGB::White`. Their directions is given as follows: Think of the graphical scene as being scaled to a standard box extending from - 1 to 1 in each coordinate direction. In these scaled coordinates, the directed lights shine into the directions given by the following `Target` attributes:

- Light 1: `Target = [-5, -6, -8]`, `LightIntensity = 0.50`,
- Light 2: `Target = [ 5,  6,  8]`, `LightIntensity = 0.60`,
- Light 3: `Target = [ 5, -6, -8]`, `LightIntensity = 0.20`,
- Light 4: `Target = [-5,  6,  8]`, `LightIntensity = 0.25`,
- Light 5: `Target = [-5,  6, -8]`, `LightIntensity = 0.20`,

- Light 6: `Target = [ 5, -6,  8]`, `LightIntensity = 0.25`.

User-defined lights in the scene are ignored with `Lighting = Automatic`.

With `Lighting = Explicit`, the light sources set via `Lighting = Automatic` are switched off and user-defined light sources are switched on.

A `plot` command searches for light objects set by the user. If `Lighting` is not specified and any kind of user-defined light object is found in the scene, `Lighting = Explicit` is set automatically.

Switching between `Lighting = Automatic` and `Lighting = Explicit` in the inspector, one can easily compare the effect of the automatic lights with the effect of ones own lights.

With `Lighting = None`, the 3D shading algorithm based on reflections of light emitted from light sources in the graphical scene is switched off. This does not mean that the graphical scene turns black: all objects are painted in the color they are defined with. However, the scene will appear flat, because the depth of a 3D scene is created via the shading caused by different reflections of light at different points of the scene.

The maximal number of lights that can be used to illuminate a 3D scene depends on the OpenGL driver installed on the computer.

Note: Some OpenGL drivers do not allow more than 6 light sources. If there are more light sources in the scene, the surplus lights are ignored. Lights that are switched off via `Visible = FALSE` are not counted.

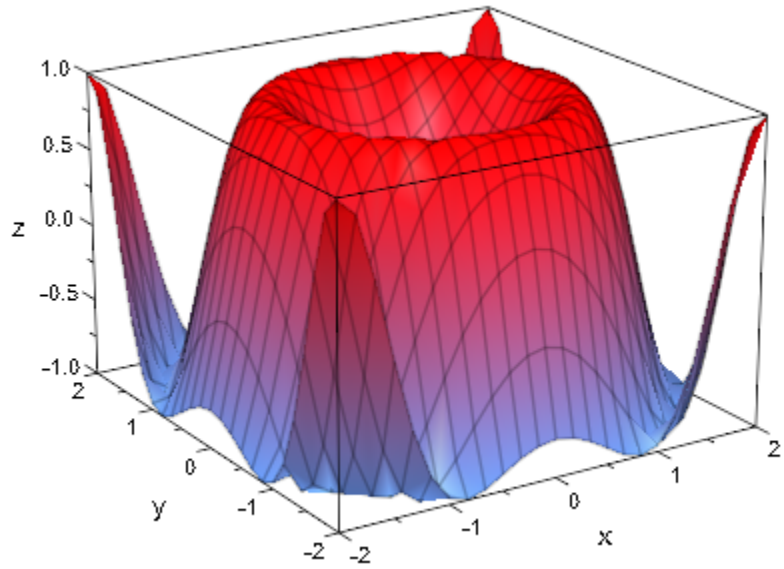
After activating a 3D plot (by clicking it), the “Help” menu contains an item “OpenGL Info” that provides information about the maximal number of lights.

Examples

Example 1

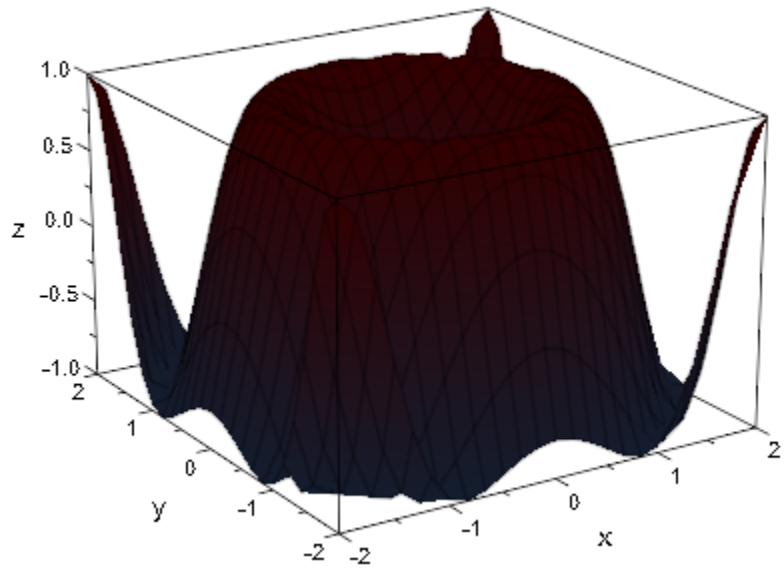
In our first scene, no lights are specified. The default setting `Lighting = Automatic` is used:

```
f := plot::Function3d(sin(x^2 + y^2), x = -2..2, y = -2..2):
plot(f):
```



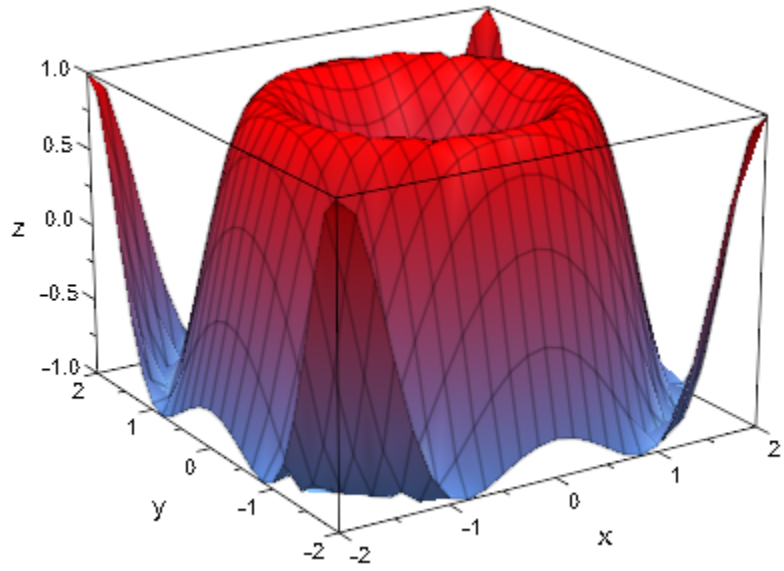
When specifying `Lighting = Explicit`, the lights set by the user are used. Since the scene does not contain any lights, the scene turns dark:

```
plot(f, Lighting = Explicit):
```



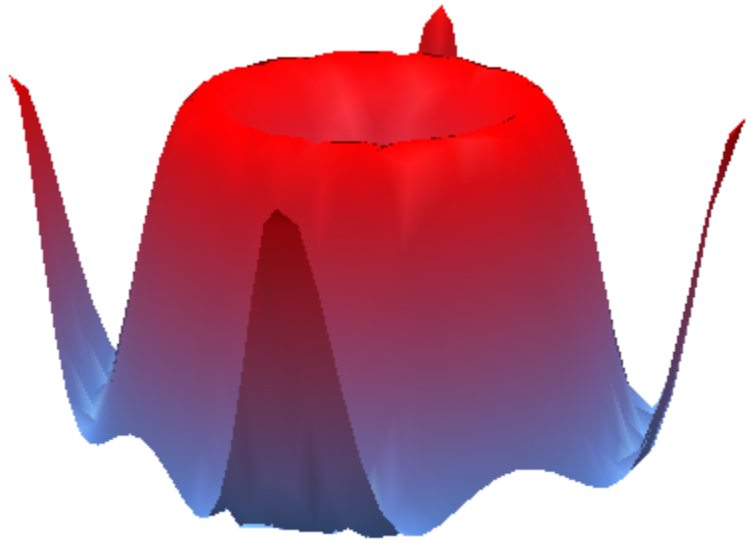
Lights are specified in the next scene. The setting `Lighting = Explicit` is used automatically:

```
Light0 := plot::AmbientLight(0.25):
Light1 := plot::DistantLight([ 1,  0, 1], [0, 0, 0], 0.3):
Light2 := plot::DistantLight([-1,  0, 1], [0, 0, 0], 0.3):
Light3 := plot::DistantLight([ 0,  1, 1], [0, 0, 0], 0.3):
Light4 := plot::DistantLight([ 0, -1, 1], [0, 0, 0], 0.3):
plot(f, Light0, Light1, Light2, Light3, Light4):
```



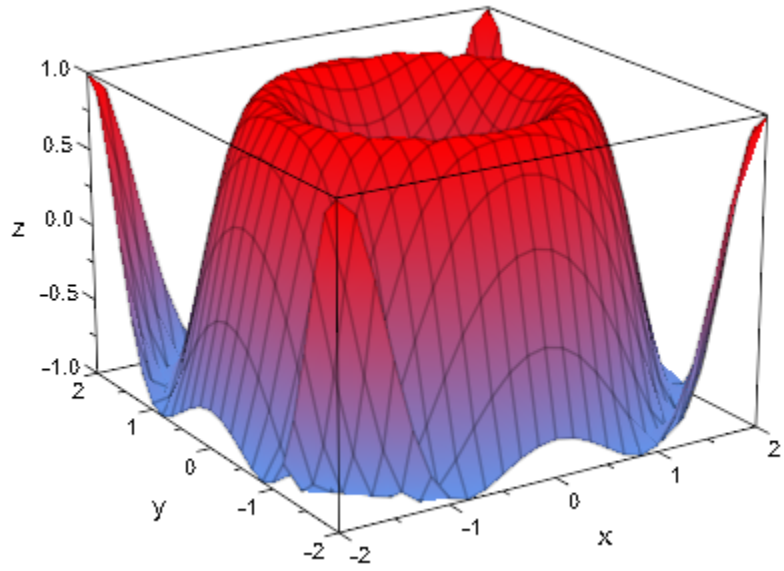
We switch off the parameter lines:

```
plot(f, Light0, Light1, Light2, Light3, Light4,
     XLinesVisible = FALSE, YLinesVisible = FALSE,
     Axes = None):
```



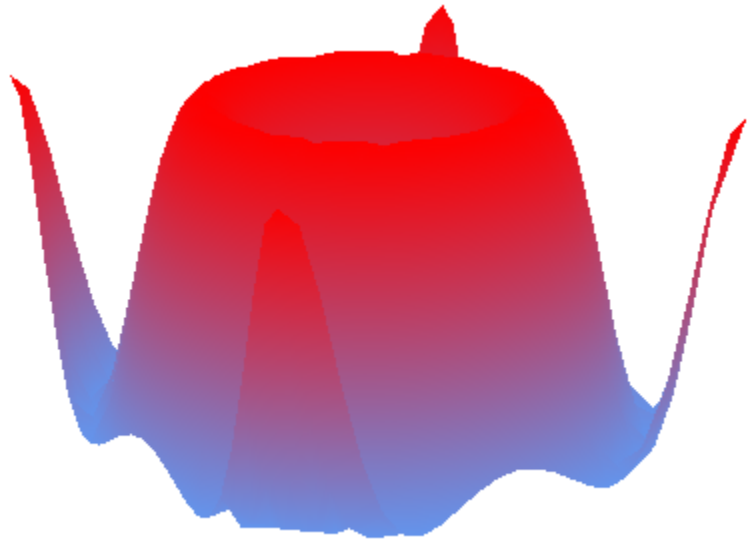
In the next scene, the 3D shading model is switched off via `Lighting = None`:

```
plot(f, Lighting = None):
```



In the previous picture, the axes box and the mesh lines are switched on and create a certain 3D effect. After switching the box and the mesh lines off, the scene appears flat when rendered without shading:

```
plot(f, Lighting = None, Axes = None,
     XLinesVisible = FALSE, YLinesVisible = FALSE):
```



```
delete f, Light0, Light1, Light2, Light3, Light4:
```

See Also

MuPAD Functions
LightColor | LightIntensity | SpotAngle

MuPAD Graphical Primitives
plot::AmbientLight | plot::DistantLight | plot::PointLight | plot::SpotLight

LightIntensity

Intensity of light

Value Summary

Optional	Arithmetical expression between 0 and 1

Description

`LightIntensity` governs the intensity of user defined light sources such as `plot::AmbientLight`, `plot::DistantLight` etc.

The intensity of all user defined light source can be set by `Intensity = intensity`. The value `intensity` must be a number between 0 and 1. Values smaller than 0 or larger than 1 are accepted and handled like 0 or 1, respectively.

This attribute can be animated.

Undirected ambient light of intensity 1 dominates all other light sources.

Examples

Example 1

When generating a light source of type `plot::DistantLight`, the third argument is the light intensity. Internally, this value is stored as the attribute `LightIntensity` and can be accessed and changed as the corresponding slot of the light object:

```
light := plot::DistantLight([2, -1, 3], [0, 0, 0], 0.5):
light::LightIntensity
```

```
    0.5
```

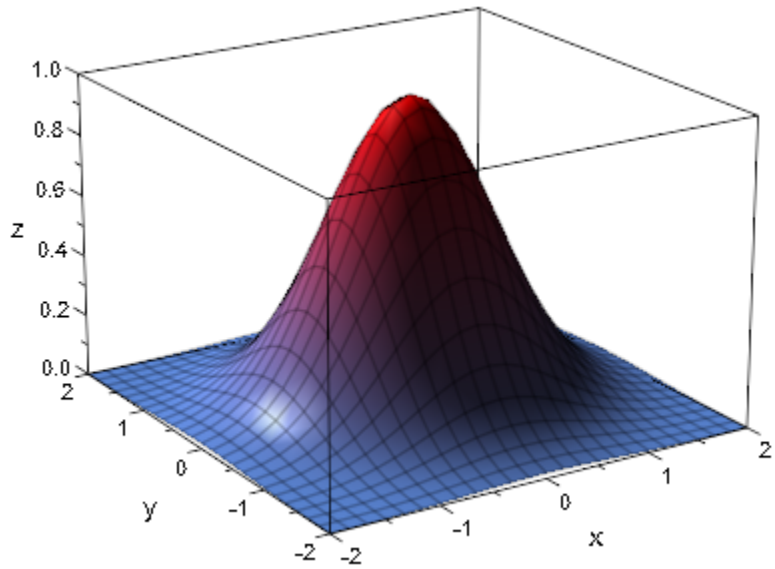
```
light::LightIntensity:= 0.4:
```

```
light::LightIntensity
```

```
0.4
```

We illuminate a function plot by two distant lights with animated intensities:

```
plot(plot::Function3d(exp(-x^2 - y^2), x = -2..2, y = -2..2),
     plot::DistantLight([5, -1, 3], [0, 0, 0], 1 - a, a = 0..1),
     plot::DistantLight([-3, 5, 2], [0, 0, 0], a, a = 0..1)):
```




```
delete light:
```

See Also

MuPAD Functions

`CameraCoordinates` | `LightColor` | `Position` | `SpotAngle` | `Target`

OrthogonalProjection

Parallel projection without perspective distortion

Value Summary

Inherited	FALSE, or TRUE

Graphics Primitives

Objects	OrthogonalProjection Default Values
`plot::Camera`	FALSE

Description

Setting `OrthogonalProjection` = `TRUE`, a camera uses parallel projection without perspective distortion.

By default, a camera uses `OrthogonalProjection` = `FALSE`. Depending on the distance of the camera to the graphical scene (set by the attribute `Position`), the scene is subject to some natural perspective distortion. The distortion is large when the camera is near the scene. It is small when the camera is far away.

In principle, using parallel projection is equivalent to placing a camera at a very large distance from the scene, looking through a very powerful tele lense.

For technical reasons, however, you should *not* suppress perspective distortion by placing the camera yourself somewhere far away via the attribute `Position` and turning the camera's lens into a tele lense by setting a small value for its opening angle (cf. `ViewingAngle`). This may lead to problems with the hidden line algorithm used by the 3D renderer. Further, a suitable opening angle has to be found experimentally such that the scene fills a reasonable portion of the drawing area.

Use `OrthogonalProjection` = `TRUE` instead. The scaling of the scene is done automatically to fill the drawing area optimally.

With `OrthogonalProjection` = `TRUE`, the view is only determined by the direction of the vector from the `FocalPoint` of the camera to its `Position`. (The camera is moved to infinity along the ray given by this “optical axis”, using an infinitesimal opening angle.)

The absolute camera position in 3-space as well as its opening angle are ignored.

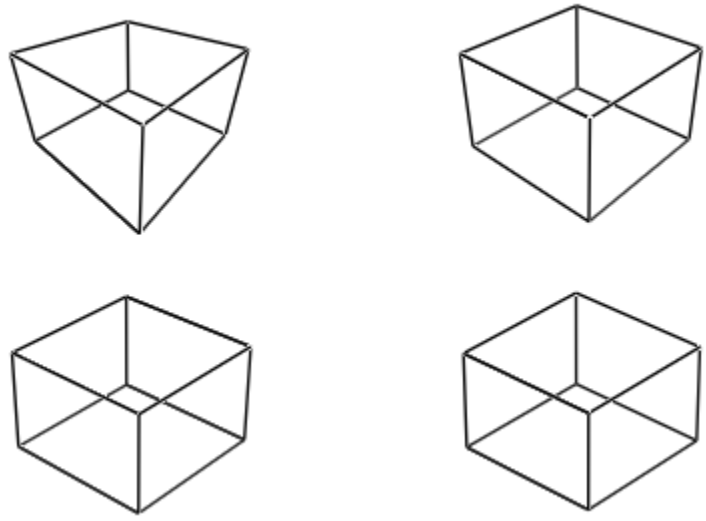
Examples

Example 1

We look at a box with side length 2 using cameras at different positions. We double the distance between the camera and the center of the box from one scene to the next. At the same time, we use more and more powerful tele lenses by decreasing the camera's opening angle by a factor of $\frac{1}{2}$, so that the box has approximately the same size.

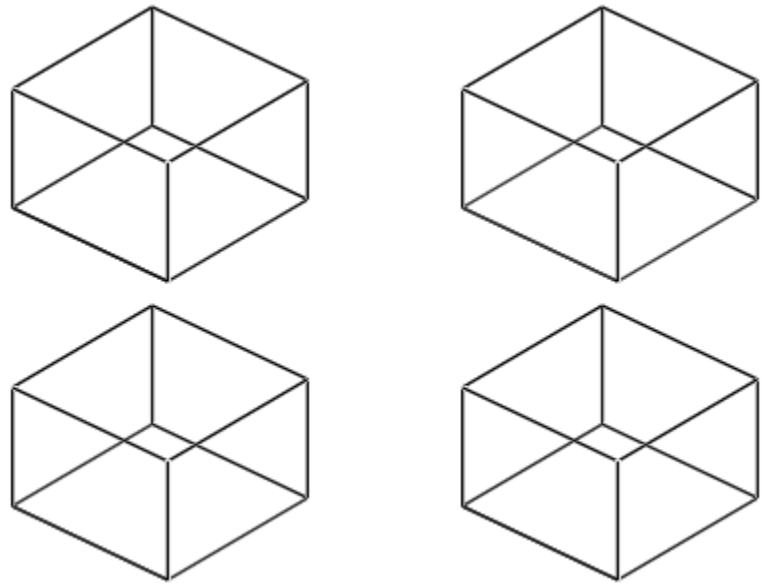
- In `S1`, the camera is close to the box. The box is distorted heavily.
- In `S2`, the camera is farther away. The perspective distortions are smaller.
- In `S3`, the distance of the camera to the box center is about 5 times the diameter of the box. Only minor perspective distortions are visible.
- In `S4`, the distance of the camera is about 10 times the diameter of the box. The perspective distortions are almost gone:

```
b := plot::Box(-1..1, -1..1, -1..1, Filled = FALSE,
               LineColor = RGB::Black):
S1:= plot::Scene3d(b, plot::Camera([ 2,  1.8,  2.5], [0, 0, 0], PI/3)):
S2:= plot::Scene3d(b, plot::Camera([ 4,  3.6,  5.0], [0, 0, 0], PI/6)):
S3:= plot::Scene3d(b, plot::Camera([ 8,  7.2, 10.0], [0, 0, 0], PI/12)):
S4:= plot::Scene3d(b, plot::Camera([16, 14.4, 20.0], [0, 0, 0], PI/24)):
plot(S1, S2, S3, S4, Axes = None)
```



We suppress the distortions completely by setting `OrthogonalProjection` = `TRUE`. Note the automatic scaling of the scene:

```
plot(S1, S2, S3, S4, Axes = None, OrthogonalProjection = TRUE)
```

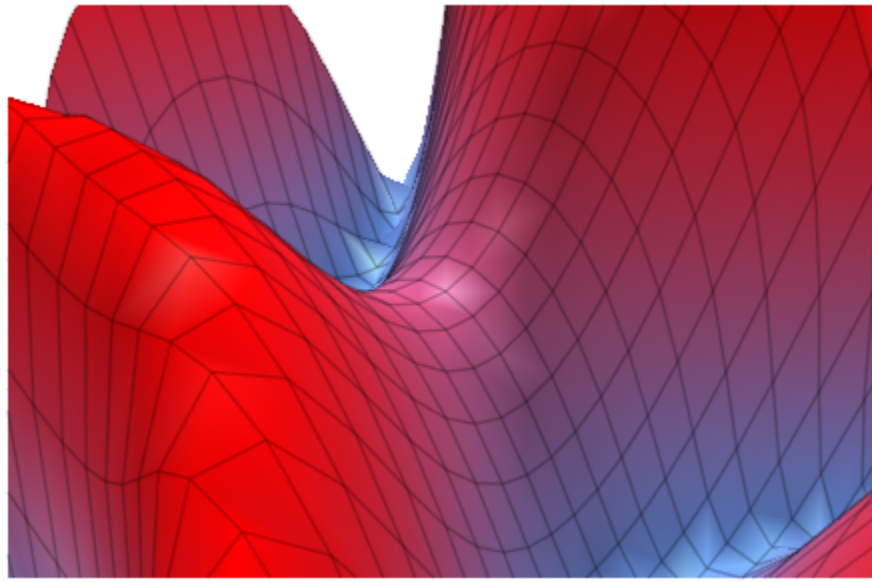


```
delete b, S1, S2, S3, S4:
```

Example 2

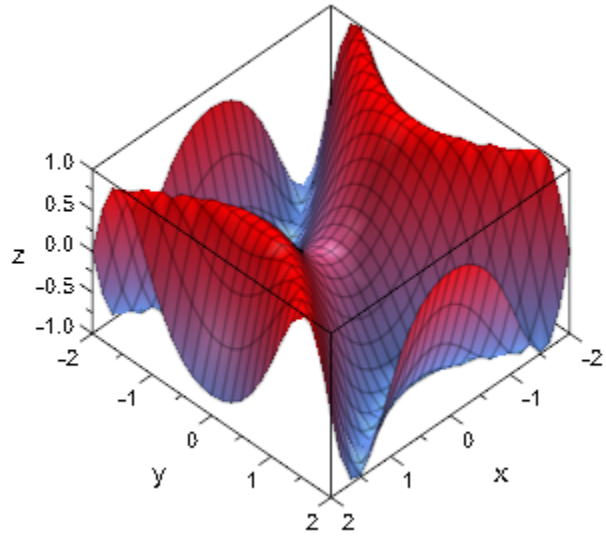
The following camera is too close to the scene to make all parts of the function graph visible:

```
f := plot::Function3d(sin(x^2 - y^2), x = -2..2, y = -2..2):
camera := plot::Camera([2, 2, 2], [0, 0, 0], PI/5):
plot(f, camera)
```



With `OrthogonalProjection` = `TRUE`, the specified position and opening angle are ignored. The effect of `OrthogonalProjection` is the same as placing the camera far away and choosing a tiny opening angle such that the scene fills the drawing area optimally:

```
camera::OrthogonalProjection := TRUE:
plot(f,camera)
```




```
delete f, camera:
```

See Also

MuPAD Functions

`CameraDirection` | `FocalPoint` | `Position` | `ViewingAngle`

SpotAngle

Opening angle of the light cone of a spot light

Value Summary

Mandatory	Real-valued expression (interpreted in radians)

Graphics Primitives

Objects	SpotAngle Default Values
`plot::SpotLight`	

Description

`SpotAngle` sets the opening angle of the light cone of a spot light in radians, and defines the opening angle of the light cone emitted by spot lights of type `plot::SpotLight`.

The values for `SpotAngle` have to be given in radians. Reasonable value lie between 0 and π.

`SpotAngle` can be animated.

Examples

Example 1

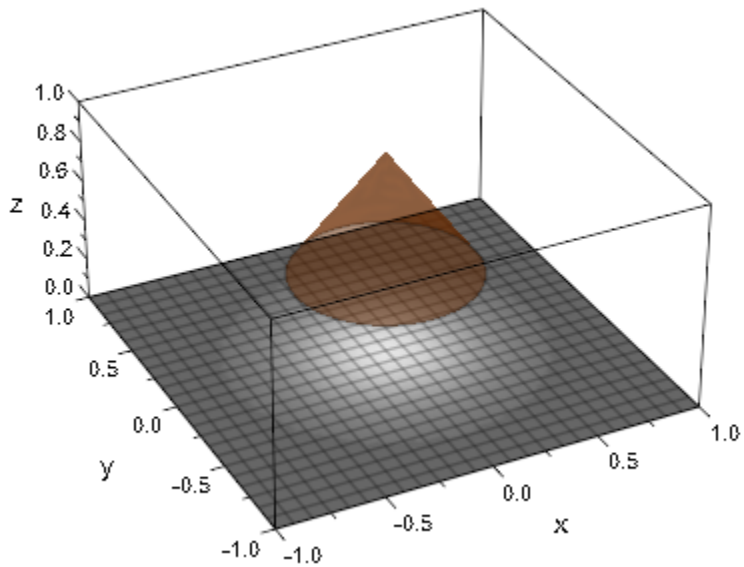
When creating a spot light, the third argument is the `SpotAngle`:

```
spotlight := plot::SpotLight([0, 0, 1], [0, 0, 0], a*PI, 1,
                             a = 0..0.4, LightColor = RGB::White):
spotlight::SpotAngle
```

$\pi\, a$

We illuminate the *x*-*y* plane by the animated spot light and some ambient light. The spot light is visualized by a cone:

```
ambientlight := plot::AmbientLight(0.2):
s := plot::Surface([x, y, 0], x = -1..1, y = -1..1,
                   Submesh = [2, 2], Color = RGB::White,
                   FillColorType = Flat):
c := plot::Cone(0, [0, 0, 1],
                0.6*tan(a*PI/2), [0, 0, 1 - 0.6],
                a = 0..0.4, Color = RGB::Orange.[0.5]):
plot(s, c, spotlight, ambientlight,
     CameraDirection = [-9, -18, 12])
```




```
delete spotlight, ambientlight, s, c:
```

See Also

MuPAD Functions

`CameraCoordinates` | `LightColor` | `LightIntensity` | `Position` | `Target`

Target, TargetX, TargetY, TargetZ

Target point of a light

Value Summary

`Target`	Library wrapper for "[`TargetX`, `TargetY`, `TargetZ`]" (3D)	List of 2 or 3 expressions, depending on the dimension
`TargetX`, `TargetY`, `TargetZ`	Mandatory	MuPAD expression

Graphics Primitives

Objects	Default Values
`plot::DistantLight`, `plot::SpotLight`	

Description

The `Target` attribute refers to the point a spot light is aimed at. It also controls the direction of a distant light which is given by the vector `Target - Position`.

`Target` sets the position of the point lights of type `plot::DistantLight` and `plot::SpotLight` are aimed at. `TargetX` etc. refer to the single coordinates of this point.

The value of `Target` is a list or vector of coordinates. `TargetX = x` etc. refer to the single coordinates of this list.

These attributes can be animated.

By default, the positions and the targets of light objects are given in model coordinates that have nothing to do with the camera that is used to view the scene.

When using the attribute `CameraCoordinates` = `TRUE`, the light source is fixed to the camera. It moves automatically, when the camera is moved.

Examples

Example 1

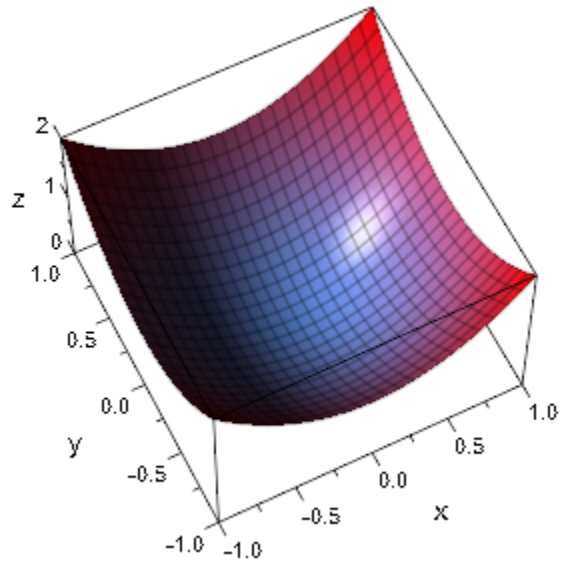
When generating lights of type `plot::DistantLight` and `plot::SpotLight`, the second argument ist the `Target`. In the following expample, it is animated:

```
sunlight := plot::DistantLight([0, 0, 2],
                               [cos(a), sin(a), 1],
                               a = 0..2*PI):
spotlight := plot::SpotLight([0, 0, 1],
                             [cos(a), sin(a), 1/2],
                             PI/5, a = 0..2*PI):
sunlight::Target, spotlight::Target
```

$$[\cos(a), \sin(a), 1], \left[\cos(a), \sin(a), \frac{1}{2}\right]$$

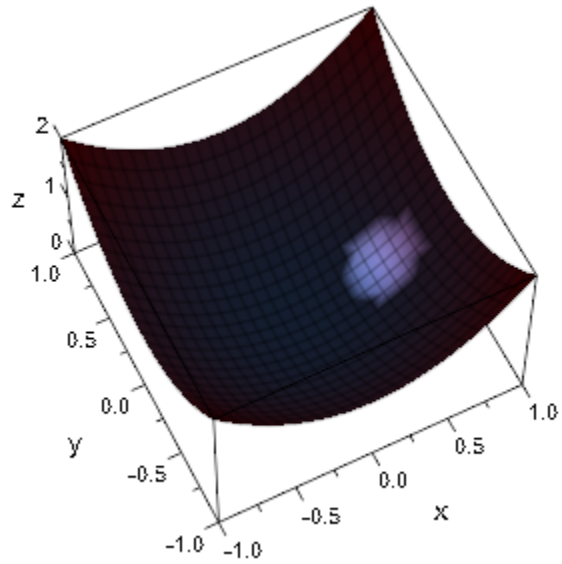
We illuminate a paraboloid with sunlight. Its direction is animated by the `Target` attribute:

```
f := plot::Function3d(x^2 + y^2, x = -1..1, y = -1..1):
plot(f, sunlight, CameraDirection = [-1, -2, 6])
```



We use the animated spot light:

```
plot(f, spotlight, CameraDirection = [-1, -2, 6])
```




```
delete sunlight, spotlight, f:
```

See Also

MuPAD Functions

`CameraCoordinates` | `LightIntensity` | `Position` | `SpotAngle`

UpVector, UpVectorX, UpVectorY, UpVectorZ, KeepUpVector

Up direction of a camera

Value Summary

`UpVector`	Library wrapper for "[`UpVectorX`, `UpVectorY`, `UpVectorZ`]" (3D)	List of 2 or 3 expressions, depending on the dimension
`UpVectorX`, `UpVectorY`, `UpVectorZ`	Optional	MuPAD expression
`KeepUpVector`	Inherited	`FALSE`, or `TRUE`

Graphics Primitives

Objects	Default Values

`plot::Camera`	`UpVector`: [`0.0`, `0.0`, `1.0`]
	`UpVectorX`, `UpVectorY`: `0.0`
	`UpVectorZ`: `1.0`
	`KeepUpVector`: `TRUE`

Description

`UpVector = [x, y, z]` sets the 3D vector that corresponds to the vertical direction of the 2D picture taken by the camera.

`UpVectorX` etc. denote the coordinates of the `UpVector`.

`KeepUpVector = TRUE` keeps the `UpVector` constant when the camera is moved. With `KeepUpVector = FALSE`, the `UpVector` is kept orthogonal to the optical axis when the camera is moved.

The picture taken by a camera is defined by the attributes `Position` (the 3D position of the camera) and `FocalPoint` (the 3D point the camera is pointed at). The vector from the position to the focal point is the optical axis of the camera.

As an additional degree of freedom, the camera may be rotated around the optical axis. This rotation is defined by specifying a 3D vector `UpVector`. In the final 2D picture taken by the camera, this vector is parallel to the vertical axis, pointing upwards.

With the default value `UpVector = [0, 0, 1]` the z-axis in 3D points upwards in the 2D picture.

The `UpVector` of a camera must not be zero and must not be parallel to the optical axis.

The default values are `UpVector = [0, 0, 1]` and `KeepUpVector = TRUE`.

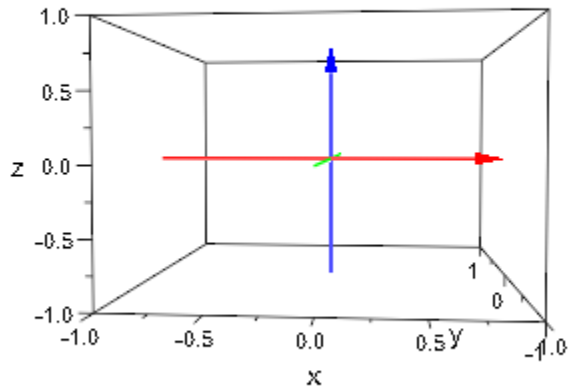
The restriction that the `UpVector` must not be parallel to the optical axis leads to discontinuities when the camera moves in such a way that this restriction is violated. In such a case, `KeepUpVector = FALSE` should be used. Cf. “Example 2” on page 24-1869.

Examples

Example 1

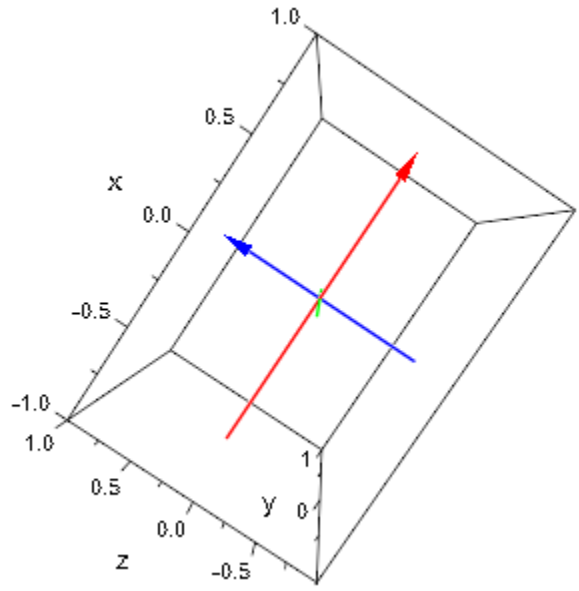
We view a cross of three arrows by a camera with the usual `UpVector` pointing into z-direction:

```
camera := plot::Camera([0.3, -4.0, 0.2], [0, 0, 0], PI/4,
                       UpVector = [0, 0, 1]):
plot(plot::Arrow3d([-1, 0, 0], [1, 0, 0], Color = RGB::Red),
     plot::Arrow3d([0, -1, 0], [0, 1, 0], Color = RGB::Green),
     plot::Arrow3d([0, 0, -1], [0, 0, 1], Color = RGB::Blue),
     camera)
```



We redefine the `UpVector` of the camera to point into the direction `[1, 0, 1]`. Now, this 3D direction becomes the vertical direction of the 2D picture:

```
camera::UpVector := [1, 0, 1]:
plot(plot::Arrow3d([-1, 0, 0], [1, 0, 0], Color = RGB::Red),
     plot::Arrow3d([0, -1, 0], [0, 1, 0], Color = RGB::Green),
     plot::Arrow3d([0, 0, -1], [0, 0, 1], Color = RGB::Blue),
     camera)
```



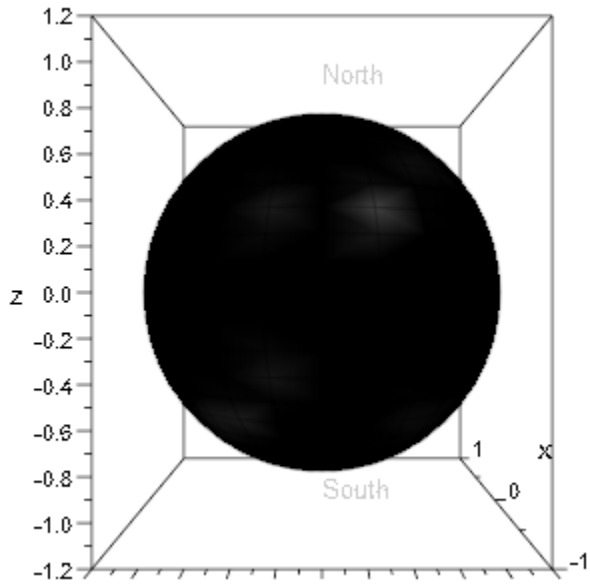

```
delete camera:
```

Example 2

We use an animated camera to fly over the north pole of a planet using the default `UpVector = [0, 0, 1]`. With `KeepUpVector = TRUE`, we encounter a discontinuity when the camera is positioned exactly over the north poly pointing straight down. The `UpVector` is parallel to the optical axis at this point:

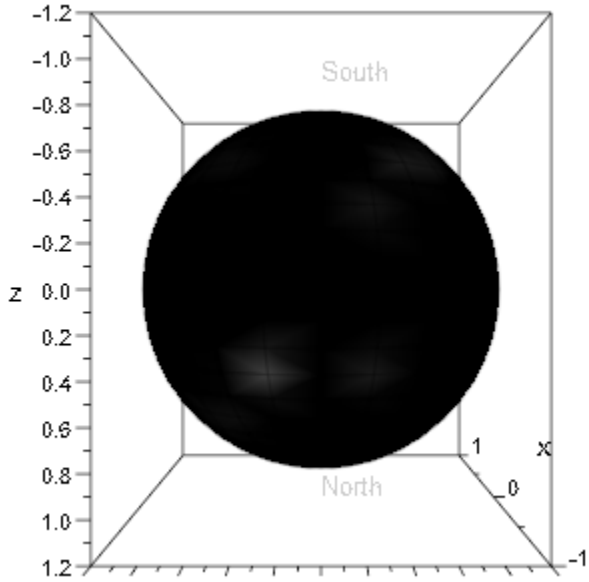
```
camera := plot::Camera([4*cos(a), 0, 4*sin(a)], [0, 0, 0], PI/4,
                       a = 0..PI, Frames = 300,
                       UpVector = [0, 0, 1],
                       KeepUpVector = TRUE):
planet := plot::Surface([cos(u)*sin(v), sin(u)*sin(v), cos(v)],
    u = 0..2*PI, v = 0..PI,
    FillColorFunction = proc(u, v)
             begin
               [cos(u)*cos(v)^2, cos(u)*cos(v)^2, cos(u)]
             end_proc):
font := ["sans-serif", 10, RGB::Grey80]:
```

```
text1 := plot::Text3d("North", [0, 0, 1.2], TextFont = font):
text2 := plot::Text3d("South", [0, 0, -1.2], TextFont = font):
plot(camera, planet, text1, text2, Scaling = Constrained);
```



With `KeepUpVector = FALSE`, no such discontinuity is encountered. However, when reaching the equator on the dark side of the planet, the `UpVector` has turned around: the upper side of the picture now is south, the lower side is north:

```
camera::KeepUpVector := FALSE:
plot(camera, planet, text1, text2, Scaling = Constrained);
```




```
delete camera, planet, font, text1, text2:
```

See Also

MuPAD Functions
`FocalPoint` | `Position`

ViewingAngle

Opening angle of the camera lense

Value Summary

Mandatory	MuPAD expression

Graphics Primitives

Objects	ViewingAngle Default Values
`plot::Camera`	

Description

`ViewingAngle` defines the viewing angle of a camera. It is also known as the “opening angle” of the camera's lense and is determined by its focal length.

Small viewing angles correspond to a tele lense, large opening angles to a wide angle lense. Angles close to π correspond to an (extreme) fish eye lense.

The values for `ViewingAngle` have to be given in radians. The angles should be larger than $\frac{1}{100}$ and smaller than π. Other values are replaced by some small positive angle or by an angle slightly less than π, respectively.

Note that when using a wide angle lense, the scene may fill only a part of the drawing area. With a tele lense, only parts of the scene may be visible.

When using a camera object with a given `Position`, you have to find out experimentally what viewing angle is suitable to make the scene fill a reasonable portion of the drawing area.

`ViewingAngle` does not have any effect when the attribute `OrthogonalProjection` = `TRUE` is set for the camera.

Note: In fact, when a parallel projection without perspective distortion is desired, one should *not* position the camera far away from the scene and use an extreme tele lense (i.e, very small `ViewingAngle` values). This may lead to problems with the hidden line algorithm of the 3D renderer. Use `OrthogonalProjection` = `TRUE` instead.

`ViewingAngle` can be animated. Increasing or decreasing values of `ViewingAngle` correspond to "zooming out" or "zooming in", respectively.

Examples

Example 1

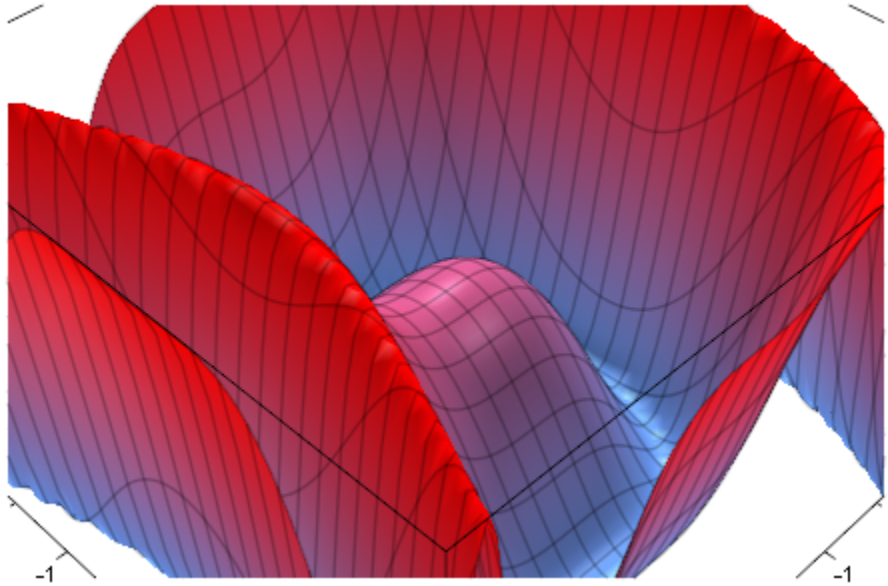
When creating a camera object, the third argument is the `ViewingAngle`:

```
camera:= plot::Camera([5, 5, 5], [0, 0, 0], PI/4):
camera::ViewingAngle
```

$$\frac{\pi}{4}$$

We animate `ViewingAngle`. With the initial value of $\frac{\pi}{3}$ the scene is fully visible (but rather small). Zooming in by decreasing the viewing angle, only parts of the scene are visible:

```
f  := plot::Function3d(sin(x^3 - y^2), x = -2..2, y = -2..2,
                       Submesh = [2, 2]):
camera:= plot::Camera([5, 5, 5], [0, 0, 0],
                      (1 - a)*PI/3 + a*PI/10,
                      a = 0..1, Frames = 200):
plot(f, camera)
```




```
delete f, camera:
```

See Also

MuPAD Functions

`CameraDirection` | `FocalPoint` | `OrthogonalProjection` | `Position`

AntiAliased

Antialiased lines and points

Value Summary

Inherited	FALSE, or TRUE

Graphics Primitives

Objects	AntiAliased Default Values
`plot::Arc2d`, `plot::Arrow2d`, `plot::Circle2d`, `plot::Conformal`, `plot::Curve2d`, `plot::Ellipse2d`, `plot::Function2d`, `plot::Hatch`, `plot::Histogram2d`, `plot::Implicit2d`, `plot::Integral`, `plot::Line2d`, `plot::Listplot`, `plot::Ode2d`, `plot::Parallelogram2d`, `plot::Piechart2d`, `plot::Point2d`, `plot::PointList2d`, `plot::Polar`, `plot::Polygon2d`, `plot::QQplot`, `plot::Rootlocus`, `plot::Scatterplot`, `plot::Sequence`, `plot::SparseMatrixplot`, `plot::Streamlines2d`, `plot::Turtle`, `plot::VectorField2d`	TRUE
`plot::Bars2d`, `plot::Boxplot`, `plot::Density`, `plot::Inequality`, `plot::Iteration`, `plot::Lsys`, `plot::Raster`, `plot::Rectangle`, `plot::Sum`	FALSE

Description

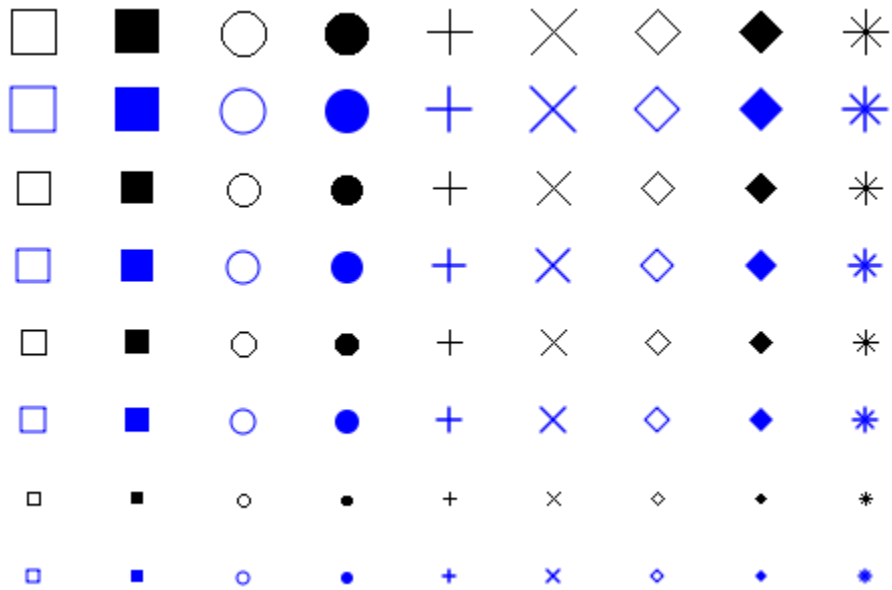
`AntiAliased` controls whether lines and points are drawn antialiased or not. With `AntiAliased` enabled graphics usually look smoother.

Examples

Example 1

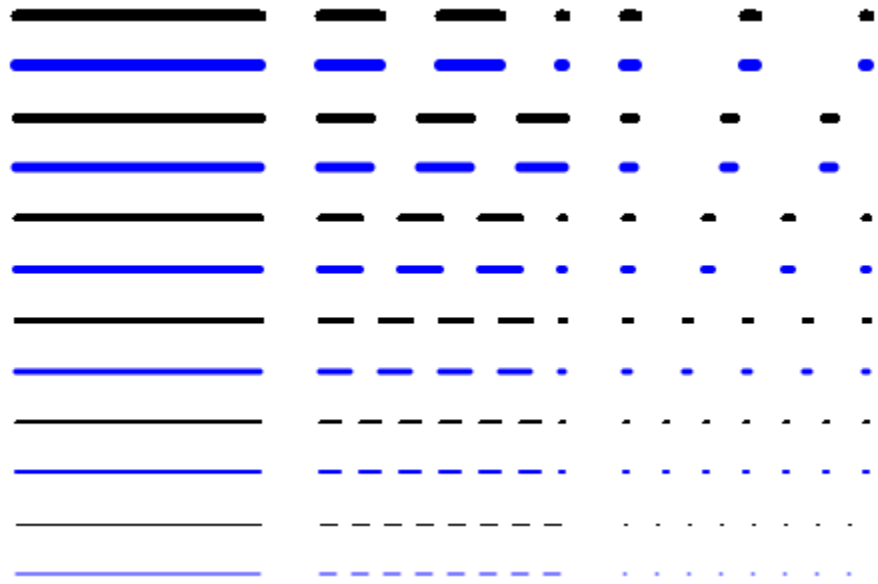
We draw a points in different sizes and point styles. The black points are drawn with `AntiAliased = FALSE`, the blue points are drawn with `AntiAliased = TRUE`:

```
pointStyles := [Squares, FilledSquares, Circles,
                FilledCircles, Crosses, XCrosses,
                Diamonds, FilledDiamonds, Stars]:
pointSizes := [1.5, 3, 4.5, 6]:
plot(Axes = None,
     (plot::Point2d(i, 2*j, AntiAliased = TRUE,
                            PointStyle = pointStyles[i],
                            PointSize = pointSizes[j],
                            Color = RGB::Blue),
      plot::Point2d(i, 2*j + 1, AntiAliased = FALSE,
                                PointStyle = pointStyles[i],
                                PointSize = pointSizes[j],
                                Color = RGB::Black))
      $ i = 1..nops(pointStyles) $ j = 1..nops(pointSizes)
    )
```



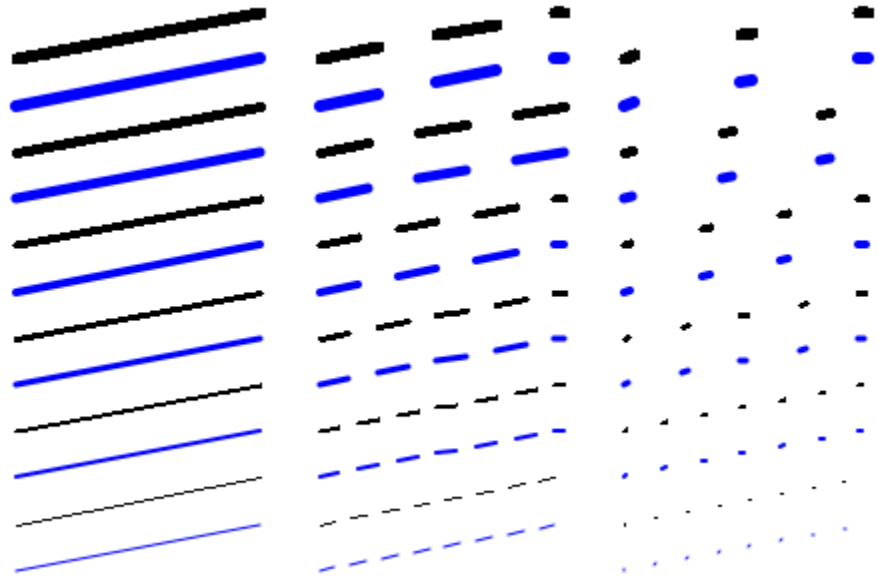
With horizontal lines we see not much difference between `AntiAliased = FALSE` (black lines) and `AntiAliased = TRUE` (blue lines):

```
lineStyles := [Solid, Dashed, Dotted]:
lineWidth := [.25, .5, .75, 1, 1.25, 1.5]:
plot(Axes = None,
     (plot::Line2d([i, 2*j], [i+.8, 2*j],
                    AntiAliased = TRUE,
                    LineStyle = lineStyles[i],
                    LineWidth = lineWidth[j],
                    Color = RGB::Blue),
      plot::Line2d([i, 2*j+1], [i+.8, 2*j+1],
                     AntiAliased = FALSE,
                     LineStyle = lineStyles[i],
                     LineWidth = lineWidth[j],
                     Color = RGB::Black))
      $ i = 1..nops(lineStyles) $ j = 1..nops(lineWidth)
    )
```



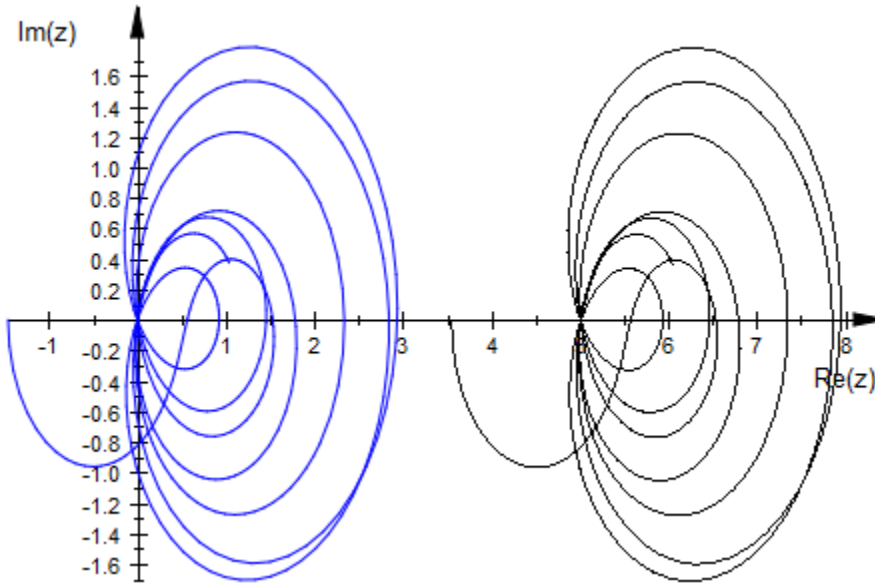
Diagonal lines are much smoother with `AntiAliased = TRUE`:

```
lineStyles := [Solid, Dashed, Dotted]:
lineWidth := [.25, .5, .75, 1, 1.25, 1.5]:
plot(Axes = None,
     (plot::Line2d([i, 2*j], [i + .8, 2*j + 1],
                           AntiAliased = TRUE,
                           LineStyle = lineStyles[i],
                           LineWidth = lineWidth[j],
                           Color = RGB::Blue),
      plot::Line2d([i, 2*j + 1], [i + .8, 2*j + 2],
                              AntiAliased = FALSE,
                              LineStyle = lineStyles[i],
                              LineWidth = lineWidth[j],
                              Color = RGB::Black))
     $ i = 1..nops(lineStyles) $ j = 1..nops(lineWidth)
    )
```



By default curves are plotted with `AntiAliased = TRUE` (blue curve) which is much nicer:

```
f := plot::Curve2d([Re, Im](zeta(I*y + 1/2)), y = 0..42,
                                    AdaptiveMesh = 3):
f1 := plot::modify(f, AntiAliased = FALSE,
                      Color = RGB::Black):
plot(
     f, plot::Translate2d([5, 0], f1), XAxisTitle = "Re(z)",
                                       YAxisTitle = "Im(z)"
    )
```



See Also

MuPAD Functions

`LineStyle` | `PointStyle`

ArrowLength

Scaling of arrows in a vector field

Value Summary

Inherited	`Fixed`, `Logarithmic`, or `Proportional`

Graphics Primitives

Objects	ArrowLength Default Values
`plot::VectorField2d`, `plot::VectorField3d`	`Proportional`

Description

`ArrowLength` determines how the lengths of the arrows in a vector field plot depend on the norms of the field at the evaluated points.

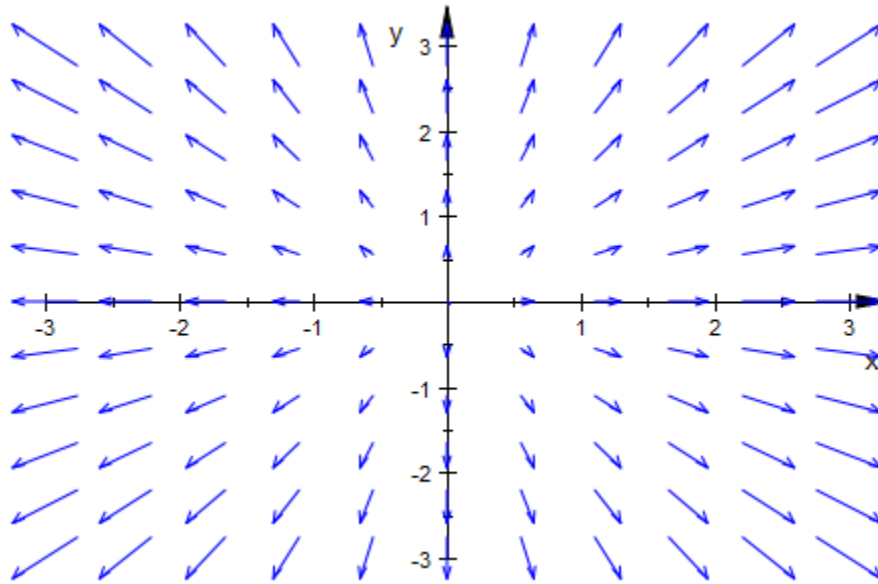
`plot::VectorField2d` plots a vector field by placing arrows at regular intervals, pointing in the directions of the field at these points. `ArrowLength` determines whether the lengths of those arrows are constant (`ArrowLength =Fixed`), proportional to the norms of the field (`ArrowLength =Proportional`, the default), or proportional to the logarithm of these values increased by 1 (`ArrowLength =Logarithmic`).

Examples

Example 1

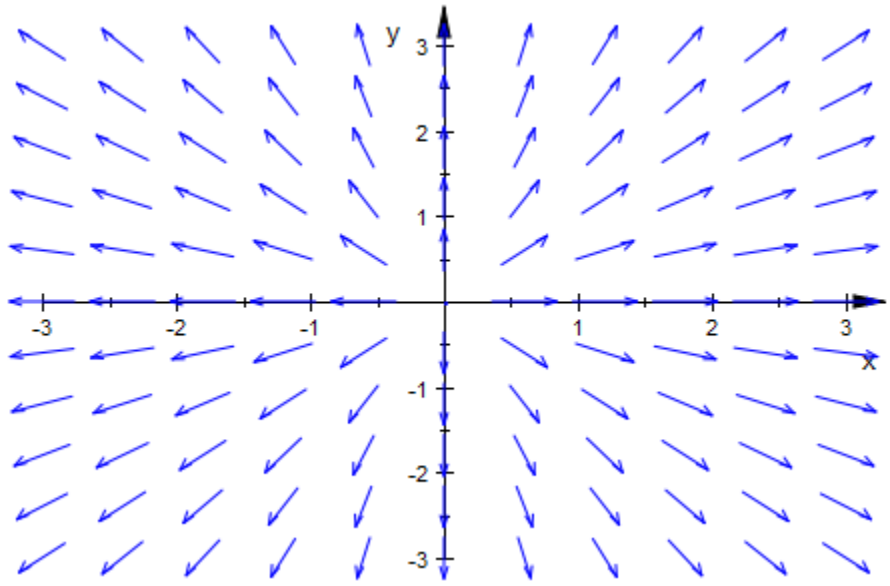
The vector field defined by $f(x, y) = (x, y)^t$ takes on different absolute values at different points. By default, `plot::VectorField2d` plots arrows whose lengths are proportional to the norms of the field:

```
v := plot::VectorField2d(x, y, x=-3..3, y=-3..3):
plot(v)
```



If you only want to display the direction of the field, not its "strength", use `ArrowLength =Fixed`:

```
v::ArrowLength := Fixed:
plot(v)
```


y
x
3
2
1
-1
-2
-3
-3
-2
-1
1
2
3

AxesTitleFont, FooterFont, HeaderFont, LegendFont, TextFont, TicksLabelFont, TitleFont

Font of axes titles

Value Summary

`AxesTitleFont`, `FooterFont`, `HeaderFont`, `LegendFont`, `TextFont`, `TicksLabelFont`, `TitleFont`	Inherited	Font definition

Graphics Primitives

Objects	Default Values

`plot::CoordinateSystem2d`, `plot::CoordinateSystem3d`	`AxesTitleFont`: `[" sans-serif ", 10]`
	
	`TicksLabelFont`: `[" sans-serif ", 8]`

`plot::Canvas`	`FooterFont`, `HeaderFont`: `[" sans-serif ", 12]`

`plot::Scene2d`, `plot::Scene3d`	`FooterFont`, `HeaderFont`: `[" sans-serif ", 12]`
	
	`LegendFont`: `[" sans-serif ", 8]`

`plot::Integral`, `plot::Piechart2d`, `plot::Piechart3d`, `plot::Text2d`, `plot::Text3d`	`TextFont`, `TitleFont`: `[" sans-serif ", 11]`

Description

`AxesTitleFont` etc. determine the fonts to be used for axes titles etc.

A font is specified as follows:

```
XXXFont = [< family >, < size >, <<Bold>>, <<Italic>, < color >>, <
alignment >]
```

The meaning of the parameters is as follows.

`family`	Font family name: a string. The available font families depend on the fonts that are installed on your machine. For example, typical font families available on Windows systems are `"Times New Roman"` (of type “serif”), `"Arial"` (of type “sans-serif”), or `"Courier New"` (of type “monospace”). To find out which fonts are available on your machine, open the menu “Format”, submenu “Font” in your MuPAD notebook. The first column in the font dialog provides the names of the font families that you may specify. You may also specify one the three generic family names `"serif"`, `"sans-serif"`, or `"monospace"`, and the system will automatically choose one of the available font families of the specified type for you.
`size`	Size of the font in integral points: a positive integer.
`Bold`	If specified, the font is bold.
`Italic`	If specified, the font is italic.
`color`	`RGB` color value: a list of 3 numerical values between 0 and 1
`alignment`	Text alignment in case of new-lines: one of the flags `Left`, `Center`, or `Right`.

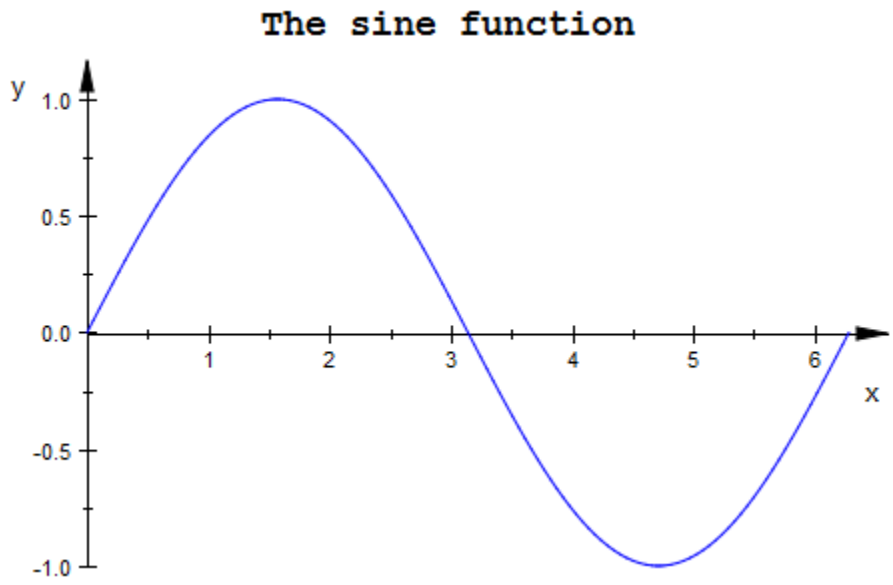
All font parameters are optional; some default values are chosen for entries that are not specified. For example, if you do not care about the footer font family for your plot, but you insist on a specific font size, you may specify an 18 pt font for the canvas footer by `FooterFont = [18]`.

Examples

Example 1

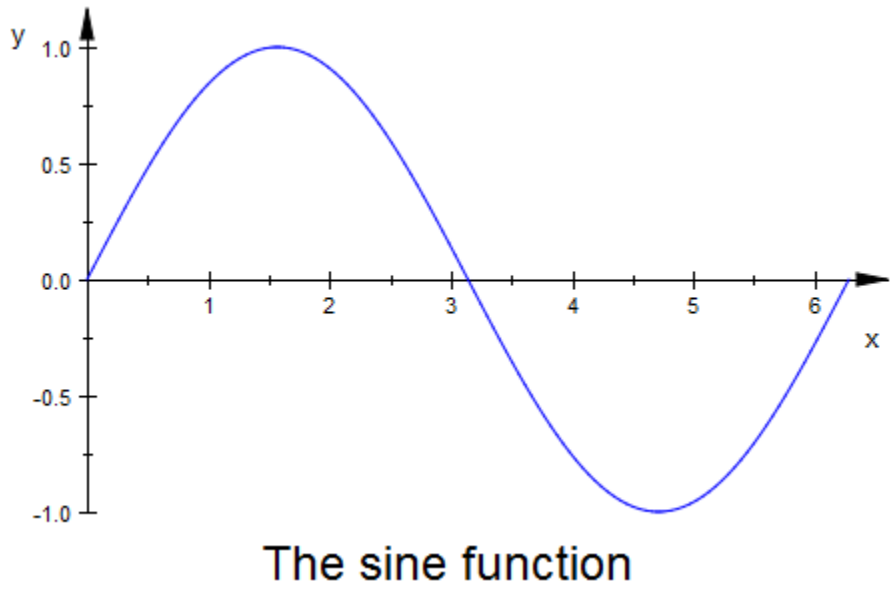
We specify the font for the canvas header:

```
plot(plot::Function2d(sin(x), x = 0 .. 2*PI),
     Header = "The sine function",
     HeaderFont = ["monospace", 14, Bold])
```



We specify a font size of 18 pt for the canvas footer:

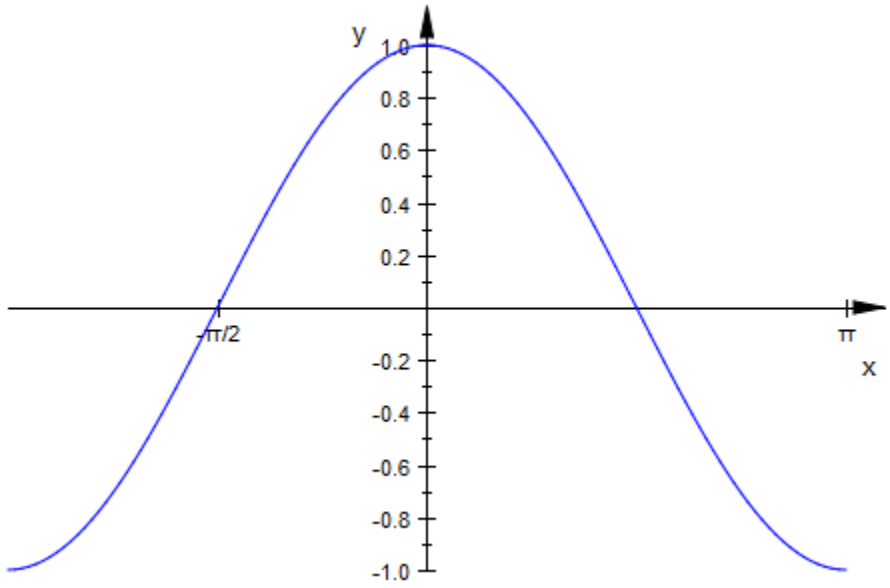
```
plot(plot::Function2d(sin(x), x = 0 .. 2*PI),
     Footer = "The sine function", FooterFont = [18])
```



Example 2

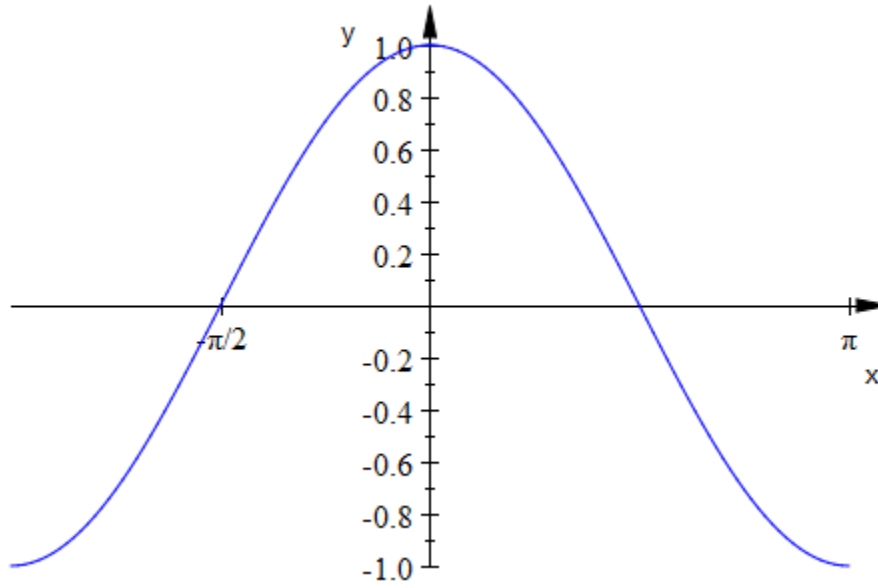
Display Greek characters for the tick labels:

```
plot(plot::Function2d(cos(x), x = -PI..PI),
     XTicksNumber = None,
     XTicksAt = [-PI/2 = "-π/2", PI = "π"])
```



You can change the appearance of Greek characters by specifying the font. Note that this font is used for all tick labels.

```
plot(plot::Function2d(cos(x), x = -PI..PI),
     XTicksNumber = None,
     XTicksAt = [-PI/2 = "-π/2", PI = "π"],
     TicksLabelFont = ["Times New Roman", 12])
```



See Also

MuPAD Functions

`AxesTitles` | `Footer` | `Header` | `LegendText` | `LegendVisible` | `TicksAt` | `TicksLabelsVisible` | `Title`

More About

- "Fonts"

BackgroundColor, BackgroundColor2

Background color

Value Summary

`BackgroundColor`, `BackgroundColor2`	Inherited	Color

Graphics Primitives

Objects	Default Values
`plot::Canvas`, `plot::Scene2d`	`BackgroundColor`: `RGB::White`

`plot::Scene3d`	`BackgroundColor`: `RGB::White`
	
	`BackgroundColor2`: `RGB::Grey75`

Description

These attributes set background colors for scenes, scene margins, and the remaining space in a canvas.

`BackgroundColor` sets the background color of a scene or canvas, where "background" refers to any area not occupied by graphical elements, including the margin.

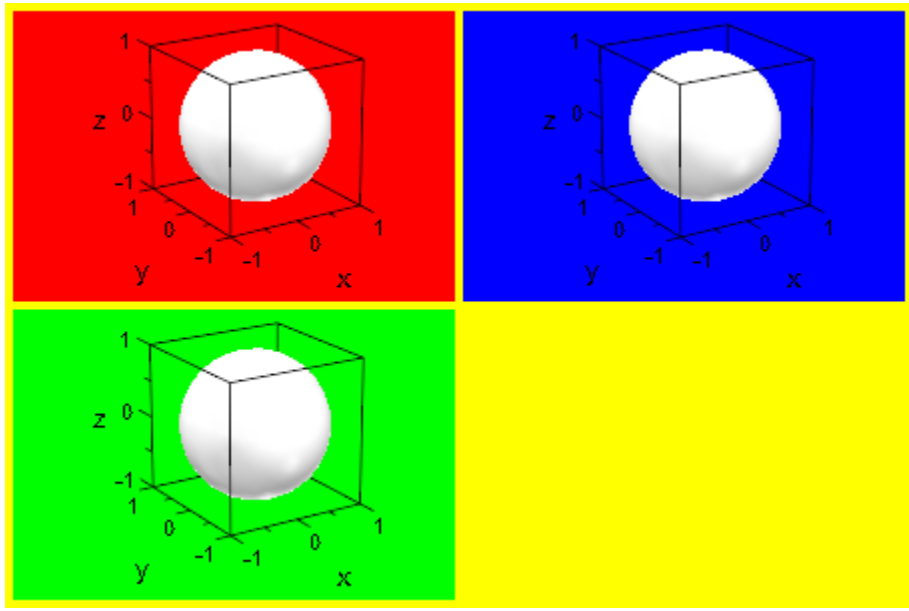
For a 3D-scene, if `BackgroundStyle` is not `Flat`, the actual scene background (not including the margin) is a blend from `BackgroundColor` to `BackgroundColor2`. See `BackgroundStyle` for details.

Examples

Example 1

In the following plot, we combine three scenes with backgrounds in red, blue, and green and set the background color of the canvas to yellow:

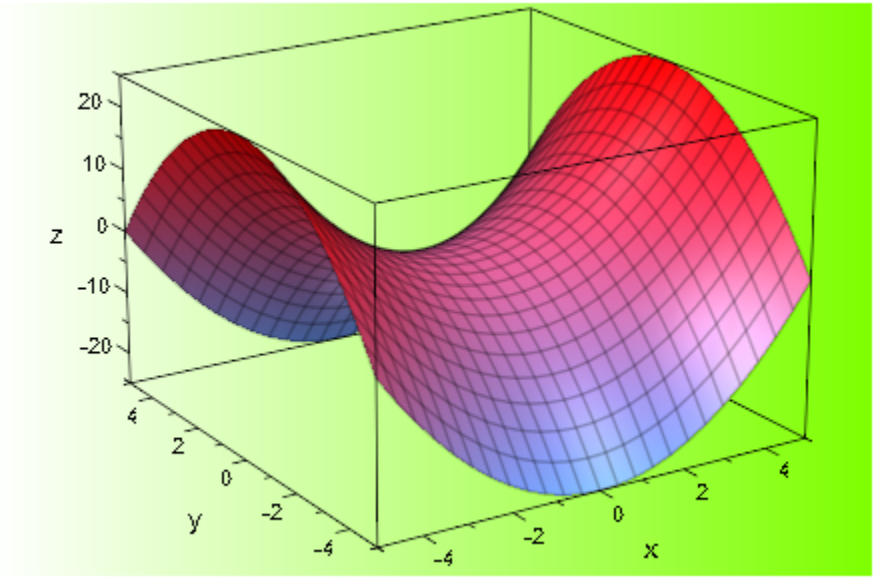
```
s1 := plot::Scene3d(plot::Sphere(1, Color = RGB::White),
                    BackgroundColor = RGB::Red):
s2 := plot::modify(s1, BackgroundColor = RGB::Blue):
s3 := plot::modify(s1, BackgroundColor = RGB::Green):
plot(s1, s2, s3, BackgroundColor = RGB::Yellow):
```



Example 2

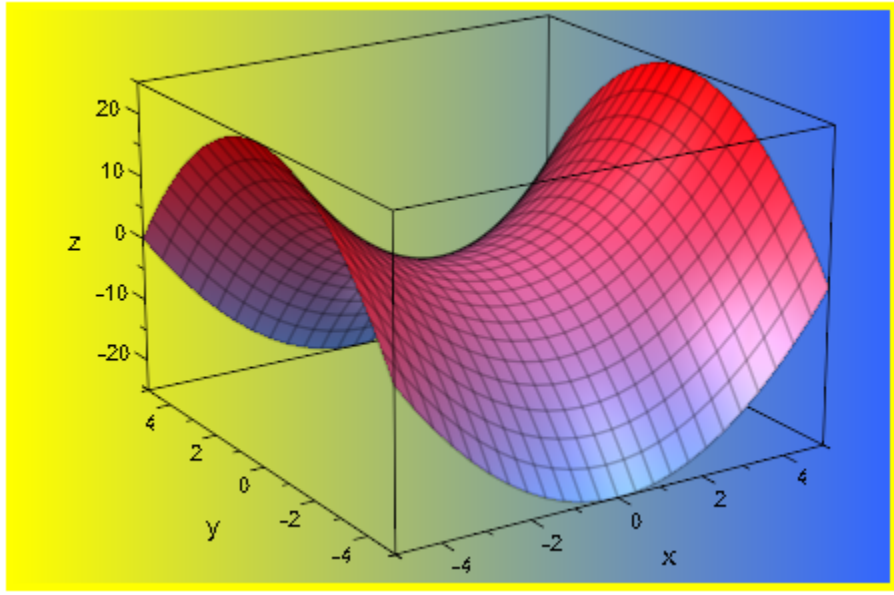
Using `BackgroundColor2` and `BackgroundStyle`, you can set the background of 3D-scenes to use a color blend:

```
plotfunc3d(x^2-y^2,
           BackgroundStyle = LeftRight,
           BackgroundColor2 = RGB::Chartreuse)
```



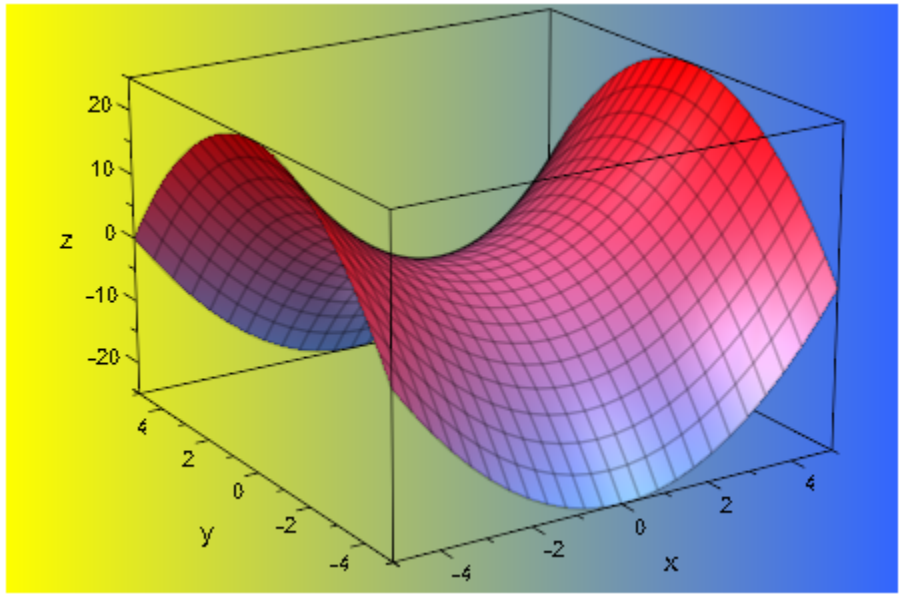
But note that the margin of the scene is still painted in its `BackgroundColor`:

```
plotfunc3d(x^2-y^2,
           plot::Scene3d::BackgroundColor = RGB::Yellow,
           plot::Scene3d::BackgroundStyle = LeftRight,
           plot::Scene3d::BackgroundColor2 = RGB::LightBlue)
```



To avoid this margin, we set its width to zero:

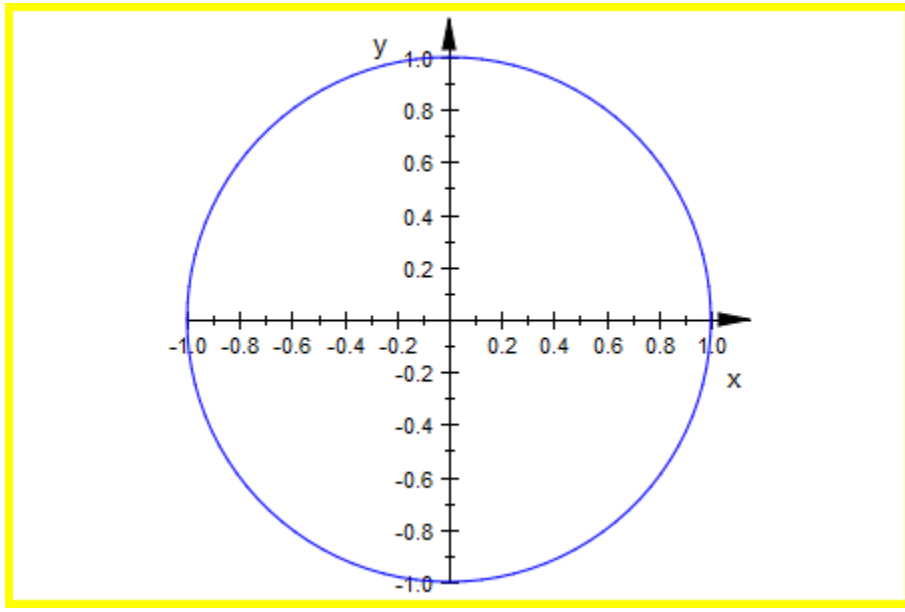
```
plotfunc3d(x^2-y^2,
           plot::Scene3d::BackgroundColor = RGB::Yellow,
           plot::Scene3d::BackgroundStyle = LeftRight,
           plot::Scene3d::BackgroundColor2 = RGB::LightBlue,
           plot::Scene3d::Margin = 0)
```



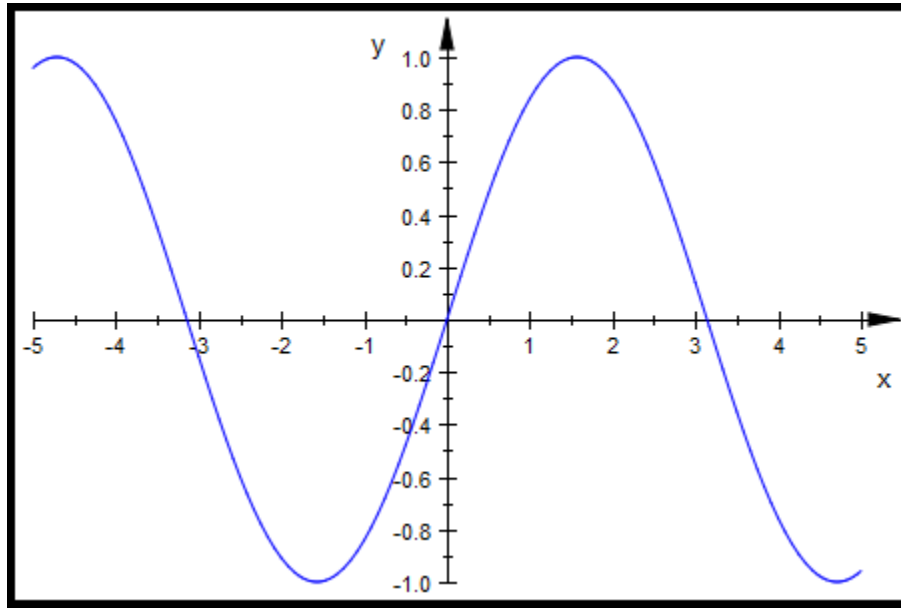
Example 3

The fact that `BackgroundColor` is an attribute of both canvas and scenes has the effect that giving it directly in a plot command will only affect the canvas, not the implicitly generated scenes of a plot:

```
plot(plot::Circle2d(1),
     BackgroundColor = RGB::Yellow)
```

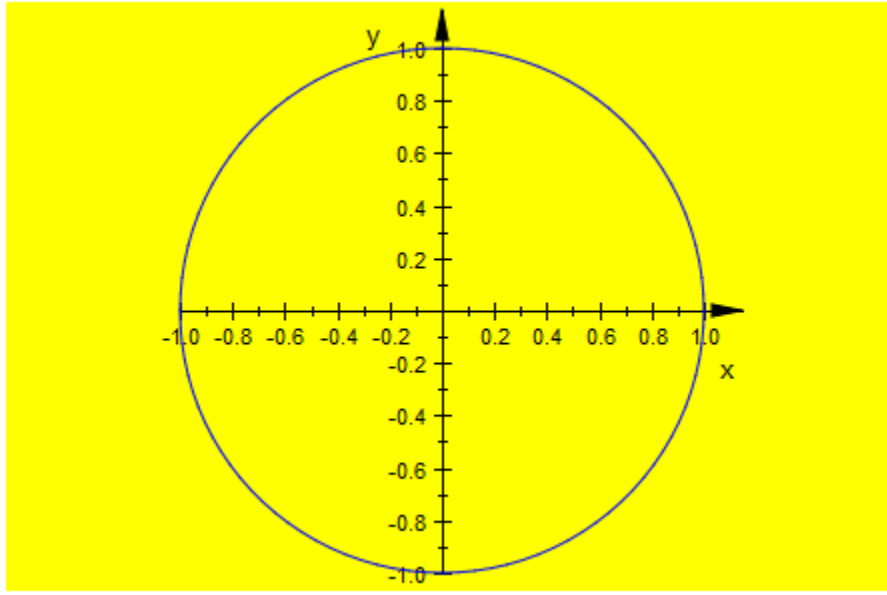



```
plotfunc2d(sin(x), BackgroundColor = RGB::Black)
```



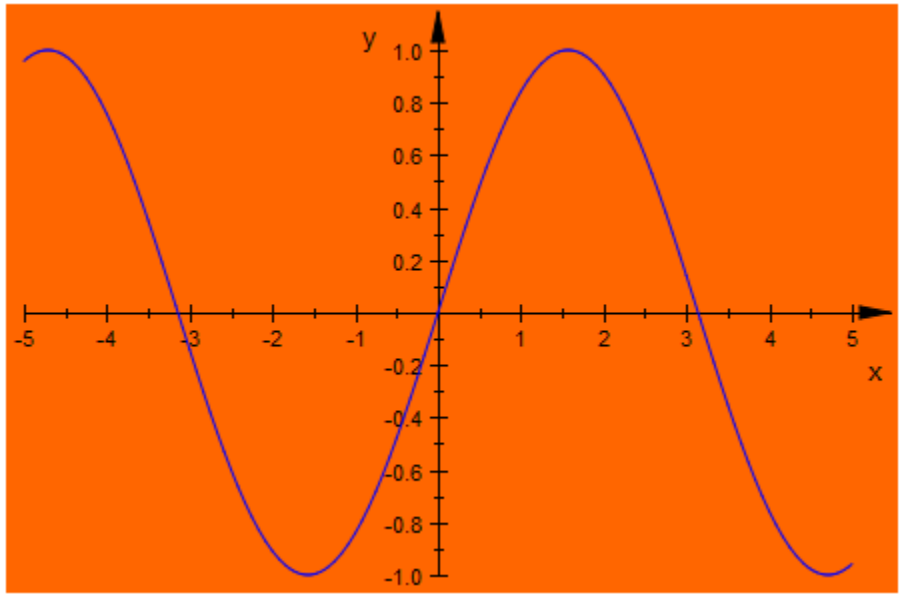
To set the background color of a scene, use one of the styles illustrated above: Either create a scene explicitly:

```
plot(plot::Scene2d(plot::Circle2d(1),
                   BackgroundColor = RGB::Yellow))
```



Or, set the attribute explicitly for scenes:

```
plotfunc2d(sin(x),
           plot::Scene2d::BackgroundColor = RGB::Orange)
```



There is also a third option, not used in the examples above: You can set `BackgroundColor` as a hint in an object to be shown (but this does not work for `plotfunc2d` and `plotfunc3d`):

```
plot(plot::Text2d("Sample", [0, 0],
                  TextFont = [RGB::White, 60],
                  HorizontalAlignment = Center,
                  BackgroundColor = RGB::Black))
```

See Also

MuPAD Functions
`BackgroundStyle` | `BackgroundTransparent` | `Margin`

BackgroundStyle

Color blends in the background

Value Summary

Inherited	`Flat`, `LeftRight`, `Pyramid`, or `TopBottom`

Graphics Primitives

Objects	BackgroundStyle Default Values
`plot::Scene3d`	`Flat`

Description

`BackgroundStyle` gives a color blend in the background of a 3D scene.

The background of a 3D scene may be set to a single color (`BackgroundStyle = Flat`, using `BackgroundColor`) or to a blend from `BackgroundColor` to `BackgroundColor2`, in one of three possible directions: `LeftRight` and `TopBottom` are linear blends from left to right or from top to bottom, respectively, while `Pyramid` sets a linear blend from the center to the borders.

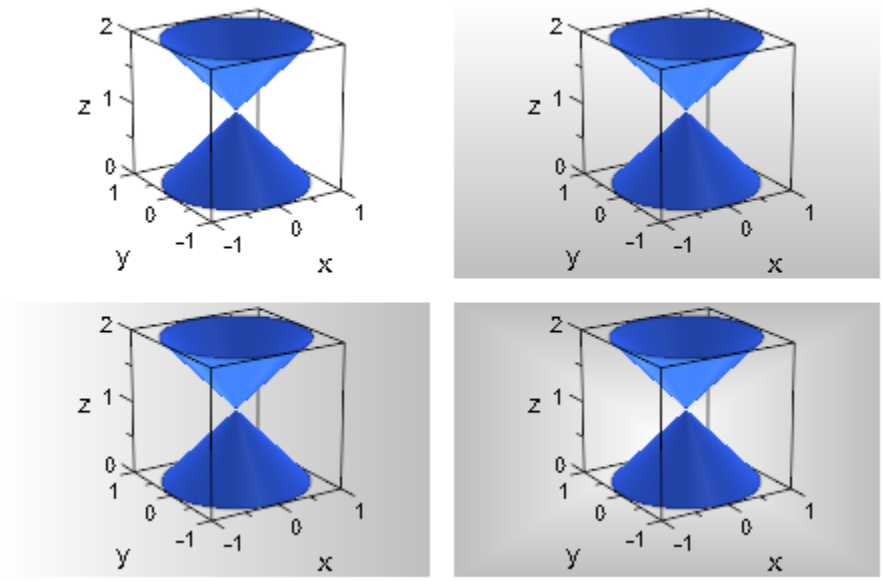
Examples

Example 1

We demonstrate all possible styles, using a simple plot and the default values for `BackgroundColor` and `BackgroundColor2`:

```
c1 := plot::Cone(1, [0, 0, 0], [0, 0, 1]):
c2 := plot::Cone(1, [0, 0, 2], [0, 0, 1]):
```

```
plot(plot::Scene3d(c1, c2, BackgroundStyle = Flat),
     plot::Scene3d(c1, c2, BackgroundStyle = TopBottom),
     plot::Scene3d(c1, c2, BackgroundStyle = LeftRight),
     plot::Scene3d(c1, c2, BackgroundStyle = Pyramid),
     Layout = Tabular)
```




```
delete c1, c2:
```

See Also

MuPAD Functions

`BackgroundColor` | `BackgroundColor2`

BackgroundTransparent

Plot a scene on a transparent background

Value Summary

Inherited	FALSE, or TRUE

Graphics Primitives

Objects	BackgroundTransparent Default Values
`plot::Scene2d`, `plot::Scene3d`	FALSE

Description

Using `BackgroundTransparent`, you can have a scene "without a background."

By default, each scene has an opaque background. In the case of overlapping scenes (which you can achieve by setting `Layout = Absolute` or `Layout = Relative` in the canvas and providing suitable values for `Bottom` and `Left` for the scenes), this may be undesirable. Using `BackgroundTransparent`, you can make the background of a scene transparent, so the canvas background and scenes behind it are visible.

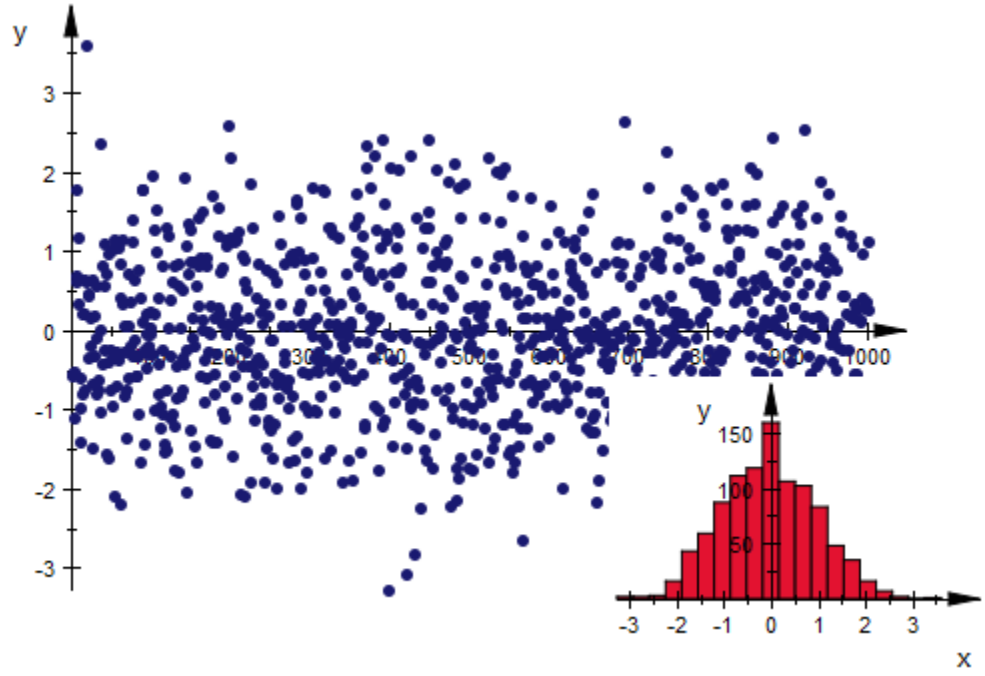
With `BackgroundTransparent = TRUE`, other background settings (`BackgroundColor`, `BackgroundStyle`, `BackgroundColor2`) are ignored.

Examples

Example 1

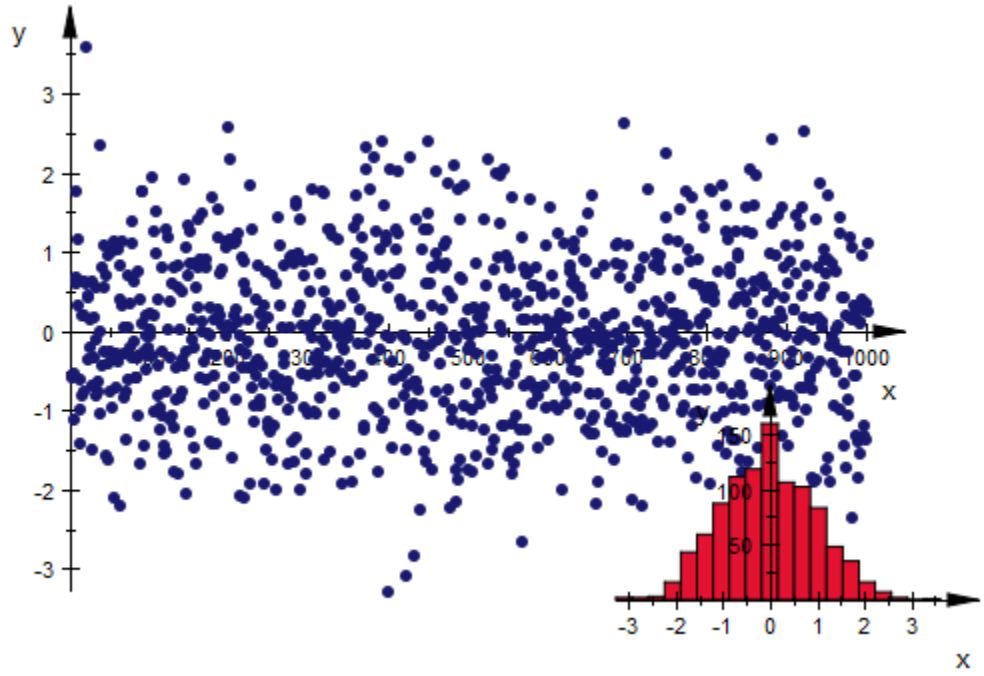
We create a number of random points and two statistical plots of this sample:

```
gen := stats::normalRandom(0, 1):
data := [gen() $ i = 1..1000]:
s1 := plot::Scene2d(plot::PointList2d([[i, data[i]]
                                       $ i = 1..1000]),
                    Left = 0, Bottom = 10,
                    Width = 120, Height = 80):
s2 := plot::Scene2d(plot::Histogram2d(data, Cells=[20]),
                    Left = 80, Bottom = 0,
                    Width = 50, Height = 40):
plot(s1, s2, Layout = Absolute, Width = 130, Height = 90)
```



The histogram plot obscures parts of the point list in a rectangle much larger than the bars of the histogram plot. Using `BackgroundTransparent`, we an set this rectangle to transparent:

```
s2::BackgroundTransparent := TRUE:
plot(s1, s2, Layout = Absolute, Width = 130, Height = 90)
```



See Also

MuPAD Functions

`BackgroundColor` | `BackgroundColor2` | `Bottom` | `Layout` | `Left`

Billboarding

Text orientation in space or towards observer

Value Summary

Inherited	FALSE, or TRUE

Graphics Primitives

Objects	Billboarding Default Values
`plot::Piechart3d`, `plot::Text3d`	TRUE

Description

With `Billboarding = TRUE`, text objects are always facing the observer. With `Billboarding = FALSE`, text objects retain their orientation relative to other objects.

Often, text objects in 3D are used to label certain places in a graphic (note that objects can contain a title, so text objects are usually only necessary for additional descriptions). In this case, it is desirable that they always face the observer to be readable and not rotate along with the rest of the scene. This is the default behavior. To get text objects that are actually part of the scene in the sense that rotating the scene also rotates the texts, set `Billboarding = FALSE`.

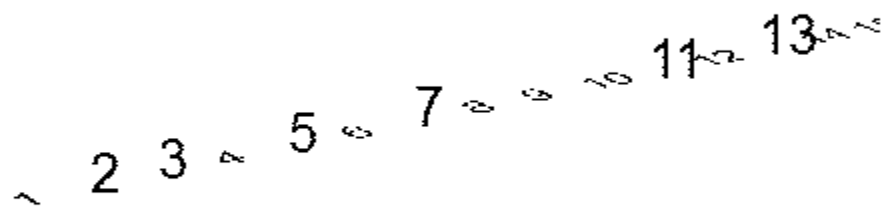
Examples

Example 1

In the following image, the prime numbers use `Billboarding = TRUE`, while other numbers do not:

```
plot(plot::Text3d(expr2text(i),
```

```
            [3*i, 0, 0],
            TextOrientation = [1, 0, 0, 0, 1, 0],
            Billboarding = isprime(i))
$ i = 1..15,
TextFont = [20], Scaling = Constrained, Axes = None)
```



Note that text objects with `Billboarding = TRUE` ignore `TextOrientation`.

See Also

MuPAD Functions
`Title`

BorderColor, BorderWidth

Color of frame/border around canvas and scenes

Value Summary

`BorderColor`, `BorderWidth`	Inherited	Color

Graphics Primitives

Objects	Default Values

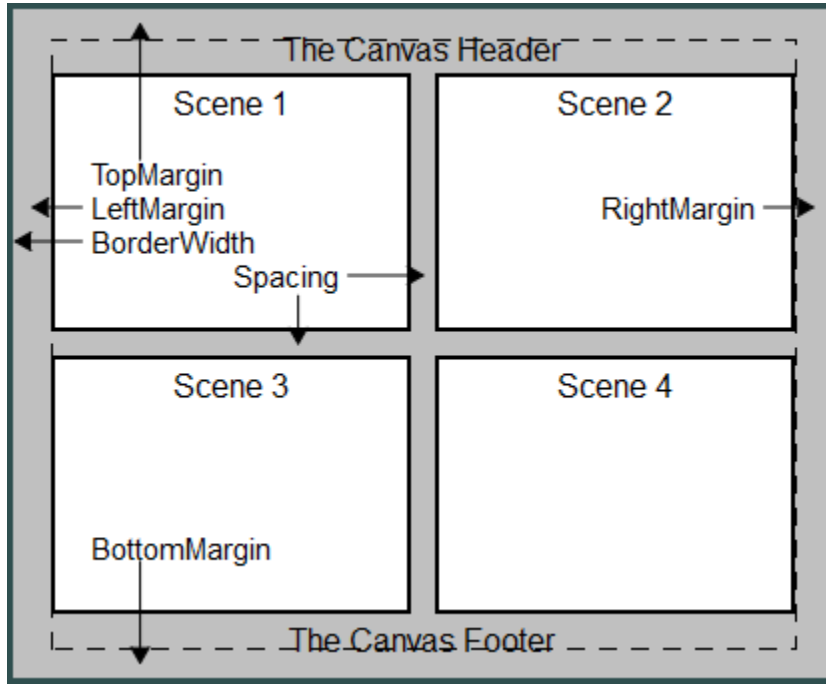
`plot::Canvas`, `plot::Scene2d`, `plot::Scene3d`	`BorderColor`: `RGB::Grey50`
	
	`BorderWidth`: `0`

Description

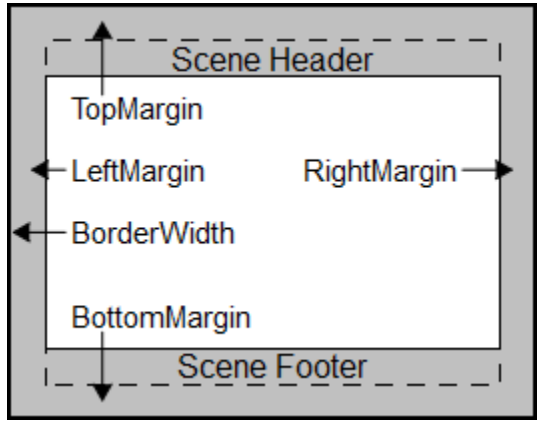
The canvas as well as the scenes in a canvas can be framed by a rectangular border. The width of the border is set by `BorderWidth`, its color is set by `BorderColor`.

With the attributes `BorderWidth` and `BorderColor`, a canvas or individual scenes can be given a border, similar to an image frame. The border is "switched off" with the default value `BorderWidth = 0`. Set the border width to some positive value such as `BorderWidth = 0.5*unit::mm` to make the border visible.

The following picture illustrates the layout of the canvas:



The following picture illustrates the layout of a scene:



The size of a canvas, set by the attributes `Width` and `Height`, includes the width of the border set by `BorderWidth`. The same holds for the scenes.

With `BackgroundTransparent = TRUE`, transparent scenes (without a background) can be created. The borders do *not* become transparent!

The scene borders do not react to `Layout = Relative`. One always has to specify the border width as absolut physical lengths such as `BorderWidth = 0.5*unit::mm`.

Scenes do *not* inherit borders from the enclosing canvas. You can set the borders for all scenes simultaneously by specifying them in `plot::setDefault` as

`plot::Scene2d::BorderWidth`, `plot::Scene2d::BorderColor`

or

`plot::Scene3d::BorderWidth`, `plot::Scene3d::BorderColor`,

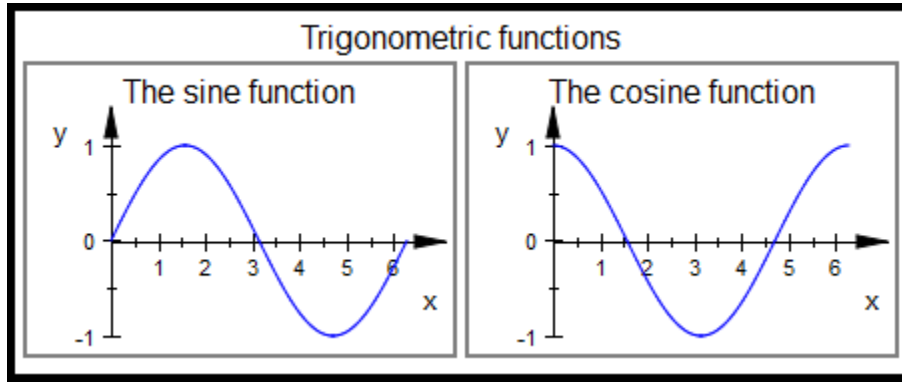
respectively. Cf. “Example 2” on page 24-1910.

Examples

Example 1

Two scenes are displayed side by side. The borders of the canvas and the two scenes are “switched on” by specifying positive values for `BorderWidth`:

```
S1 := plot::Scene2d(plot::Function2d(sin(x), x = 0 .. 2*PI),
                    Header = "The sine function",
                    BorderWidth = 0.5*unit::mm):
S2 := plot::Scene2d(plot::Function2d(cos(x), x = 0 .. 2*PI),
                    Header = "The cosine function",
                    BorderWidth = 0.5*unit::mm):
plot(S1, S2, Header = "Trigonometric functions",
     Width = 120*unit::mm, Height = 50*unit::mm,
     BorderWidth = 1.0*unit::mm, BorderColor = RGB::Black,
     Layout = Horizontal):
```




```
delete S1, S2:
```

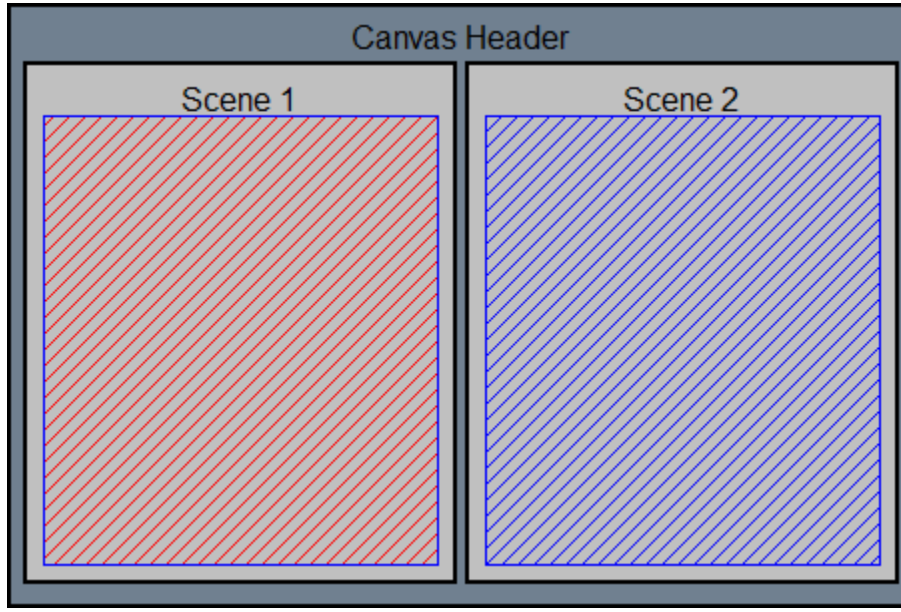
Example 2

We use `plot::setDefault` to define new default values for the layout and style parameters `BorderWidth`, `BorderColor`, `Margin`, and `BackgroundColor`:

```
plot::setDefault(
  plot::Canvas::BorderWidth = 0.5*unit::mm,
  plot::Canvas::BorderColor = RGB::Black,
  plot::Canvas::Margin = 1.5*unit::mm,
  plot::Canvas::BackgroundColor = RGB::SlateGrey,
  plot::Scene2d::BorderWidth = 0.5*unit::mm,
  plot::Scene2d::BorderColor = RGB::Black,
  plot::Scene2d::Margin = 2*unit::mm,
  plot::Scene2d::BackgroundColor = RGB::Grey
):
```

The following canvas contains two scenes. This plot uses the new defaults:

```
plot(plot::Scene2d(plot::Rectangle(-1..1, -1..1,
                        Filled = TRUE, FillColor = RGB::Red,
                        Header = "Scene 1")),
     plot::Scene2d(plot::Rectangle(-1..1, -1..1,
                        Filled = TRUE, FillColor = RGB::Blue,
                        Header = "Scene 2")),
     Layout = Horizontal, Axes = None,
     Header = "Canvas Header"):
```



See Also

MuPAD Functions

`BackgroundColor` | `BackgroundColor2` | `BackgroundStyle` | `BackgroundTransparent` | `Bottom` | `Left` | `Margin`

BoxCenters, BoxWidths

Position of boxes in a box plot

Value Summary

`BoxCenters`, `BoxWidths`	Optional	List of arithmetical expressions

Graphics Primitives

Objects	Default Values
`plot::Boxplot`	`BoxCenters`: [1] `BoxWidths`: [0.8]

Description

`BoxCenters` and `BoxWidths` govern horizontal center positions and widths of boxes in statistical box plots of Type `plot::Boxplot`.

A plot of type `plot::Boxplot` serves for visualizing and comparing statistical data samples.

A data sample defines the vertical coordinates of the corresponding box. The position along the horizontal axis as well as the horizontal width, however, is arbitrary and may be manipulated by the attributes `BoxCenters` and `BoxWidths`.

By default, the box of the i-th data sample is positioned at the horizontal value $x = i$. With the default width of 0.8, the i-th box extends from $x = i$ - 0.4 to $x = i$ + 0.4.

The value of the attribute `BoxCenters` must be a list of x-values for the horizontal centers of the boxes.

If the length of this list is smaller than the number of data samples in the box plot, the center values are incremented by 1 for each surplus box.

If the length of the `BoxCenters` list is larger than the number of data samples, the surplus center values are ignored.

Setting `BoxCenters` = `[`x_1`]`, the first box is centered at $x = x_1$, while the standard distance between the boxes is kept. Thus, `BoxCenters` = `[`x_1`]` allows to shift the entire box plot along the horizontal axis.

The value of the attribute `BoxWidths` must be a list of positive real values.

If the length of this list is smaller than the number of data samples in the box plot, the default width 0.8 is used for the surplus boxes.

If the length of the `BoxWidth` list is larger than the number of data samples, the surplus width values are ignored.

If the attribute `DrawMode` = `Horizontal` is set in the `plot::Boxplot` object, the boxes are drawn from left to right instead from bottom to top.

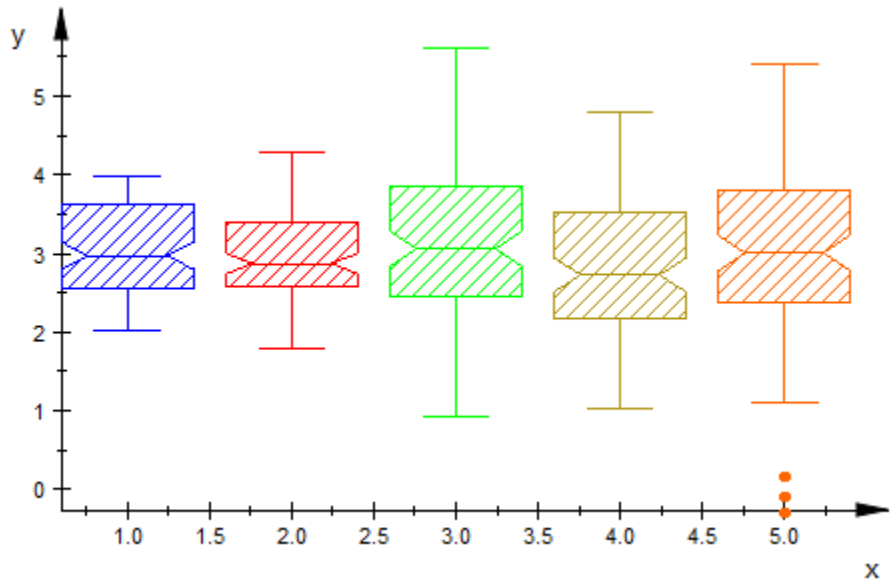
In this case, the attributes `BoxCenters` and `BoxWidths` refer to the vertical coordinates of the boxes.

Examples

Example 1

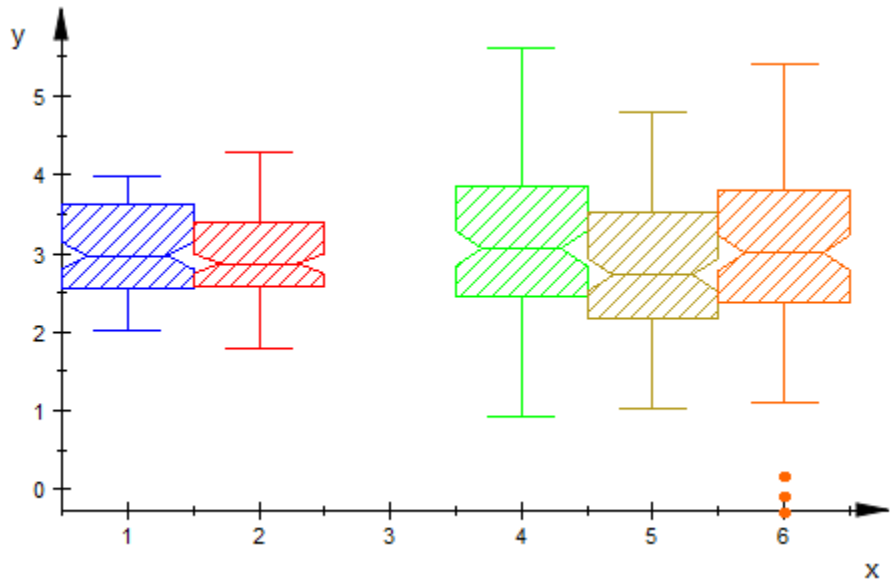
We create a box plot visualizing 5 data samples:

```
data1 := [stats::uniformRandom(2, 4)() $ k = 1..100]:
data2 := [stats::normalRandom(3, 0.3)() $ k = 1..100]:
data3 := [stats::normalRandom(3, 1)() $ k = 1..100]:
data4 := [stats::normalRandom(3, 1)() $ k = 1..100]:
data5 := [stats::normalRandom(3, 1)() $ k = 1..100]:
plot(plot::Boxplot(data1, data2, data3, data4, data5,
                   Notched = TRUE)):
```



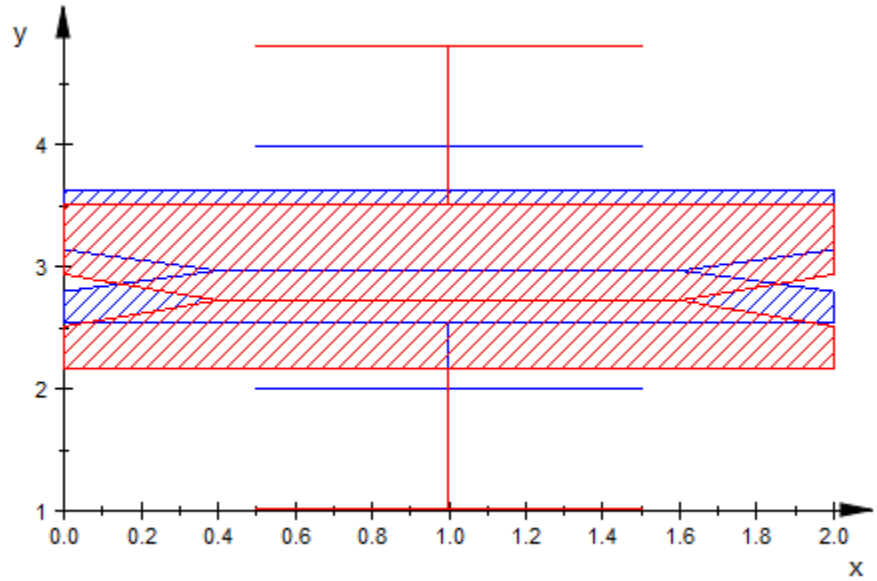
We specify the horizontal centers and the widths of the boxes such that the first two and the last three boxes touch each other:

```
plot(plot::Boxplot(data1, data2, data3, data4, data5,
                   Notched = TRUE,
                   BoxCenters = [1, 2, 4, 5, 6],
                   BoxWidths = [1, 1, 1, 1, 1])):
```



We place two of the data boxes on top of each other for direct comparison:

```
plot(plot::Boxplot(data1, data4, Notched = TRUE,
                   BoxCenters = [1, 1], BoxWidths = [2, 2]))
```




```
delete data1, data2, data3, data4, data5:
```

See Also

MuPAD Functions

`DrawMode` | `Notched` | `NotchWidth`

DrawMode

Orientation of boxes and bars

Value Summary

Optional	`Horizontal` or `Vertical`

Graphics Primitives

Objects	DrawMode Default Values
`plot::Bars2d`, `plot::Boxplot`, `plot::Histogram2d`	`Vertical`

Description

`DrawMode` = `Vertical` versus `DrawMode` = `Horizontal` determines the orientation of boxes in a box plot and bars in bar plots and histogram plots.

A plot of type `plot::Boxplot` serves for visualizing and comparing statistical data samples. The plot reduces the data to few simple descriptive parameters.

One coordinate direction provides information on the statistical data (25% quantile, median, 75% quantile etc.). The other coordinate direction just serves for placing several boxes associated with different data samples side by side for comparison.

With `DrawMode` = `Vertical`, the vertical direction provides the information on the statistical data.

With `DrawMode` = `Horizontal`, the boxes are turned by 90 degrees. Now, the horizontal direction provides the information on the statistical data.

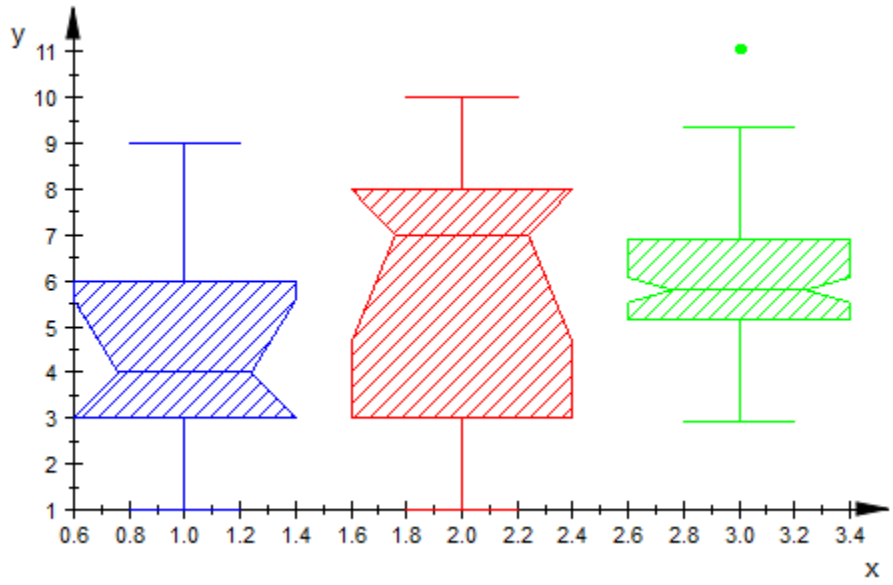
Corresponding statements hold for the bars in a 2D bar plot of type `plot::Bars2d` and 2D histograms of type `plot::Histogram2d`.

Examples

Example 1

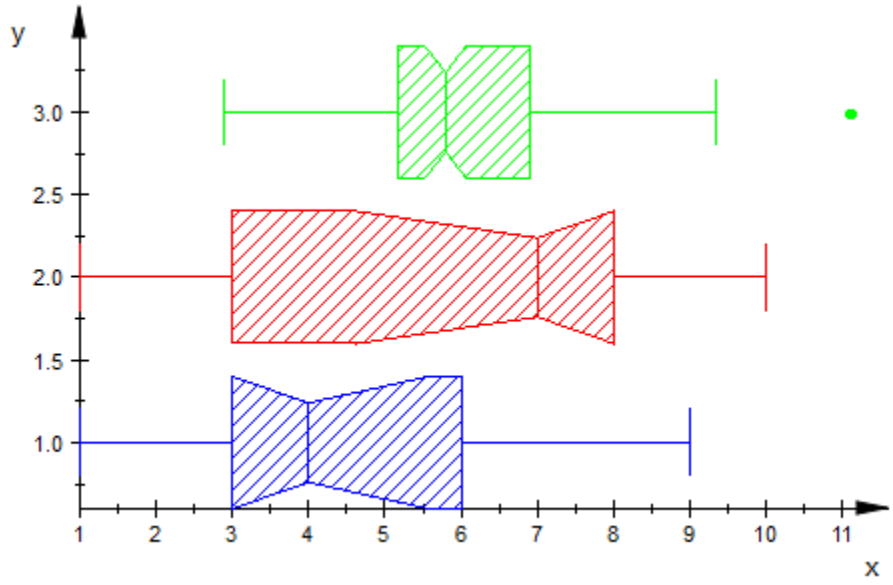
We create a box plot:

```
data1 := [2, 6, 4, 3, 1, 7, 9, 5, 3]:
data2 := [2, 4, 8, 8, 7, 6, 8, 7, 3, 1, 10]:
data3 := [stats::normalRandom(6, 2)() $ k = 1 .. 100]:
plot(plot::Boxplot(data1, data2, data3, Notched = TRUE)):
```



The boxes are rendered horizontally:

```
plot(plot::Boxplot(data1, data2, data3, Notched = TRUE,
                   DrawMode = Horizontal)):
```




```
delete data1, data2, data3:
```

See Also

MuPAD Functions

BoxCenters | BoxWidths | Notched | NotchWidth

Gap, XGap, YGap

Gaps between the bars of a bar chart

Value Summary

`Gap`	[[`XGap`, `YGap`]]	See below
`XGap`, `YGap`	Optional	MuPAD expression

Graphics Primitives

Objects	Default Values
`plot::Bars3d`	`Gap`: [0, 0] `XGap`, `YGap`: 0

Description

`Gap`, `XGap`, `YGap` sets gaps between the bars of a bar chart.

In `plot::Bars3d`, the attribute `Gap = [`g_x`, `g_y`]` or, equivalently, `XGap =` g_x, `YGap =` g_y allows to introduce gaps between adjacent bars. The values g_x, g_y may be real numerical values between 0 and 1 or expressions of the animation parameter. These values set the fraction of the space reserved for a bar that is not filled by the bar.

With g_x `= 0`, g_y `= 0`, there are no gaps. With g_x `= 0.5`, g_y `= 0.5`, the gaps between adjacent bars are of the same size as the bars. With g_x `= 1`, g_y `= 1`, there bars become lines.

Values of g_x, g_y larger than 1 are treated like 1, negative values like 0.

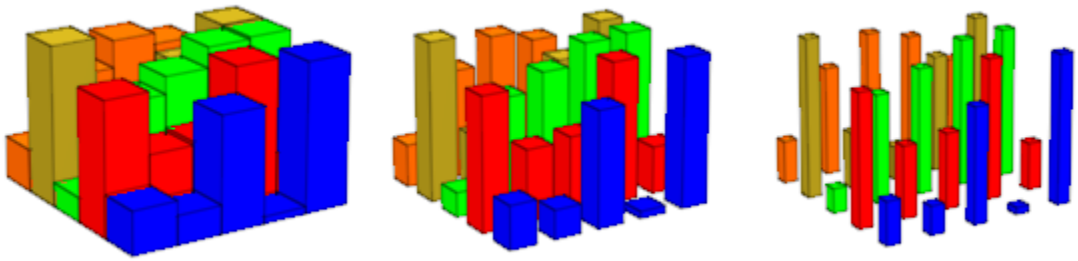
The `Gap` attribute has an effect only for `BarStyle = Boxes`.

Examples

Example 1

We display the same data with different `Gap` values:

```
A := matrix::random(5, 5, frandom) :
plot(plot::Scene3d(plot::Bars3d(A, Gap = [0, 0])),
     plot::Scene3d(plot::Bars3d(A, Gap = [0.4, 0.4])),
     plot::Scene3d(plot::Bars3d(A, Gap = [0.7, 0.7])),
     Width = 150*unit::mm, Height = 50*unit::mm,
     Layout = Horizontal):
```



```
delete A:
```

Notched, NotchWidth

Notched boxes in box plots

Value Summary

`Notched`, `NotchWidth`	Optional	`TRUE` or `FALSE`

Graphics Primitives

Objects	Default Values

`plot::Boxplot`	`Notched`: `FALSE`
	
	`NotchWidth`: `0.2`

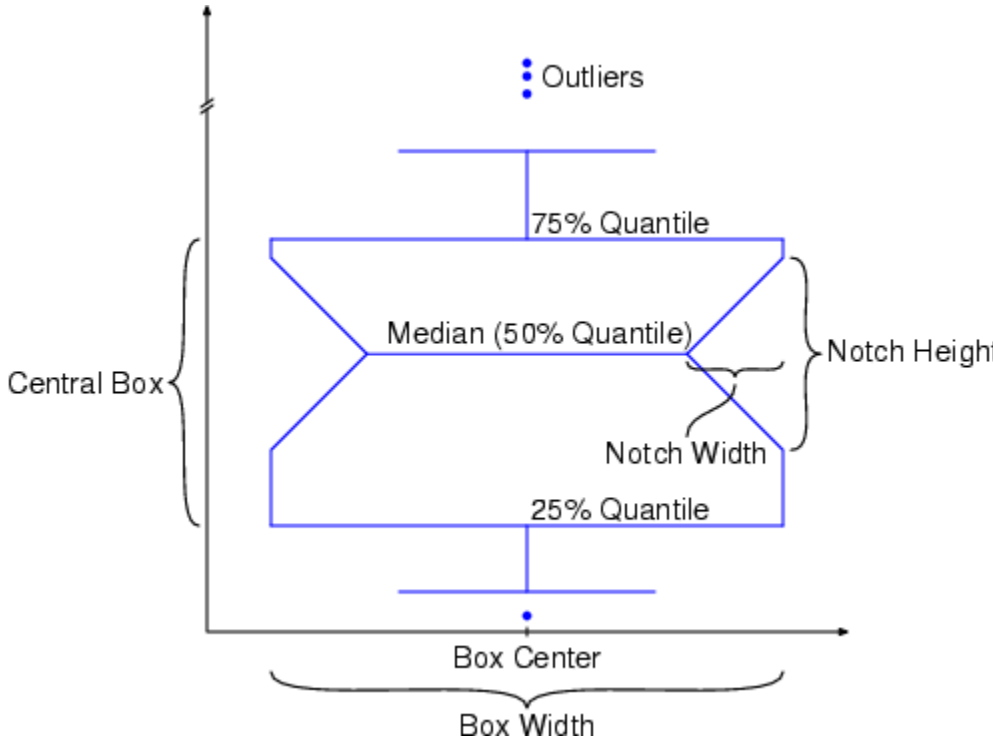
Description

With `Notched = TRUE`, the boxes in a statistical box plot of type `plot::Boxplot` are notched. The notches provide further information on the statistical data.

The attribute `NotchWidth` determines the horizontal width of the notches.

A plot of type `plot::Boxplot` serves for visualizing and comparing statistical data samples. The plot reduces the data to few simple descriptive parameters.

One graphical parameter is the height of notches that are displayed in the sides of the boxes when using `Notched = TRUE`. A typical notched box looks like this:



The height of the notches is 3.14 times the height of the central box divided by the square root of the number of data elements in the corresponding data sample.

Notched box plots are useful for determining whether two random samples were drawn from the same population. Similar notches of boxes indicate that the data visualized by the boxes have the same distribution.

This, however, is not a *rigorous* criterion that the data samples are indeed identically distributed.

The horizontal width of the notches bears no statistical significance and is just a layout parameter. Setting `NotchWidth` = *r*, the absolute horizontal notch width of a box is *r* times the width of the box. Reasonable values for *r* lie between 0 and $\frac{1}{2}$.

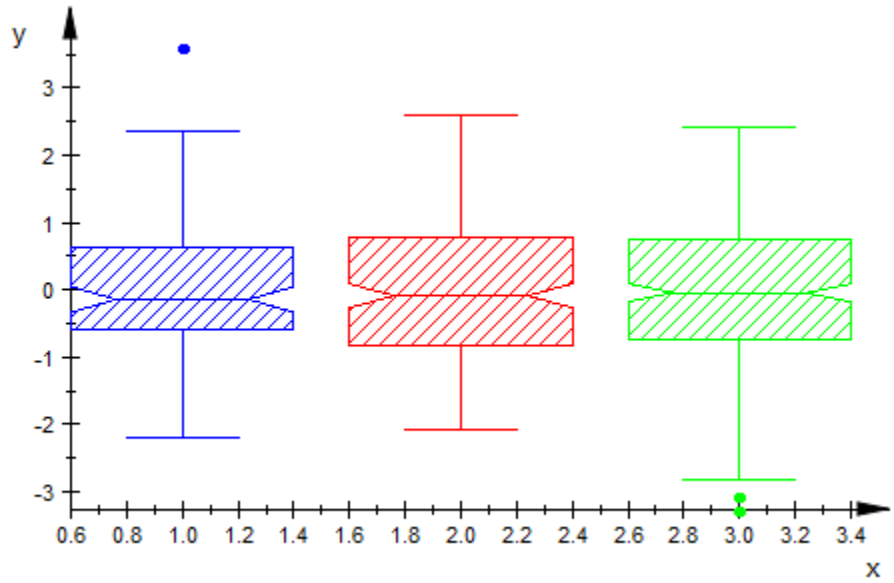
The widths of the boxes can be set via the attribute `BoxWidths`.

Examples

Example 1

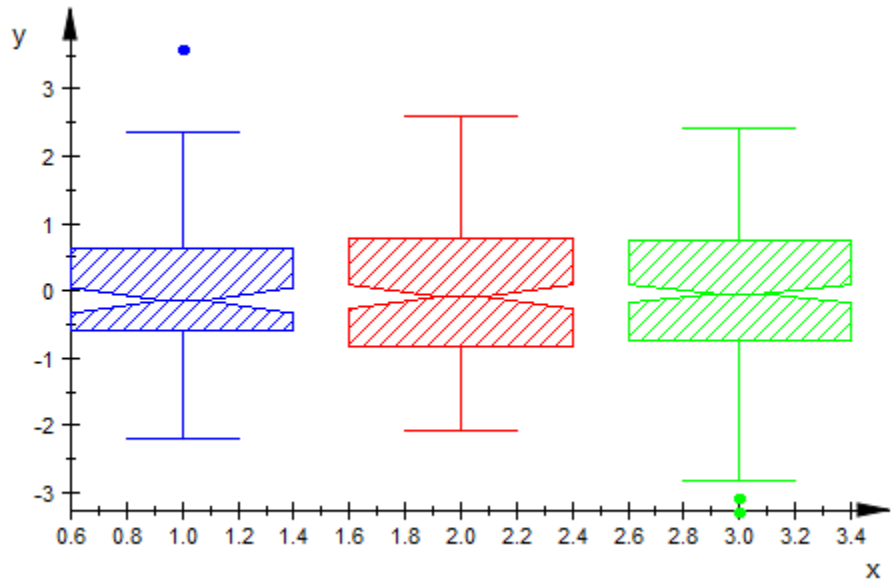
We create a notched box plot of several random samples:

```
r := stats::normalRandom(0, 1):
data1 := [r() $ k = 1..100]:
data2 := [r() $ k = 1..200]:
data3 := [r() $ k = 1..300]:
plot(plot::Boxplot(data1, data2, data3, Notched = TRUE)):
```



We change the `NotchWidth`:

```
plot(plot::Boxplot(data1, data2, data3, Notched = TRUE,
                   NotchWidth = 0.4)):
```




```
delete r, data1, data2, data3:
```

See Also

MuPAD Functions

`BoxCenters` | `BoxWidths` | `DrawMode`

Projectors

Project an ODE solution to graphical points

Value Summary

Mandatory	List of arithmetical expressions

Graphics Primitives

Objects	Projectors Default Values
`plot::Ode2d`, `plot::Ode3d`	

Description

`Projectors` defines "generators of plot data" that project solution points $(t, Y(t))$ of an ODE to graphical points $[x, y]$ in 2D or $[x, y, z]$ in 3D, respectively.

Internally, `plot::Ode2d` and `plot::Ode3d` generate a sequence of numerical solution points $(t_0, Y(t_0))$, $(t_1, Y(t_1))$ etc. of an ODE. Each of these solution points is mapped to a graphical point via the "projectors" defined by `Projectors`.

Each projector G_i in `Projectors = [[G1], [G2], ...]` is a list

$[G_i]$=`[(t, Y) -> [x(t,Y), y(t,Y), <z(t, Y)>], <Style = style>, <Color = color>]` .

The procedures `(t, Y) -> [x(t, Y), y(t, Y), <z(t, Y)>]` map the solution points (t_i, Y_i) of the ODE to points $[x(t_i, Y_i), y(t_i, Y_i)]$ in 2D (for `plot::Ode2d`) or $[x(t_i, Y_i), y(t_i, Y_i), z(t_i, Y_i)]$ in 3D (for `plot::Ode3d`). These points are drawn in the picture, interpolated by linear or cubic spline interpolation according to the attribute `Style= style` in the color set by the attribute `Color = color`.

The style parameter may be one of the flags `Points` (only the points are displayed), `Lines` (only interpolating line segments are displayed), `Splines` (only the interpolating cubic spline curve is displayed), `[Lines, Points]` (interpolating line segments together

with the interpolation points are displayed) , or `[Splines, Points]` (the interpolating cubic spline curve together with the interpolation points are displayed).

The default style is `Style= [Splines, Points]`.

Each of the projectors G_1, G_2 etc. (denoted by G in the following) is a mapping $G\colon (t, Y) \to [x(t, Y), y(t, Y)]$ in 2D or $G\colon (t, Y) \to [x(t, Y), y(t, Y), z(t, Y)]$ in 3D. It must accept a numerical argument t and a vector Y (a list or a one-dimensional array) and must return a list of numerical coordinate values `[x, y]` (in `plot::Ode2d`) or `[x, y, z]` (in `plot::Ode3d`), respectively. Defining appropriate projectors, any information on the solution curve of the ODE can be displayed graphically.

Here are some examples:

`G := (t, Y) -> [t, Y[1]]` creates a 2D plot of the first component of the solution vector along the y-axis, plotted against the time variable `t` along the x-axis

`G := (t, Y) -> [Y[1], Y[2]]` creates a 2D phase plot, plotting the first component of the solution along the x-axis and the second component along the y-axis. The result is a solution curve in phase space (parametrized by the time `t`).

`G := (t, Y) -> [Y[1], Y[2], Y[3]]` creates a 3D phase plot of the first three components of the solution curve.

If no projectors are specified in a call to `plot::Ode2d`, the default projectors `Generators = [[`G_1`], [`G_2`], …]` are used, where

`[G_i] = [(t, Y) -> [t, Y[i]], Style = [Splines, Points]]`.

This plots the i-th component of the solution vector along the y-axis against the "time" t plotted along the x-axis.

In `plot::Ode3d`, the default projectors are

```
[G_i] = [(t, Y) -> [t, Y[2*i - 1], Y[2*i]], Style =
[Splines,Points]]
```

This plots two of the components of the solution vector along the y- and z-axis against the "time" t plotted along the x-axis.

Examples

Example 1

We consider the 2nd order ODE $y'' = -y + \sin(3\,y)$, $y(0) = 1$, $y'(0) = 0$. As a dynamical system for $Y = (Y_1, Y_2) = (y, y')$, the ODE to be solved is

$$\frac{d}{dt}\,Y = \frac{d}{dt}\begin{pmatrix} Y_1 \\ Y_2 \end{pmatrix} = \begin{pmatrix} Y_2 \\ -Y_1 + \sin(3\,Y_1) \end{pmatrix}.$$

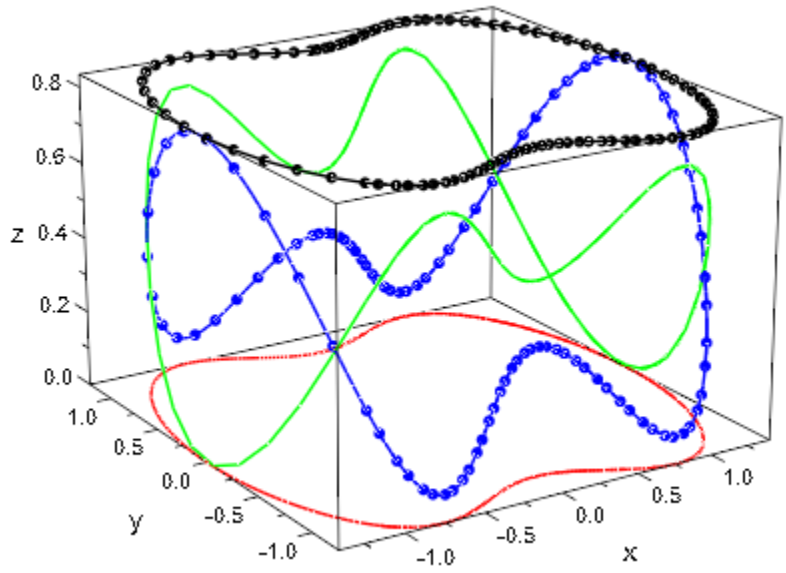
The first projector G_1 plots the solution in red as a phase curve in the (x, y)-plane.

The second projector G_2 plots the kinectic energy $\frac{{Y_2}^2}{2} = \frac{y'^2}{2}$ in green along the z-axis.

The third projector G_3 plots the potential energy ${Y_1}^2 + \frac{\cos(3\,Y_1)}{3} = \frac{y^2}{2} + \frac{\cos(3\,y)}{3}$ in blue along the z-axis.

The fourth projector G_4 plots the total energy in black along the z-axis:

```
f := (t, Y) -> [Y[2], - Y[1] + sin(3*Y[1])]:
Y0 := [0, 1]:
G1 := (t, Y) -> [Y[1], Y[2], 0]:
G2 := (t, Y) -> [Y[1], Y[2], Y[2]^2/2]:
G3 := (t, Y) -> [Y[1], Y[2], Y[1]^2/2 + cos(3*Y[1])/3]:
G4 := (t, Y) -> [Y[1], Y[2], Y[1]^2/2 + cos(3*Y[1])/3 + Y[2]^2/2]:
plot(plot::Ode3d(
      f, [i/10 $ i = 0..100], Y0,
      [G1, Style = Splines, Color = RGB::Red],
      [G2, Style = Lines, Color = RGB::Green],
      [G3, Color = RGB::Blue],
      [G4, Style = [Lines, Points], Color = RGB::Black]))
```




```
delete f, Y0, G1, G2, G3, G4:
```

See Also

MuPAD Functions

`AbsoluteError` | `InitialConditions` | `ODEMethod` | `RelativeError` | `Stepsize` | `TimeMesh`

Scaling, YXRatio, ZXRatio

Scaling ratios

Value Summary

`Scaling`	Inherited	`Automatic`, `Constrained`, or `Unconstrained`
`YXRatio`, `ZXRatio`	Inherited	Positive real numberReal number

Graphics Primitives

Objects	Default Values
`plot::CoordinateSystem2d`, `plot::CoordinateSystem3d`	`Scaling`: `Unconstrained`

`plot::Scene3d`	`YXRatio`: `1`
	
	`ZXRatio`: `2/3`

Description

With `Scaling = Constrained`, the graphics output is scaled like the model coordinates, i.e., circle appear as circles, spheres as spheres.

With `Scaling = Unconstrained`, the graphics output is scaled independently in each coordinate direction such that the graphics fits optimally into the viewing area. Circles may appear as ellipses, spheres as ellipsoids.

For `Scaling = Unconstrained`, the scaling ratios of the different coordinate directions in a 3D plot can be set via the attributes `YXRatio` and `ZXRatio`.

If the graphics consists of geometrical objects such as circles, pie charts, spheres etc., the setting `Scaling = Constrained` is appropriate. This prevents circles from being deformed to ellipses in the graphical output.

For the visualization of non-geometrical data (usually, in function plots etc.), a scaling constrained to model coordinates is usually not appropriate. Think of the graph of $y = e^x$ for $x \in [0, 10]$, where the y values extend over the range $y \in [e^0, e^{10}]$, which is roughly $[1, 22026]$. With `Scaling = Constrained`, the graphical output would consist of a narrow vertical strip with the side ratio `y : x = 22025 : 10`. Here, `Scaling = Unconstrained` is appropriate.

The default value is `Scaling = Unconstrained`. However, many "geometrical" objects in the MuPAD `plot` library override this default setting via the "hint mechanism" (see section Primitives Requesting Special Scene Attributes: "Hints" in this document). Whenever such an object is plotted in a scene, the whole scene uses `Scaling = Constrained`. A complete list of these "geometrical objects" such as circles, spheres, cones etc. is given further up on this help page.

With `Scaling = Automatic`, the graphics uses `Scaling = Constrained` for plots in which the coordinate ranges to be displayed have a ratio close to `1 : 1` in 2D or `1 : 1 : 1` in 3D. Otherwise, `Scaling = Unconstrained` is used.

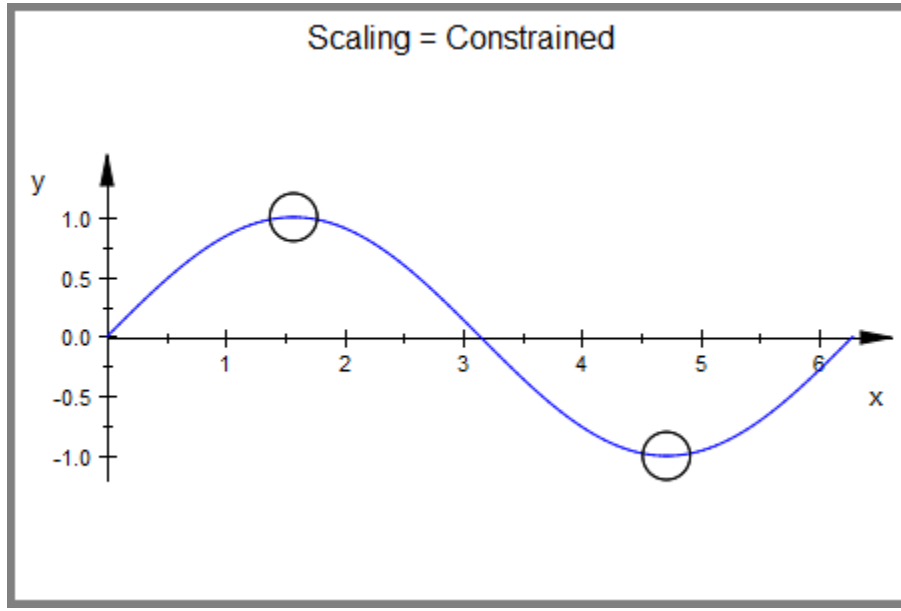
The attributes `YXRatio = r1` and `ZXRatio = r2` only have an effect in 3D with `Scaling = Unconstrained`. The graphical scene is scaled to a box with side ratios `z : y : x = r2: r1: 1`. On the screen, the bounding box of the scene looks like a box with these side ratios.

Examples

Example 1

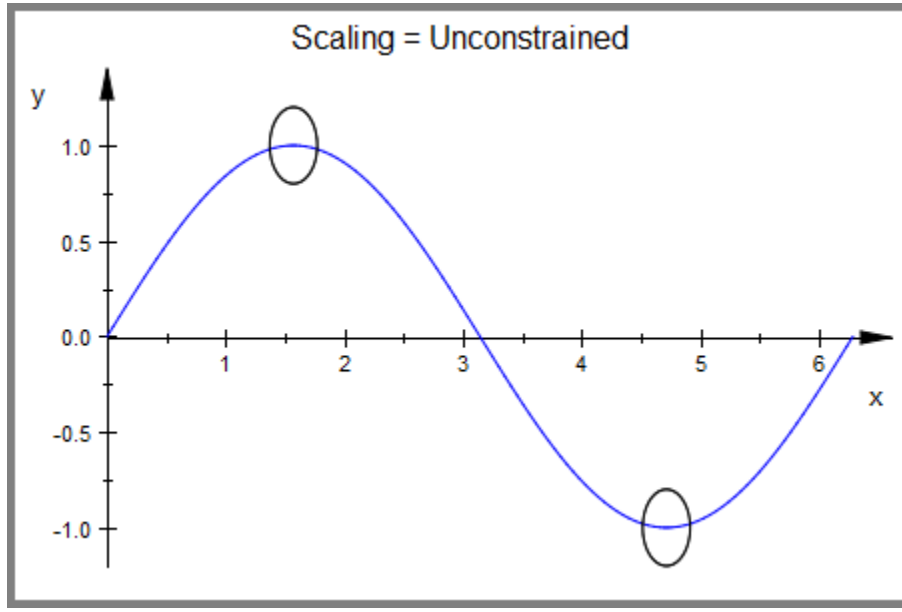
We plot a scene containing a function graph and some circles highlighting the extrema. Because the circle sends the "hint"`Scaling = Constrained`, this scaling is used for the whole scene. Consequently, the circles appear as circles:

```
plot(plot::Function2d(sin(x), x = 0 .. 2*PI),
     plot::Circle2d(0.2, [PI/2, 1], Color = RGB::Black),
     plot::Circle2d(0.2, [3*PI/2, -1], Color = RGB::Black),
     BorderWidth = 1.0*unit::mm,
     Header = "Scaling = Constrained")
```



With `Scaling = UnConstrained`, we get a better fit of the plot in the canvas. However, the circles are deformed to ellipses:

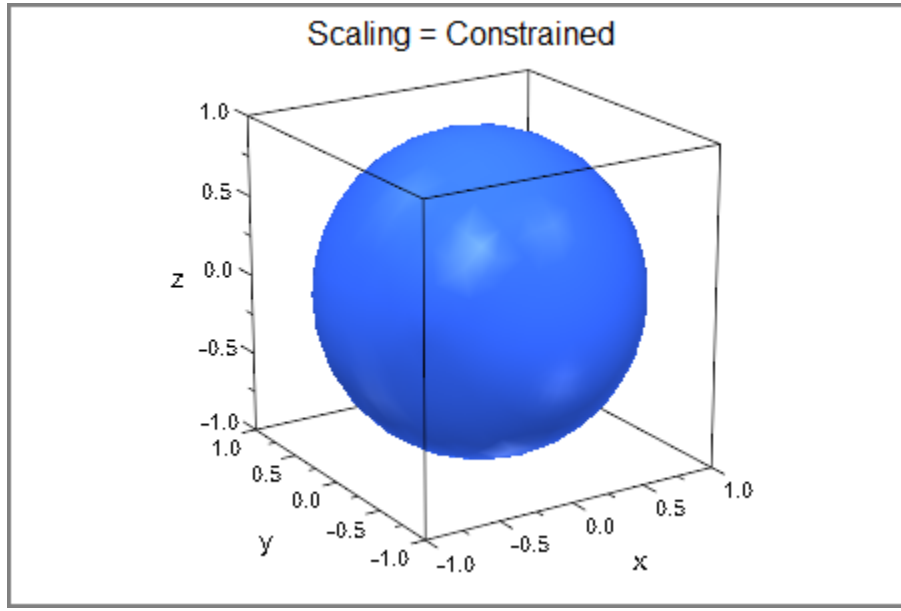
```
plot(plot::Function2d(sin(x), x = 0 .. 2*PI),
     plot::Circle2d(0.2, [PI/2, 1], Color = RGB::Black),
     plot::Circle2d(0.2, [3*PI/2, -1], Color = RGB::Black),
     Scaling = Unconstrained, BorderWidth = 1.0*unit::mm,
     Header = "Scaling = Unconstrained")
```



Example 2

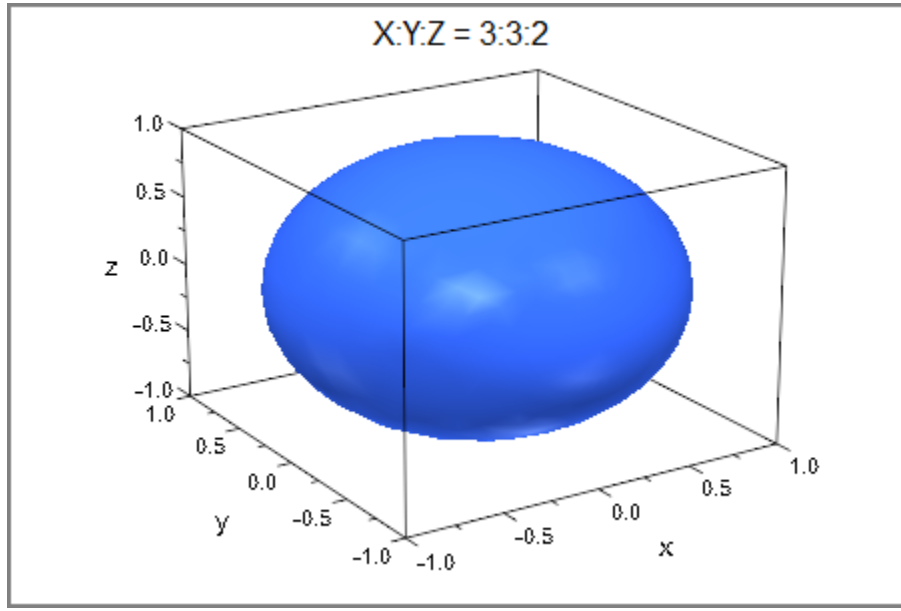
A sphere of type `plot::Sphere` sends the "hint" `Scaling` = `Constrained`. With this scaling, `YXRatio`, `ZXRatio` have no effect:

```
s := plot::Sphere(1, [0, 0, 0]):
plot(s, BorderWidth = 0.5*unit::mm,
     Header = "Scaling = Constrained",
     YXRatio = 3, ZXRatio = 10)
```



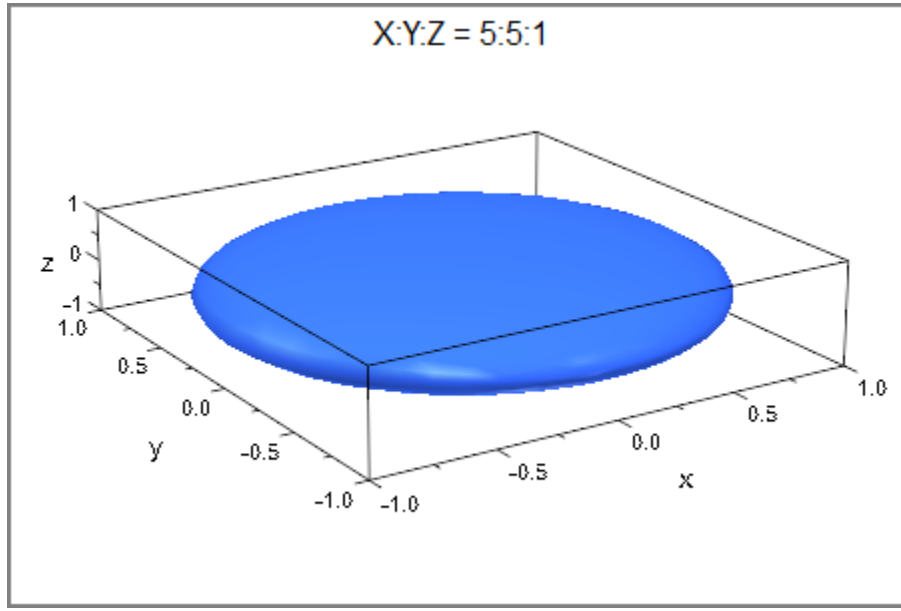
We use `Scaling = Unconstrained`. With the default values `YXRatio = 1`, `ZXRatio = 2/3`, the objects in a 3D scene are displayed like a box with side ratios `X : Y : Z = 3 : 3 : 2`:

```
plot(s, BorderWidth = 0.5*unit::mm,
     Scaling = Unconstrained, Header = "X:Y:Z = 3:3:2")
```

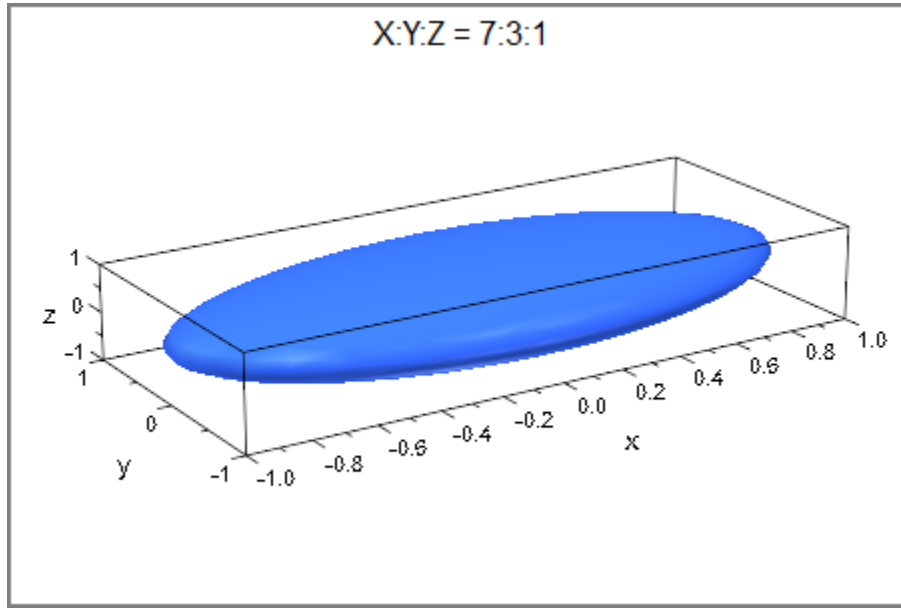


We request different scaling ratios:

```
plot(s, BorderWidth = 0.5*unit::mm, Header = "X:Y:Z = 5:5:1",
     Scaling = Unconstrained, YXRatio = 1, ZXRatio = 1/5,
     BorderWidth = 0.5*unit::mm)
```




```
plot(s, BorderWidth = 0.5*unit::mm, Header = "X:Y:Z = 7:3:1",
     Scaling = Unconstrained, YXRatio = 3/7, ZXRatio = 1/7,
     BorderWidth = 0.5*unit::mm)
```




```
delete s:
```

VerticalAsymptotesVisible, VerticalAsymptotesStyle, VerticalAsymptotesColor, VerticalAsymptotesWidth

Vertical asymptotes indicating poles

Value Summary

`VerticalAsymptotesVisible`	Inherited	`FALSE`, or `TRUE`
`VerticalAsymptotesStyle`	Inherited	`Dashed`, `Dotted`, or `Solid`
`VerticalAsymptotesColor`, `VerticalAsymptotesWidth`	Inherited	Color

Graphics Primitives

Objects	Default Values

`plot::Function2d`	`VerticalAsymptotesVisible`: `TRUE`
	
	`VerticalAsymptotesStyle`: `Dashed`
	
	`VerticalAsymptotesColor`: `RGB::Grey50`
	
	`VerticalAsymptotesWidth`: `0.2`

Description

These options control the appearance of vertical asymptotes in 2D function plots.

`plot::Function2d` and `plotfunc2d` are able to indicate poles by drawing vertical asymptotes. These asymptotes can be switched off with `VerticalAsymptotesVisible = FALSE`. Other than that, the attributes `VerticalAsymptotesStyle`,

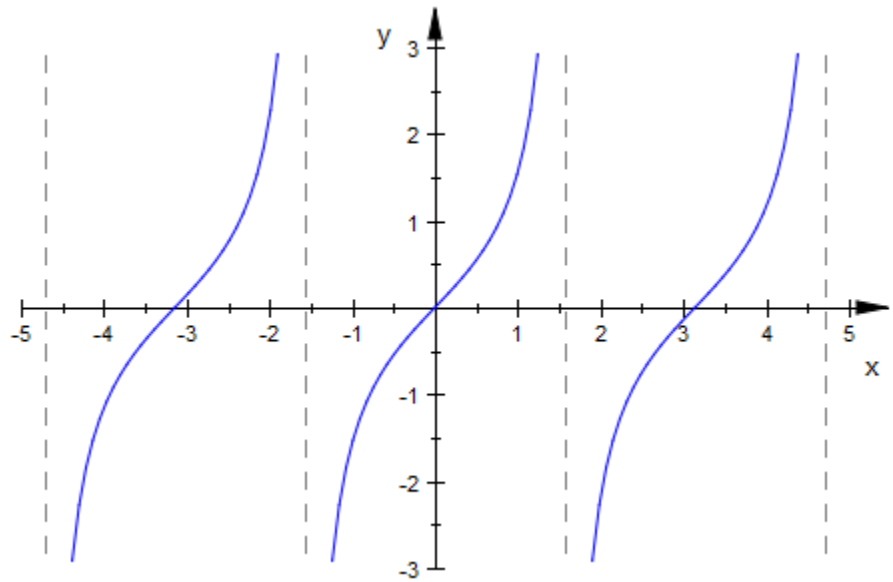
`VerticalAsymptotesColor`, and `VerticalAsymptotesWidth` influence their appearance, in the same way `LineStyle`, `LineColor`, and `LineWidth` do for other lines.

Examples

Example 1

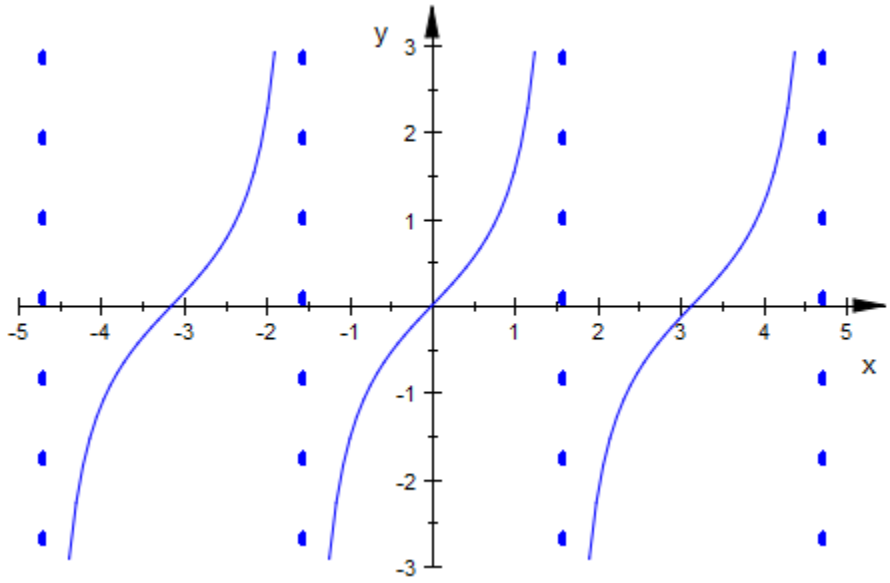
By default, vertical asymptotes are drawn as dashed, gray lines:

```
plotfunc2d(tan(x))
```



The attributes mentioned above can be used to change these settings:

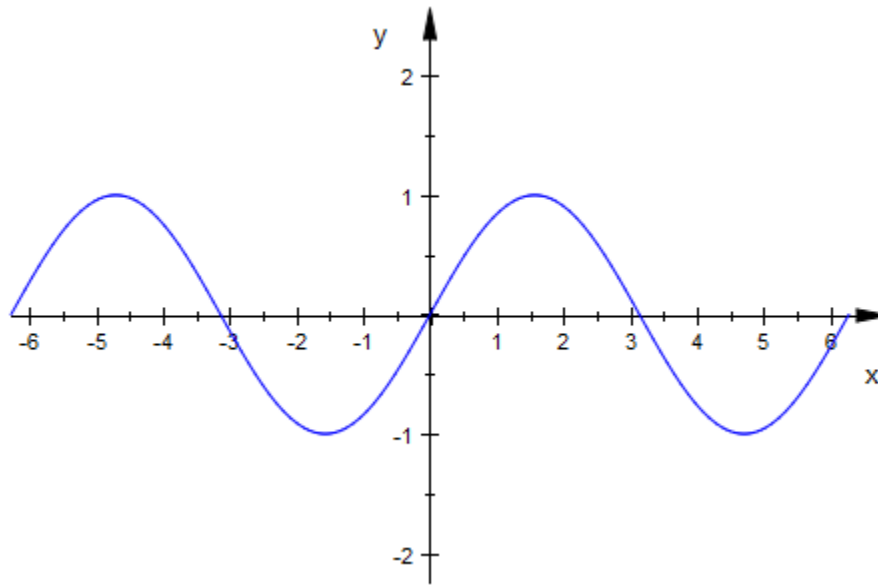
```
plotfunc2d(tan(x),
           VerticalAsymptotesColor = RGB::Blue,
           VerticalAsymptotesWidth = 1.0*unit::mm,
           VerticalAsymptotesStyle = Dotted)
```



Example 2

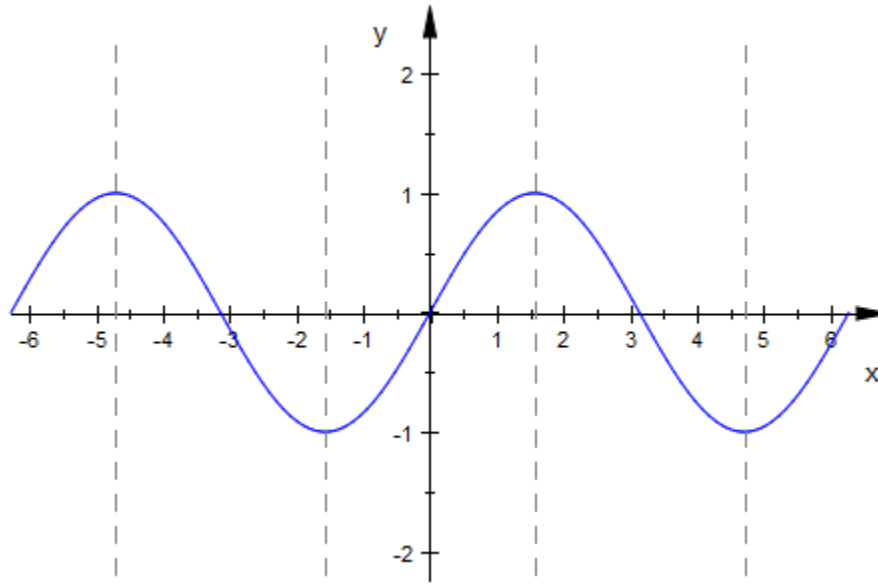
Note that vertical asymptotes obey the setting of `Visible` of their function object: No asymptotes are drawn for an invisible object.

```
t := plot::Function2d(tan(x), x = -2*PI..2*PI, Visible = FALSE):
s := plot::Function2d(sin(x), x = -2*PI..2*PI):
plot(s, t)
```



To have `t` show its asymptotes, we must set `Visible` to `TRUE`. If we only want to see the asymptotes, we can set `LinesVisible` to `FALSE`:

```
t::Visible := TRUE:
t::LinesVisible := FALSE:
plot(s, t)
```



See Also

MuPAD Functions

`LineColor` | `LineStyle` | `LineWidth`

LineColor, LineColor2

Color of lines

Value Summary

`LineColor`, `LineColor2`	Inherited	Color

Graphics Primitives

Objects	Default Values
`plot::Bars2d`, `plot::Histogram2d`, `plot::Piechart2d`	`LineColor`: `RGB::Black`

`plot::Cylindrical`, `plot::Dodecahedron`, `plot::Function3d`, `plot::Hexahedron`, `plot::Icosahedron`, `plot::Matrixplot`, `plot::Octahedron`, `plot::Prism`, `plot::Pyramid`, `plot::Spherical`, `plot::Surface`, `plot::SurfaceSet`, `plot::SurfaceSTL`, `plot::Sweep`, `plot::Tetrahedron`, `plot::Tube`, `plot::XRotate`, `plot::ZRotate`	`LineColor`: `RGB::Black.[0.25]`
	
	`LineColor2`: `RGB::DeepPink`

`plot::Arc2d`, `plot::Arrow2d`, `plot::Arrow3d`, `plot::Circle2d`, `plot::Circle3d`, `plot::Density`, `plot::Ellipse2d`, `plot::Inequality`, `plot::Line2d`, `plot::Line3d`, `plot::Lsys`, `plot::Parallelogram2d`, `plot::Raster`, `plot::Rectangle`, `plot::Rootlocus`, `plot::Turtle`	`LineColor`: `RGB::Blue`

`plot::Arc3d`, `plot::Conformal`, `plot::Curve2d`, `plot::Curve3d`,	`LineColor`: `RGB::Blue`
	
	`LineColor2`: `RGB::DeepPink`

Objects	Default Values
plot::Ellipse3d, plot::Function2d, plot::Implicit2d, plot::Listplot, plot::Polar, plot::Polygon2d, plot::Polygon3d, plot::Sequence, plot::Sum, plot::VectorField2d, plot::VectorField3d	
plot::Bars3d, plot::Box, plot::Cone, plot::Cylinder, plot::Parallelogram3d, plot::Piechart3d, plot::Plane	LineColor: RGB::Black.[0.25]
plot::QQplot, plot::Scatterplot	LineColor: RGB::Red
plot::Iteration	LineColor: RGB::Grey50
plot::Implicit3d	LineColor: RGB::Black.[0.15] LineColor2: RGB::DeepPink
plot::Waterman	LineColor: RGB::Grey40.[0.4] LineColor2: RGB::DeepPink
plot::Streamlines2d	LineColor: RGB::Black LineColor2: RGB::DeepPink
plot::Integral	LineColor: RGB::Black LineColor2: RGB::Grey

Description

LineColor sets the color of line objects such as 2D function graphs, curves in 2D and 3D, parameter lines on surfaces etc.

LineColor2 is a secondary color used for color blends.

LineColor determines the RGB color of line objects. The RGB library provides many predefined colors such as RGB::Red etc. See the section Colors of this document for more information on colors.

For pure line objects such as lines, curves, arrows, 2D function graphs etc., the line color can also be set by the attribute `Color`.

For surface objects such as 3D function graphs, surfaces etc., however, the attribute `Color` sets the `FillColor`. If you wish to change the color of the parameter lines on a surface, you have to use `LineColor`.

The RGB color set by `LineColor` cannot be animated. However, setting `LineColorType = Functional`, you can define a `LineColorFunction` that overrides the color set by `LineColor`. The line color function accepts an animation parameter, thus allowing to implement animated coloring of lines. See the help page of `LineColorFunction` for further details.

When the attribute `LineColorType` is set to one of the values `Dichromatic` or `Rainbow`, many line objects react to a secondary color set by the attribute `LineColor2`.

A gradient between the colors defined by `LineColor` and `LineColor2` is created.

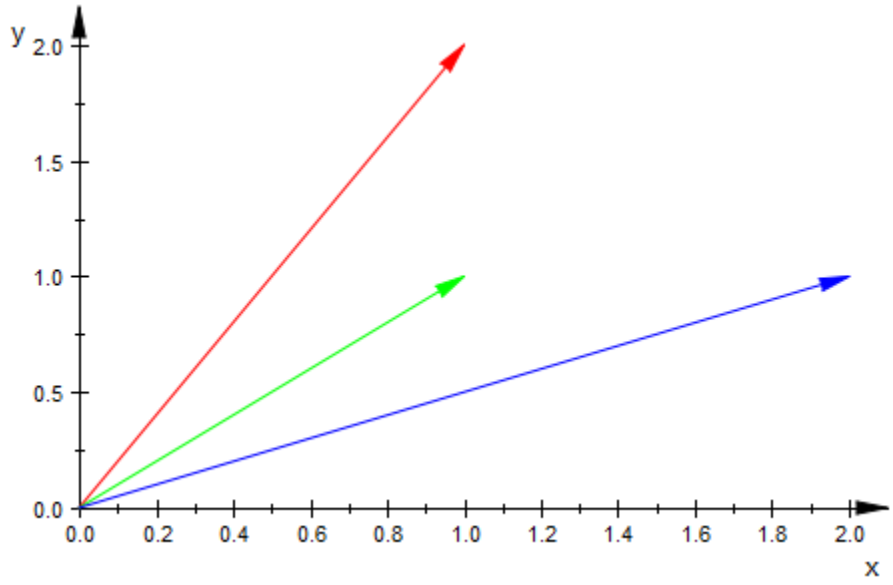
The color of the coordinate axes is set by the attribute `AxesLineColor`.

Examples

Example 1

We draw arrows of different colors:

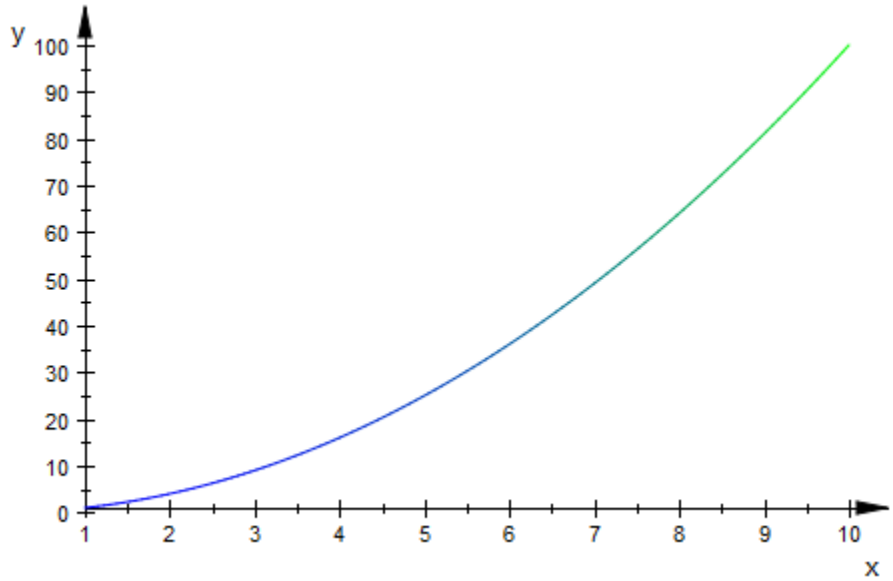
```
plot(plot::Arrow2d([0, 0], [1, 2], LineColor = RGB::Red),
     plot::Arrow2d([0, 0], [1, 1], LineColor = RGB::Green),
     plot::Arrow2d([0, 0], [2, 1], LineColor = RGB::Blue)):
```



Example 2

We draw a parabola with a gradient between green and blue:

```
plot(plot::Function2d(x^2, x = 1..10,
                      LineColorType = Dichromatic,
                      LineColor = RGB::Green,
                      LineColor2 = RGB::Blue)):
```



Example 3

As with any attribute, the line color can be read and changed using the `::`-notation:

```
p := plot::Line2d([1, 2], [4, 5]):
p::LineColor := RGB::Blue
```

```
     [0.0, 0.0, 1.0]
```

```
p::LineColor
```

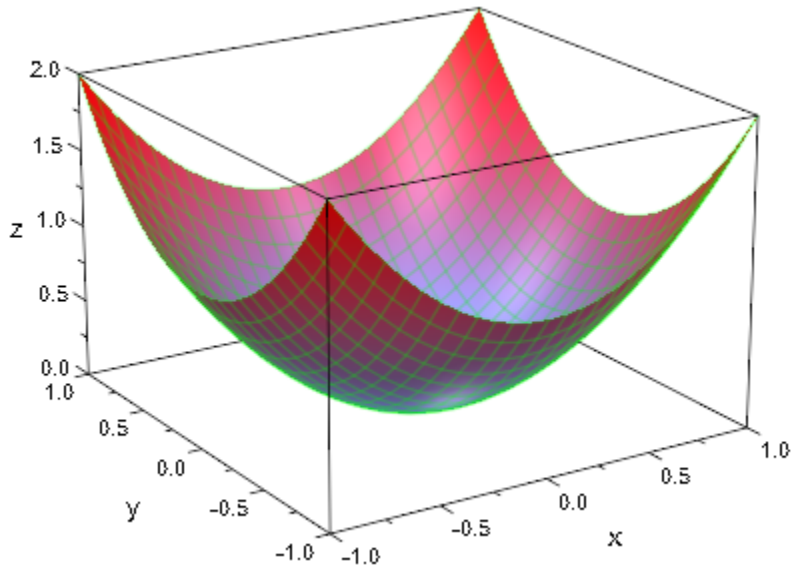
```
     [0.0, 0.0, 1.0]
```

```
delete p:
```

Example 4

For surface objects such as 3D function graphs, `LineColor` sets the color of the parameter lines on the surface. Here, a semi-transparent RGBa color is chosen that gives only a faint indication of these lines:

```
plot(plot::Function3d(x^2 + y^2, x = -1..1, y = -1 ..1,
                      LineColor = RGB::Green.[0.25])):
```



See Also

MuPAD Functions

`AxesLineColor` | `LineColorFunction` | `LineColorType` | `LineStyle` | `LinesVisible` | `LineWidth`

LineColorDirection, LineColorDirectionX, LineColorDirectionY, LineColorDirectionZ

Direction of color transitions on lines

Value Summary

`LineColorDirection`	Library wrapper for "[`LineColorDirectionX`, `LineColorDirectionY`]" (2D), "[`LineColorDirectionX`, `LineColorDirectionY`, `LineColorDirectionZ`]" (3D)	See below
`LineColorDirectionX`, `LineColorDirectionY`, `LineColorDirectionZ`	Inherited	Real number

Graphics Primitives

Objects	Default Values
`plot::Arc3d`, `plot::Circle3d`, `plot::Curve3d`, `plot::Cylindrical`, `plot::Dodecahedron`, `plot::Ellipse3d`, `plot::Function3d`, `plot::Hexahedron`, `plot::Icosahedron`, `plot::Implicit3d`, `plot::Matrixplot`, `plot::Octahedron`, `plot::Polygon3d`, `plot::Prism`, `plot::Pyramid`, `plot::Spherical`, `plot::Surface`, `plot::SurfaceSet`, `plot::SurfaceSTL`, `plot::Sweep`, `plot::Tetrahedron`, `plot::Tube`, `plot::VectorField3d`, `plot::XRotate`, `plot::ZRotate`	`LineColorDirection`: [0, 0, 1] `LineColorDirectionX`, `LineColorDirectionY`: 0 `LineColorDirectionZ`: 1

Objects	Default Values
plot::Arrow2d, plot::Circle2d, plot::Conformal, plot::Curve2d, plot::Ellipse2d, plot::Function2d, plot::Implicit2d, plot::Polar, plot::Polygon2d, plot::Rectangle, plot::Sum, plot::VectorField2d	LineColorDirection: [0, 1] LineColorDirectionX: 0 LineColorDirectionY: 1
plot::Arrow3d, plot::Box, plot::Cone	LineColorDirection: [0, 0, 1] LineColorDirectionX, LineColorDirectionY: 0
plot::Listplot	LineColorDirection: [0, 1] LineColorDirectionX: 0 LineColorDirectionY, LineColorDirectionZ: 1
plot::Waterman	LineColorDirection: [0, 1, 1] LineColorDirectionX: 0 LineColorDirectionY, LineColorDirectionZ: 1

Description

LineColorDirection determines the direction in which the color transitions for LineColorType = Dichromatic etc. take place.

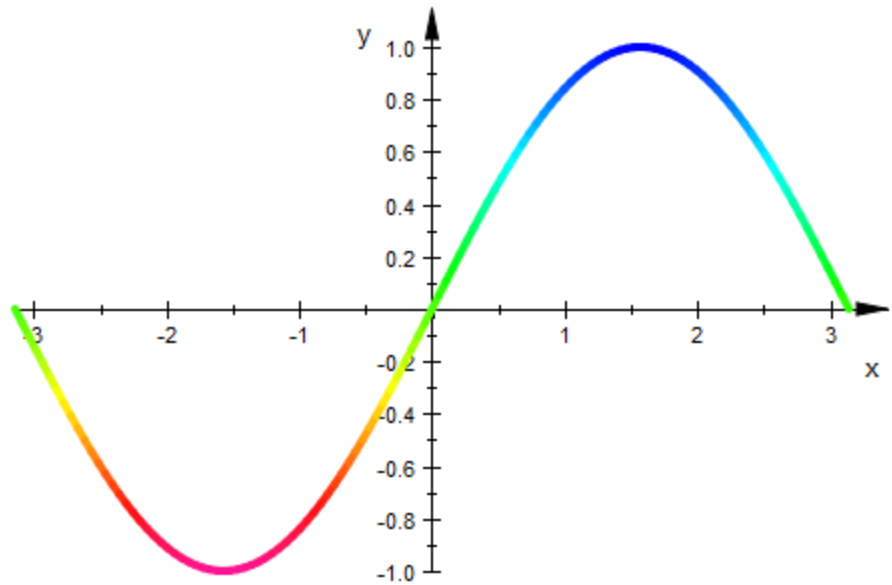
When setting LineColorType to some other value than Flat or Functional, MuPAD produces a “height-coloring.” By default, this color method actually uses the height of a point. Using LineColorDirection, the axis along which the color method should be applied can be changed.

Examples

Example 1

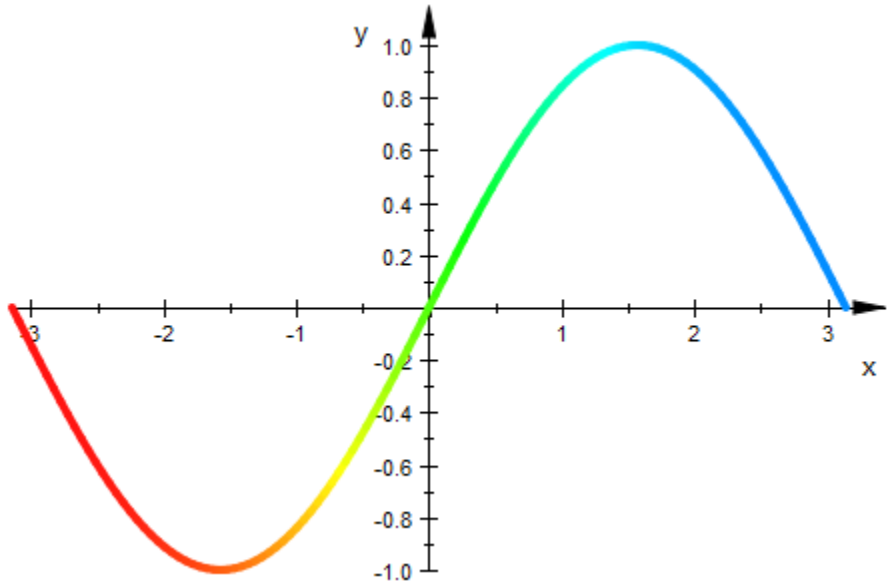
By default, MuPAD uses height coloring along the *y* axis for 2D objects:

```
f := plot::Function2d(sin(x), x=-PI..PI,
                      LineWidth = 1, LineColorType = Rainbow):
plot(f)
```



By changing `LineColorDirection`, this direction can be set to any angle. Note that `LineColorDirection` is an inherited attribute and may therefore be set at "top level" in the plot call:

```
plot(f, LineColorDirection = [1, 1])
```



See Also

MuPAD Functions

`FillColorDirection` | `LineColor` | `LineColor2` | `LineColorType`

LineColorType

Line coloring types

Value Summary

Inherited	`Dichromatic`, `Flat`, `Functional`, `Monochrome`, or `Rainbow`

Graphics Primitives

Objects	LineColorType Default Values
`plot::Arc3d`, `plot::Conformal`, `plot::Curve2d`, `plot::Curve3d`, `plot::Cylindrical`, `plot::Dodecahedron`, `plot::Ellipse3d`, `plot::Function2d`, `plot::Function3d`, `plot::Hexahedron`, `plot::Icosahedron`, `plot::Implicit2d`, `plot::Implicit3d`, `plot::Integral`, `plot::Listplot`, `plot::Matrixplot`, `plot::Octahedron`, `plot::Polar`, `plot::Polygon2d`, `plot::Polygon3d`, `plot::Prism`, `plot::Pyramid`, `plot::Rootlocus`, `plot::Sequence`, `plot::Spherical`, `plot::Streamlines2d`, `plot::Sum`, `plot::Surface`, `plot::SurfaceSet`, `plot::SurfaceSTL`, `plot::Sweep`, `plot::Tetrahedron`, `plot::Tube`, `plot::VectorField2d`, `plot::VectorField3d`, `plot::Waterman`, `plot::XRotate`, `plot::ZRotate`	`Flat`

Description

`LineColorType` controls the type of line coloring used. With the exception of `Flat` and `Functional`, the coloring schemes depend on the height, i.e., the *z* or *y* value (in 2D or 3D, respectively) of points on the line, relative to the extension of the viewing box.

By default, lines are drawn in the color set by the attribute `LineColor`. This is caused by the setting `LineColorType = Flat`. The other possible values for `LineColorType` mean:

- `Dichromatic`

  The color of a point on a line depends on its height, with the lowest point using `LineColor2`, the highest one using `LineColor`, and all other points using a linear interpolation in RGB color space.

- `Flat`

  The line is drawn with `LineColor`. No blend is used.

- `Monochrome`

  The line is drawn with a blend from `LineColor` to a dimmed version of `LineColor`.

- `Rainbow`

  This setting is technically similar to `Dichromatic`, but the effect is vastly different, since interpolation takes place in HSV color space. This creates a “rainbow effect”, which mostly conforms with a physical rainbow for suitable choices of colors.

- `Functional`

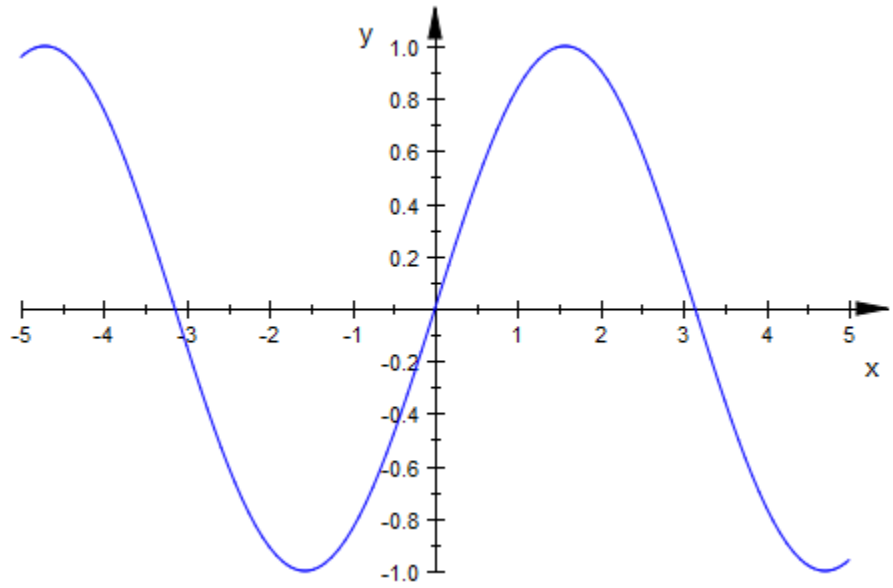
  Both `LineColor` and `LineColor2` are ignored; the color scheme is derived from `LineColorFunction`. See `?LineColorFunction` for details (which depend on the object type). If no color function is given, the object will be rendered with `LineColorType = Flat`.

Examples

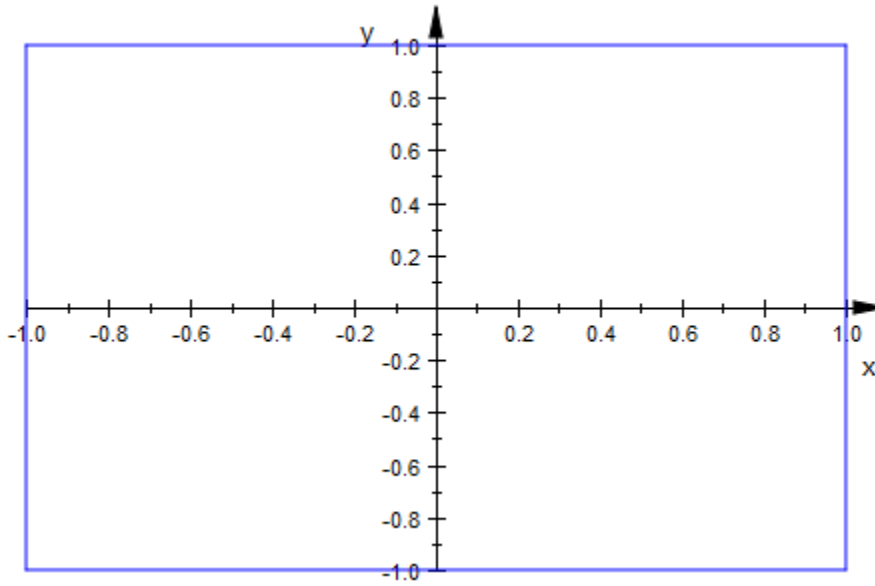
Example 1

By default, lines are drawn in one flat color:

```
plotfunc2d(sin(x))
```



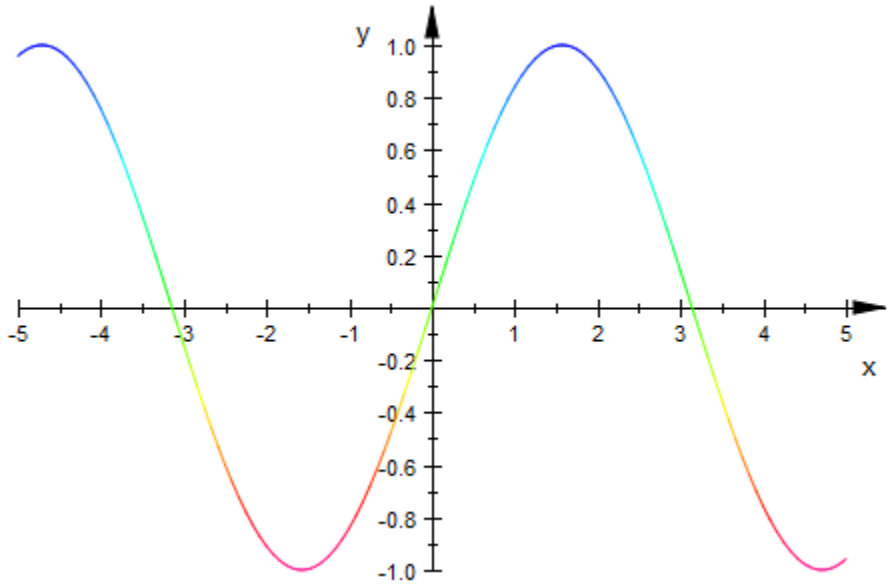

```
plot(plot::Polygon2d([[-1,-1], [1,-1], [1,1], [-1,1]],
                     Closed = TRUE, Filled = FALSE))
```



Example 2

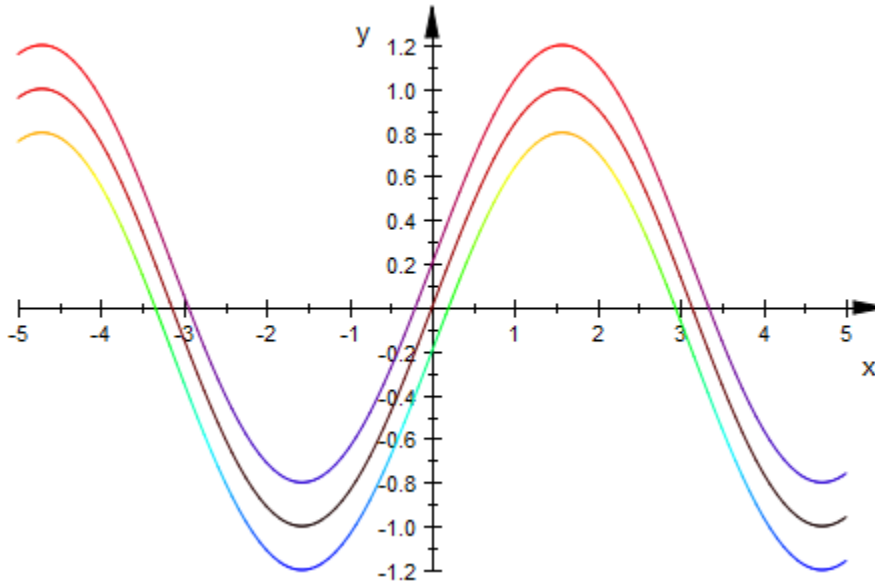
Using `LineColorType = Dichromatic`, `Monochrome`, or `Rainbow` causes a height-dependent color effect:

```
plotfunc2d(sin(x), LineColorType = Rainbow)
```



Note that height coloring depends on the height of the whole scene, not only on that of individual objects:

```
plot(
 plot::Function2d(sin(x) + 0.2, LineColorType = Dichromatic),
 plot::Function2d(sin(x) + 0.0, LineColorType = Monochrome),
 plot::Function2d(sin(x) - 0.2, LineColorType = Rainbow),
 LineColor = RGB::Red, LineColor2 = RGB::Blue
)
```



See Also

MuPAD Functions

`FillColorType` | `LineColor` | `LineColor2` | `LineColorFunction`

LineStyle

Solid, dashed or dotted lines?

Value Summary

Inherited	`Dashed`, `Dotted`, or `Solid`

Graphics Primitives

Objects	LineStyle Default Values
`plot::Arc2d`, `plot::Arc3d`, `plot::Arrow2d`, `plot::Arrow3d`, `plot::Bars2d`, `plot::Bars3d`, `plot::Box`, `plot::Boxplot`, `plot::Circle2d`, `plot::Circle3d`, `plot::Cone`, `plot::Conformal`, `plot::Curve2d`, `plot::Curve3d`, `plot::Cylinder`, `plot::Cylindrical`, `plot::Density`, `plot::Dodecahedron`, `plot::Ellipse2d`, `plot::Ellipse3d`, `plot::Function2d`, `plot::Function3d`, `plot::Hexahedron`, `plot::Histogram2d`, `plot::Icosahedron`, `plot::Implicit2d`, `plot::Implicit3d`, `plot::Inequality`, `plot::Integral`, `plot::Iteration`, `plot::Line2d`, `plot::Line3d`, `plot::Listplot`, `plot::Lsys`, `plot::Matrixplot`, `plot::Octahedron`, `plot::Ode2d`, `plot::Ode3d`, `plot::Parallelogram2d`, `plot::Parallelogram3d`, `plot::Piechart2d`, `plot::Piechart3d`,	`Solid`

Objects	LineStyle Default Values
plot::Polar, plot::Polygon2d, plot::Polygon3d, plot::Prism, plot::Pyramid, plot::QQplot, plot::Raster, plot::Rectangle, plot::Rootlocus, plot::Scatterplot, plot::Sequence, plot::Spherical, plot::Streamlines2d, plot::Sum, plot::Surface, plot::SurfaceSet, plot::SurfaceSTL, plot::Sweep, plot::Tetrahedron, plot::Turtle, plot::Waterman, plot::XRotate, plot::ZRotate	

Description

`LineStyle` controls whether lines are drawn as solid, dashed or dotted lines.

`LineStyle` sets the style of line objects such as 2D function graphs, curves in 2D and 3D, arrows, parameter lines on surfaces etc.

The available line styles are `Solid`, `Dashed`, or `Dotted`.

This attribute may be useful to distinguish different curves.

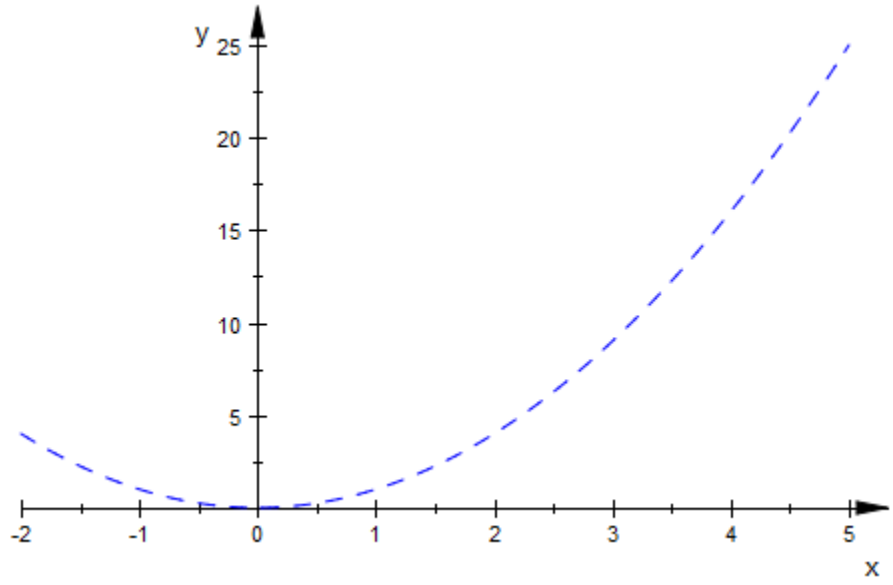
`LineStyle` does not have an effect on the line style of axes and coordinate grid lines. Axes are always displayed as solid lines. The style of the coordinate grid can be set by the attribute `GridLineStyle`.

Examples

Example 1

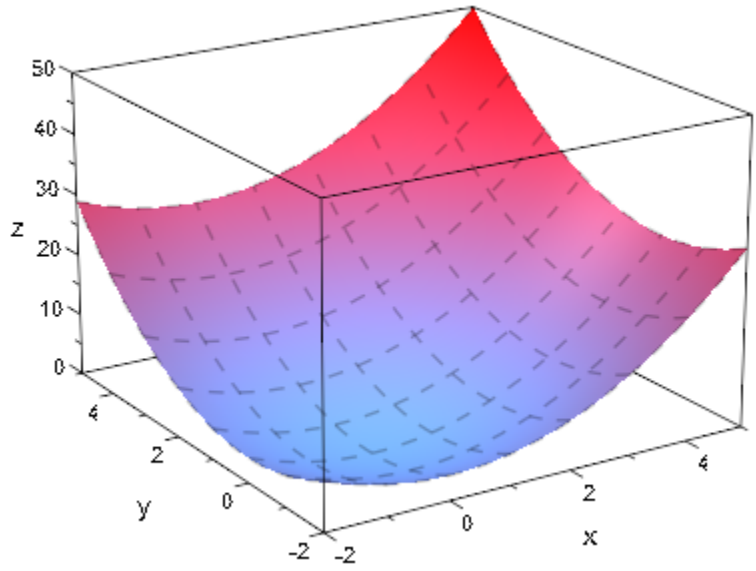
We draw a dashed parabola:

```
plot(plot::Function2d(x^2, x = -2..5, LineStyle = Dashed))
```



We draw a paraboloid with dashed coordinate lines:

```
plot(plot::Function3d(x^2 + y^2, x = -2..5, y = -2..5,
                      Mesh = [8, 8], Submesh = [3, 3],
                      LineStyle = Dashed))
```



See Also

MuPAD Functions

`GridLineStyle` | `LineColor` | `LineColorType` | `LinesVisible` | `LineWidth`

LinesVisible, ULinesVisible, VLinesVisible, XLinesVisible, YLinesVisible

Visibility of lines

Value Summary

`LinesVisible`, `ULinesVisible`, `VLinesVisible`, `XLinesVisible`, `YLinesVisible`	Inherited	FALSE, or TRUE

Graphics Primitives

Objects	Default Values
`plot::Arc2d`, `plot::Arc3d`, `plot::Bars2d`, `plot::Bars3d`, `plot::Box`, `plot::Boxplot`, `plot::Circle2d`, `plot::Circle3d`, `plot::Cone`, `plot::Conformal`, `plot::Curve2d`, `plot::Curve3d`, `plot::Cylinder`, `plot::Dodecahedron`, `plot::Ellipse2d`, `plot::Ellipse3d`, `plot::Function2d`, `plot::Hexahedron`, `plot::Histogram2d`, `plot::Icosahedron`, `plot::Implicit2d`, `plot::Integral`, `plot::Listplot`, `plot::Lsys`, `plot::Octahedron`, `plot::Ode2d`, `plot::Ode3d`, `plot::Parallelogram2d`, `plot::Parallelogram3d`, `plot::Piechart2d`, `plot::Piechart3d`, `plot::Plane`, `plot::Polar`, `plot::Polygon2d`, `plot::Polygon3d`, `plot::Prism`, `plot::Pyramid`,	`LinesVisible`: TRUE

Objects	Default Values
plot::QQplot, plot::Rectangle, plot::Rootlocus, plot::Scatterplot, plot::Sum, plot::Tetrahedron, plot::Turtle, plot::Waterman	
plot::Density, plot::Inequality, plot::Raster, plot::Sequence	LinesVisible: FALSE
plot::Function3d	LinesVisible, XLinesVisible, YLinesVisible: TRUE
plot::Cylindrical, plot::Matrixplot, plot::Spherical, plot::Surface, plot::Sweep, plot::Tube, plot::XRotate, plot::ZRotate	ULinesVisible, VLinesVisible, XLinesVisible, YLinesVisible: TRUE

Description

LinesVisible = TRUE versus LinesVisible = FALSE governs the visibility of line objects.

ULinesVisible, VLinesVisible governs the visibility of coordinate lines on parametrized surfaces in 3D.

XLinesVisible, YLinesVisible governs the visibility of coordinate lines on 3D function graphs and matrix plots.

For most object types, LinesVisible determines whether lines are drawn. This includes the lines making up 2D function plots, curves, polygons, etc. as well as the circumference of (filled) circles, the edges of 2D rectangles and 3D boxes etc.

The exception are surface objects that exhibit parameter lines in two directions, such as 3D function plots, parameterized surfaces, tube plots etc. Depending on whether they react to XMesh, YMesh or to UMesh, VMesh, the parameter lines on the surfaces can be made visible or invisible with the attributes XLinesVisible, YLinesVisible or ULinesVisible, VLinesVisible, respectively.

Note that setting LinesVisible = FALSE for a 2D function plot without setting PointsVisible = TRUE will render the function invisible. (In case of singular functions, the vertical asymptotes may remain visible, though).

The same holds true for plots involving filled areas: switching off the lines and the filling makes such objects invisible.

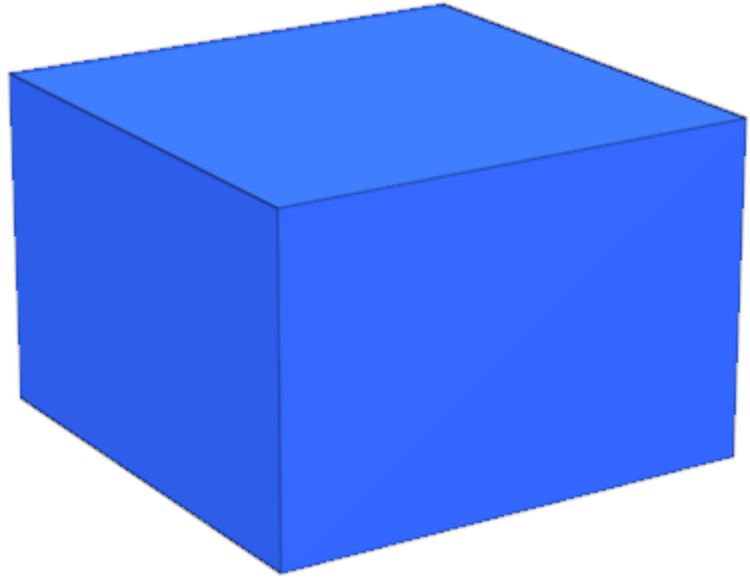
`LinesVisible` etc. do not have an effect on coordinate axes and coordinate grid lines. Use the attributes `AxesVisible` and `GridVisible` to control the visibility of axes and coordinate grid.

Examples

Example 1

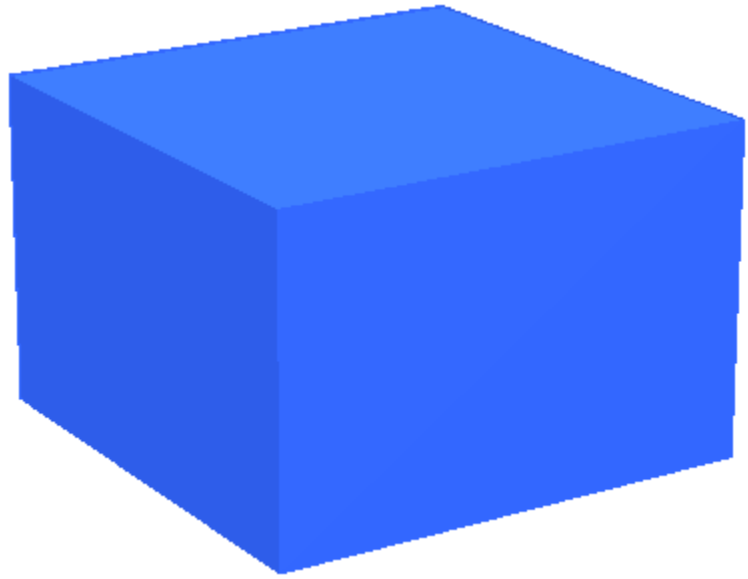
By default, the lines of a box are visible:

```
plot(plot::Box(1..4, 2..5, 3..6), Axes = None)
```



We set `LinesVisible` = `FALSE` to switch them off:

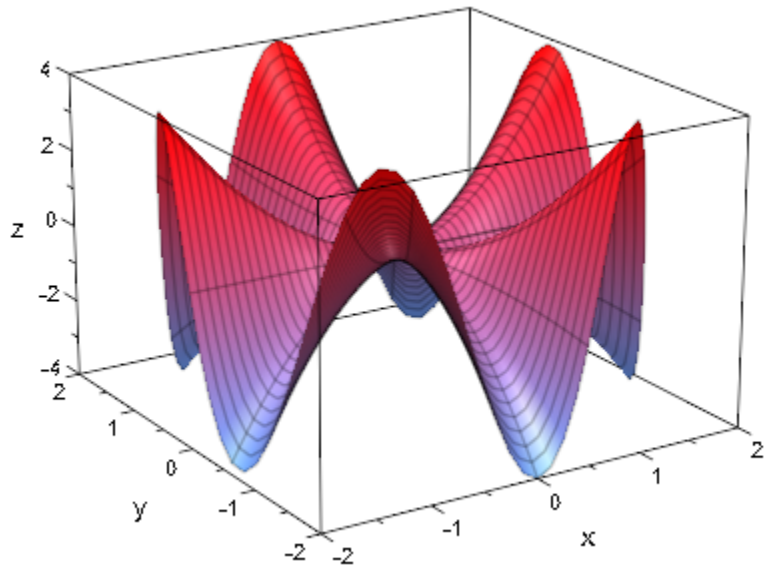
```
plot(plot::Box(1..4, 2..5, 3..6, LinesVisible = FALSE),
     Axes = None)
```



Example 2

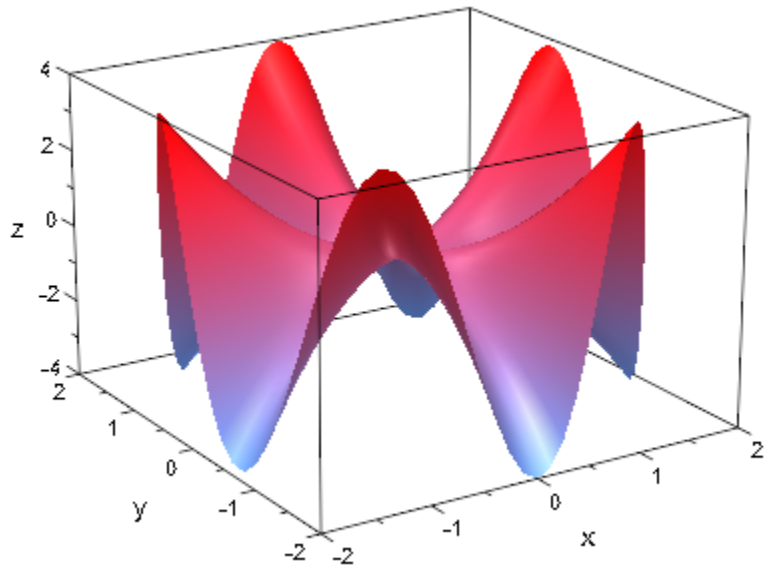
By default, parameter lines are drawn on a parametrized surface:

```
plot(plot::Surface([u*cos(v), u*sin(v), u^2*sin(5*v)],
                   u = 0..2, v = 0..2*PI, VSubmesh = 3)):
```



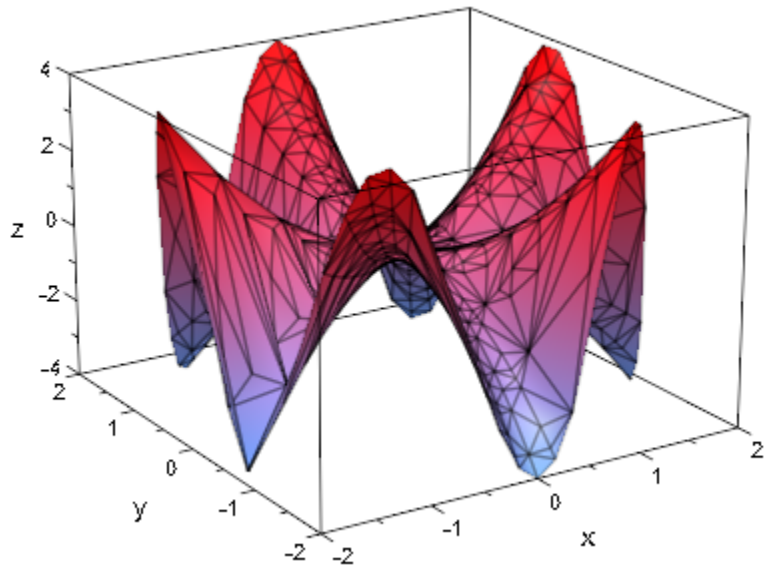
You can switch these lines off interactively, or, as we do here, by setting `ULinesVisible` and `VLinesVisible` to `FALSE` in the plot command:

```
plot(plot::Surface([u*cos(v), u*sin(v), u^2*sin(5*v)],
                   u = 0..2, v = 0..2*PI, VSubmesh = 3,
                   ULinesVisible = FALSE,
                   VLinesVisible = FALSE)):
```



When the surface is created with an adaptive mesh, we can make the irregular adaptive mesh visible by setting `MeshVisible` = `TRUE`:

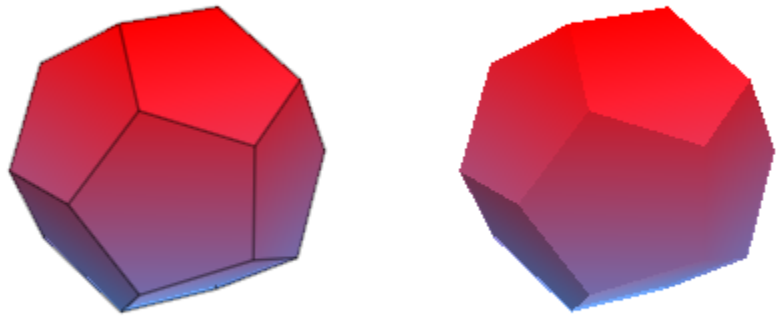
```
plot(plot::Surface([u*cos(v), u*sin(v), u^2*sin(5*v)],
                   u = 0..2, v = 0..2*PI,
                   UMesh =  5, VMesh = 10,
                   ULinesVisible = FALSE,
                   VLinesVisible = FALSE,
                   AdaptiveMesh = 3,
                   MeshVisible = TRUE)):
```



Example 3

We plot a dodecahedron with and without the border lines of its faces:

```
plot(plot::Scene3d(plot::Dodecahedron(LinesVisible = TRUE)),
     plot::Scene3d(plot::Dodecahedron(LinesVisible = FALSE)),
     Layout = Horizontal, Axes = None):
```



See Also

MuPAD Functions

`AxesVisible` | `GridVisible` | `LineColor` | `LineColorType` | `LineStyle` | `LineWidth`

LineWidth

Width of lines

Value Summary

Inherited	Positive output size

Graphics Primitives

Objects	LineWidth Default Values
`plot::Arc2d`, `plot::Arc3d`, `plot::Arrow2d`, `plot::Arrow3d`, `plot::Bars2d`, `plot::Bars3d`, `plot::Box`, `plot::Boxplot`, `plot::Circle2d`, `plot::Circle3d`, `plot::Cone`, `plot::Conformal`, `plot::Curve2d`, `plot::Curve3d`, `plot::Cylinder`, `plot::Cylindrical`, `plot::Density`, `plot::Dodecahedron`, `plot::Ellipse2d`, `plot::Ellipse3d`, `plot::Function2d`, `plot::Function3d`, `plot::Hexahedron`, `plot::Histogram2d`, `plot::Icosahedron`, `plot::Implicit2d`, `plot::Implicit3d`, `plot::Inequality`, `plot::Integral`, `plot::Iteration`, `plot::Line2d`, `plot::Line3d`, `plot::Listplot`, `plot::Lsys`, `plot::Matrixplot`, `plot::Octahedron`, `plot::Ode2d`, `plot::Ode3d`, `plot::Parallelogram2d`, `plot::Parallelogram3d`, `plot::Piechart2d`, `plot::Piechart3d`, `plot::Polar`, `plot::Polygon2d`, `plot::Polygon3d`, `plot::Prism`, `plot::Pyramid`, `plot::QQplot`,	`0.35`

Objects	LineWidth Default Values
plot::Raster, plot::Rectangle, plot::Rootlocus, plot::Scatterplot, plot::Sequence, plot::Spherical, plot::Sum, plot::Surface, plot::SurfaceSet, plot::SurfaceSTL, plot::Sweep, plot::Tetrahedron, plot::Tube, plot::Turtle, plot::VectorField2d, plot::XRotate, plot::ZRotate	
plot::VectorField3d	0.1
plot::Waterman	0.25
plot::Streamlines2d	0.35*unit::mm

Description

LineWidth sets the width of line objects such as 2D function graphs, curves in 2D and 3D, arrows, parameter lines on surfaces etc.

The value should be specified as an absolute physical length including a length unit such as LineWidth = 1.5*unit::mm. Numbers without a physical unit give the size in mm.

Note that the graphics cannot always react to small changes of the line width because of the discretization into pixels.

One cannot make lines invisible by setting their width to 0. Use LinesVisible = FALSE instead.

LineWidth does not have an effect on the line width of axes and coordinate grid lines. Use the attributes AxesLineWidth and GridLineWidth to manipulate axes and coordinate grid, respectively.

Examples

Example 1

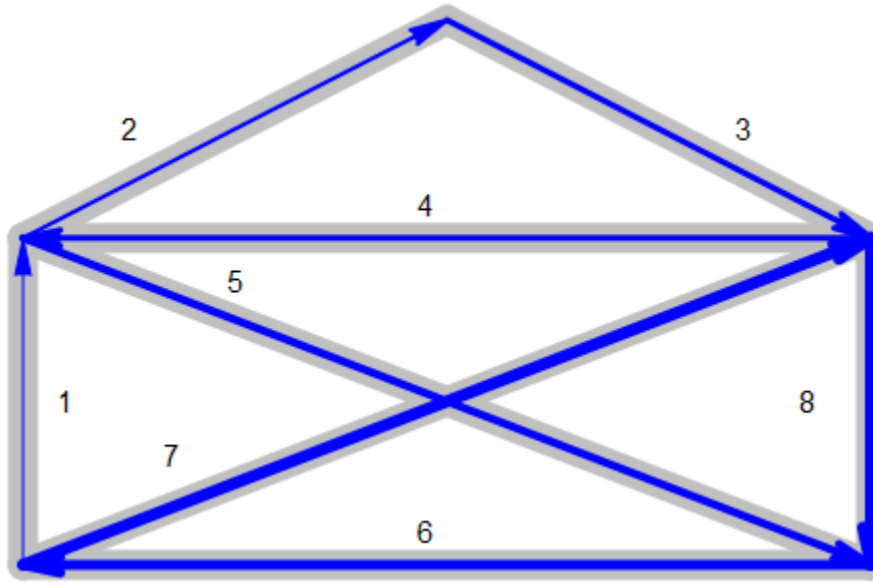
We draw a house with thick walls:

```
plot(plot::Polygon2d(
     [[0, 0], [0, 3], [2, 5], [4, 3], [0, 3],
      [4, 0], [0, 0], [4, 3], [4, 0] ],
     LineWidth = 4*unit::mm, Color = RGB::Grey),
     Axes = None):
```

The building instructions are added by arrows. The drawing order is indicated by the titles of the arrows and their increasing line width:

```
plot(plot::Polygon2d(
     [[0, 0], [0, 3], [2, 5], [4, 3], [0, 3],
      [4, 0], [0, 0], [4, 3], [4, 0] ],
     LineWidth = 4*unit::mm, Color = RGB::Grey),
     plot::Arrow2d([0, 0], [0, 3], LineWidth = 0.3*unit::mm,
                   Title = "1", TitlePosition = [0.2, 1.4]),
     plot::Arrow2d([0, 3], [2, 5], LineWidth = 0.5*unit::mm,
                   Title = "2", TitlePosition = [0.5, 3.9]),
     plot::Arrow2d([2, 5], [4, 3], LineWidth = 0.7*unit::mm,
                   Title = "3", TitlePosition = [3.4, 3.9]),
     plot::Arrow2d([4, 3], [0, 3], LineWidth = 0.9*unit::mm,
                   Title = "4", TitlePosition = [1.9, 3.2]),
     plot::Arrow2d([0, 3], [4, 0], LineWidth = 1.1*unit::mm,
```

```
                Title = "5", TitlePosition = [1.0, 2.5]),
 plot::Arrow2d([4, 0], [0, 0], LineWidth = 1.3*unit::mm,
                Title = "6", TitlePosition = [1.9, 0.2]),
 plot::Arrow2d([0, 0], [4, 3], LineWidth = 1.5*unit::mm,
                Title = "7", TitlePosition = [0.7, 0.9]),
 plot::Arrow2d([4, 3], [4, 0], LineWidth = 1.7*unit::mm,
                Title = "8", TitlePosition = [3.7, 1.4]),
 Axes = None,
 TipLength = 5*unit::mm
):
```



See Also

MuPAD Functions

`AxesLineWidth` | `GridLineWidth` | `LineColor` | `LineColorType` | `LineStyle` | `LinesVisible`

MeshVisible

Visibility of irregular mesh lines in 3D

Value Summary

Inherited	FALSE, or TRUE

Graphics Primitives

Objects	MeshVisible Default Values
plot::Cylindrical, plot::Function3d, plot::Implicit3d, plot::Spherical, plot::Surface, plot::SurfaceSet, plot::SurfaceSTL, plot::XRotate, plot::ZRotate	FALSE

Description

MeshVisible = TRUE versus MeshVisible = FALSE controls the visibility of the irregular mesh defining surfaces that are either computed by an adaptive algorithm or are given explicitly by a triangulation.

3D function plots and parametrized surfaces are usually defined over a regular mesh. When setting AdaptiveMesh = n with n > 0, an irregular adaptive mesh is created that refines the graphical object automatically in critical regions.

While visibility of the regular mesh is controlled by the attributes XLinesVisible, YLinesVisible or ULinesVisible, VLinesVisible, respectively, the visibility of the adaptively refined mesh is set MeshVisible.

Also special surfaces created from a given triangulation such as plot::SurfaceSet and plot::SurfaceSTL allow to make the triangulation visible by setting MeshVisible = TRUE.

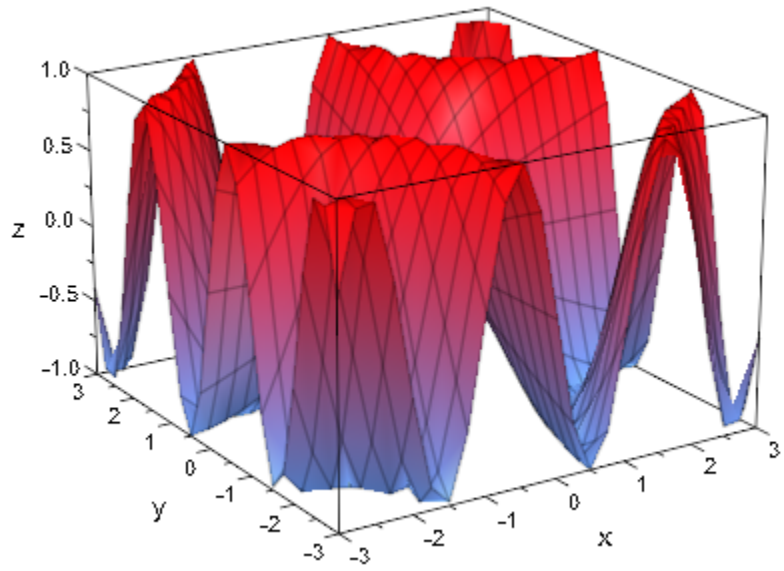
The irregular mesh lines switched on by `MeshVisible` = `TRUE` react to the attributes `LineColor`, `LineStyle`, and `LineWidth`.

Examples

Example 1

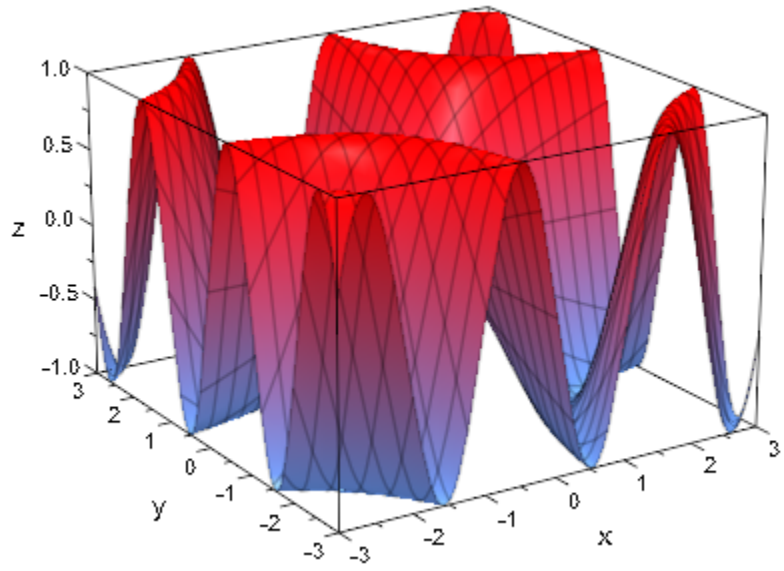
We create a 3D function plot:

```
plot(plot::Function3d(sin(x*y), x = -3..3, y = -3..3))
```



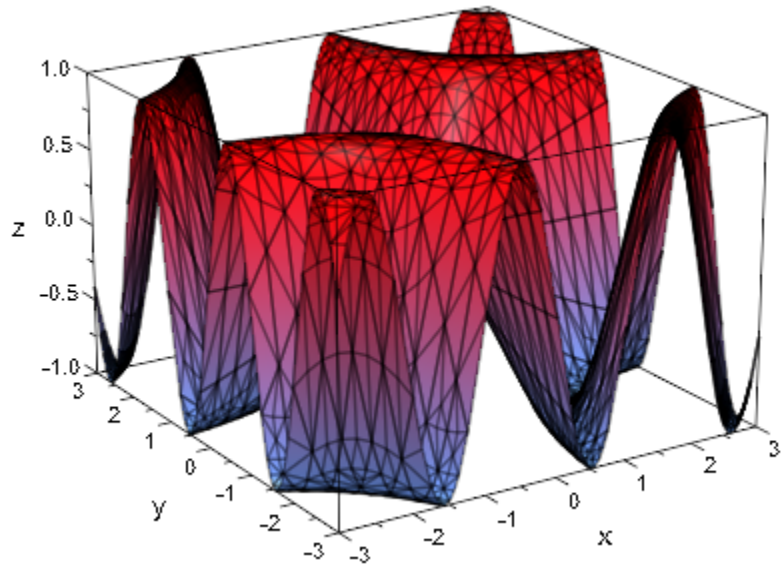
By default, only the regular mesh is visible, even if adaptive evaluation is used:

```
plot(plot::Function3d(sin(x*y), x = -3..3, y = -3..3,
                      AdaptiveMesh = 2))
```



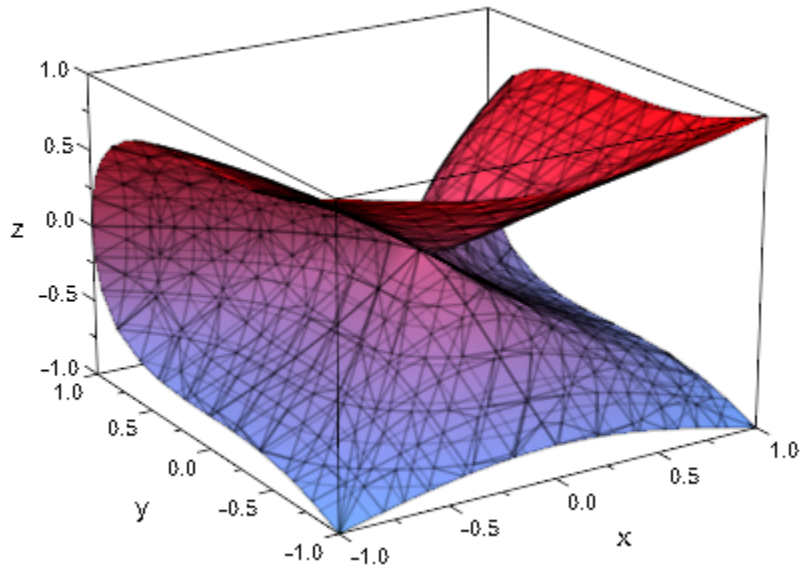
The irregular mesh is made visible when using `MeshVisible = TRUE`:

```
plot(plot::Function3d(sin(x*y), x = -3..3, y = -3..3,
                      AdaptiveMesh = 2, MeshVisible = TRUE))
```

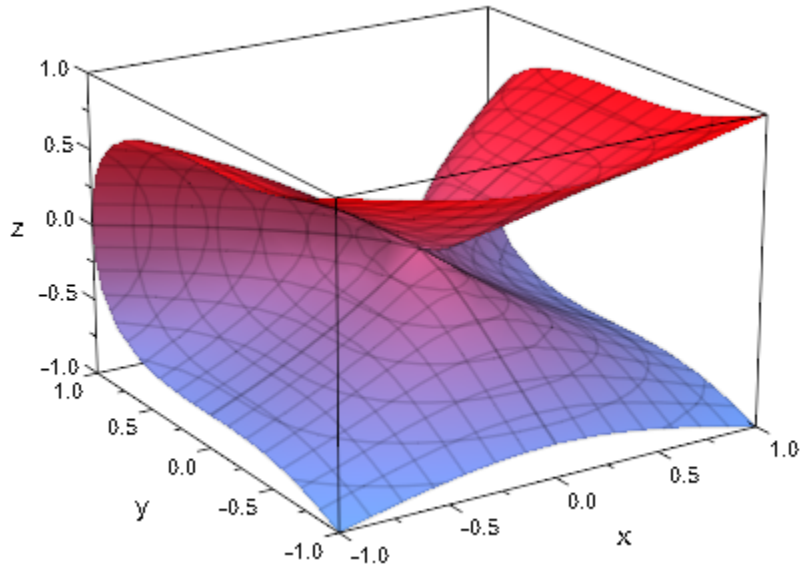


A 3D plot of an implicit surface does not have regular mesh lines. We plot such a surface with and without the irregular mesh:

```
plot(plot::Implicit3d(z^4 + z^2 - x^2 + y^3,
                      x = -1..1, y = -1..1, z = -1..1,
                      MeshVisible = TRUE))
```




```
plot(plot::Implicit3d(z^4 + z^2 - x^2 + y^3,
                      x = -1..1, y = -1..1, z = -1..1,
                      MeshVisible = FALSE))
```



See Also

MuPAD Functions

`AdaptiveMesh` | `LineColor` | `LineStyle` | `LineWidth`

XContours, YContours, ZContours

Contour lines at constant x values

Value Summary

`XContours`, `YContours`, `ZContours`	Optional	List of arithmetical expressions

Graphics Primitives

Objects	Default Values
`plot::Implicit3d`	`XContours`, `YContours`, `ZContours`: [`Automatic`, `15`]
`plot::Cylindrical`, `plot::Function3d`, `plot::Spherical`, `plot::Surface`, `plot::XRotate`, `plot::ZRotate`	`XContours`, `YContours`, `ZContours`: []

Description

`XContours`, `YContours`, and `ZContours` cause contour lines on surface objects at constant *x*, *y*, or *z*-values, respectively.

By setting these attributes, many surface objects (such as implicit surfaces, function objects etc.) can be instructed to display contour lines.

By setting `ZContours = [`z_1`, `z_2`, …]`, contour lines can be requested at specific places. This is demonstrated in “Example 1” on page 24-1982.

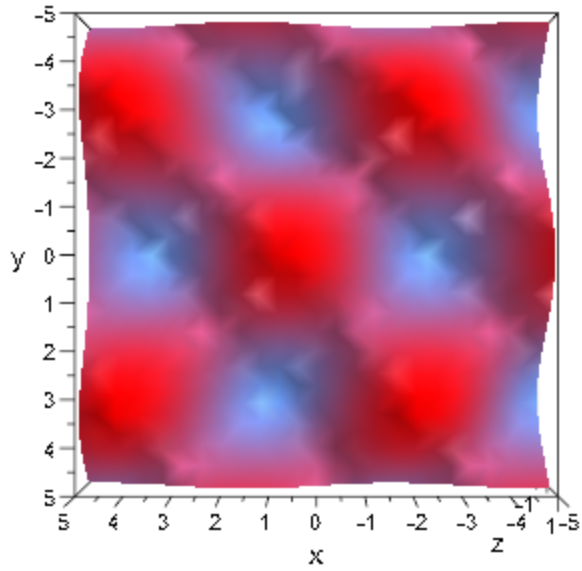
`ZContours = [Automatic, n]` causes *n* contour lines to be evenly spaced along the range of *z* values of the object. Cf. “Example 2” on page 24-1983.

Examples

Example 1

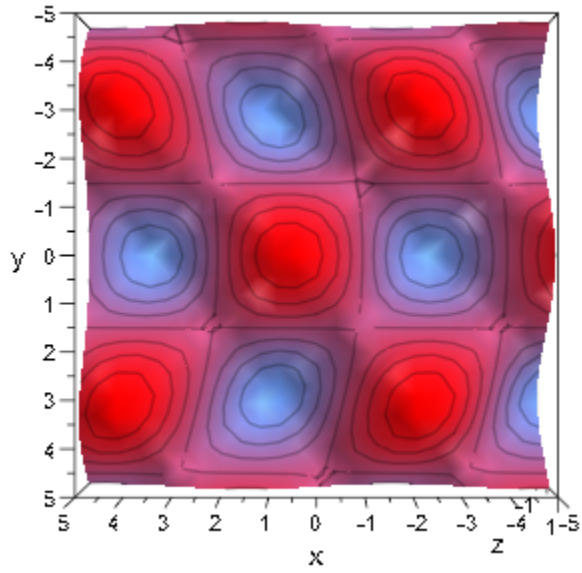
A function plot by default uses height coloring and mesh lines to improve the visual display. With mesh lines disabled, height coloring is often still sufficient:

```
plotfunc3d(sin(x+cos(0.3*y))*cos(y),
           XLinesVisible=FALSE, YLinesVisible=FALSE,
           CameraDirection=[0,0.01,1])
```



To get a better depth impression, it would help in this example to add contour lines:

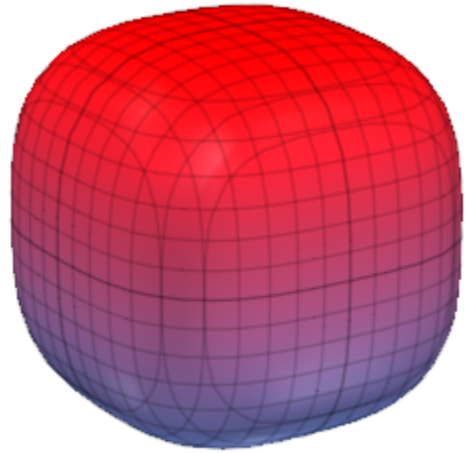
```
plotfunc3d(sin(x+cos(0.3*y))*cos(y),
           ZContours=[$ -1..1 step 0.25],
           XLinesVisible=FALSE, YLinesVisible=FALSE,
           CameraDirection=[0,0.01,1])
```



Example 2

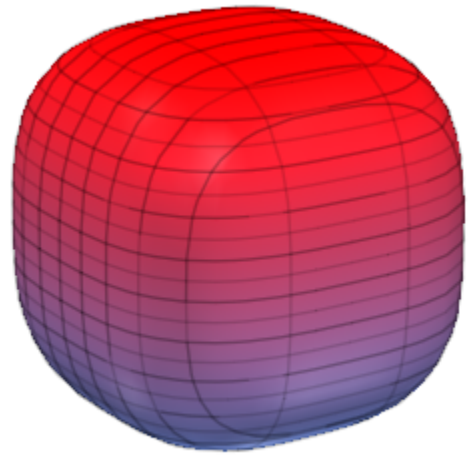
In the previous example, we set *z* values for the contour lines explicitly. There is an easier way of specifying equidistant lines, though, by giving the special value `Automatic`, followed by the number of lines to use. For example, implicit surfaces by default use 15 lines in each direction of space:

```
plot(plot::Implicit3d(abs(x)^3+abs(y)^3+abs(z)^3 - 1,
                      x = -1..1, y=-1..1, z=-1..1),
     Axes = None, Scaling = Constrained)
```



To change the number of lines, we use the syntax outlined above:

```
plot(plot::Implicit3d(abs(x)^3+abs(y)^3+abs(z)^3 - 1,
                      x = -1..1, y=-1..1, z=-1..1,
                      XContours = [Automatic, 4],
                      YContours = [Automatic, 11],
                      ZContours = [Automatic, 21]),
     Axes = None, Scaling = Constrained)
```

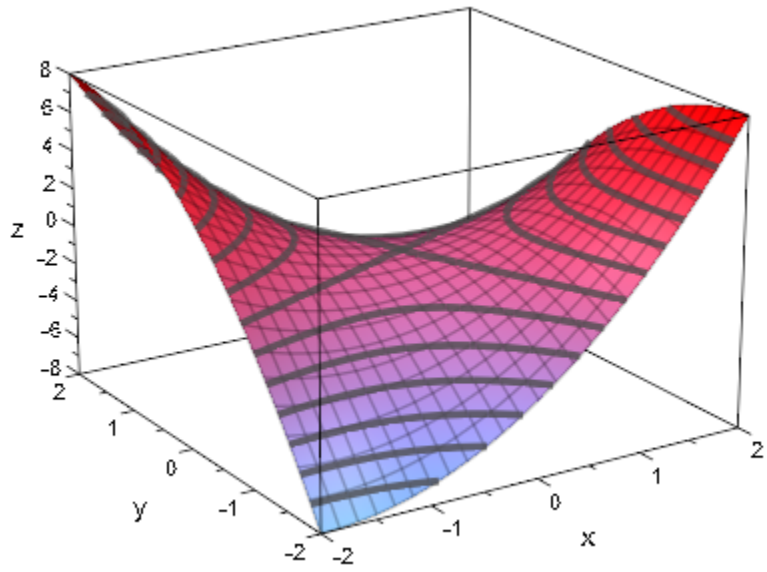


Note that two of the lines are at the extremal values and therefore usually not visible.

Example 3

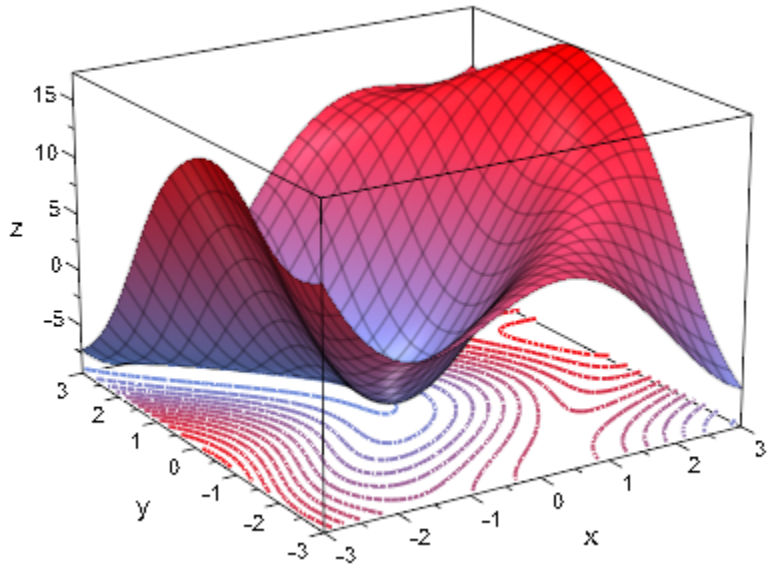
Contour lines are drawn using the same settings for `LineWidth` and `LineColor` as parameter lines are. In the following example, we use a modified copy of a function object that *only* displays contour lines, but with settings different from the function object proper.

```
f := plot::Function3d(x^2 - 2*x*y - y^2, x = -2..2, y = -2..2):
plot(f, plot::modify(f, ZContours = [Automatic, 15],
                        LineWidth = 1,
                        LineColor = RGB::Gray30.[0.8],
                        XLinesVisible = FALSE,
                        YLinesVisible = FALSE,
                        Filled = FALSE))
```



By using a transformation that maps space into a plane, we can use this technique (by setting some more options) to display height-colored contour lines below a function plot:

```
f := plot::Function3d(8*sin(x-cos(y))+(x^2+x*y),
                      x = -3..3, y = -3..3, Submesh=[2,2]):
plot(f, plot::Transform3d([0, 0, -9], [1, 0, 0, 0, 1, 0, 0, 0, 0],
             plot::modify(f, ZContours = [Automatic, 15],
                             LineWidth = 0.5,
                             LineColorType = Dichromatic,
                             LineColor = RGB::Red,
                             LineColor2 = RGB::CornflowerBlue,
                             XLinesVisible = FALSE,
                             YLinesVisible = FALSE,
                             Filled = FALSE)))
```



See Also

MuPAD Functions

`LineColor` | `LinesVisible` | `LineWidth`

PointColor

Color of points

Value Summary

Inherited	Color

Graphics Primitives

Objects	PointColor Default Values
`plot::Matrixplot`, `plot::Point2d`, `plot::Point3d`, `plot::PointList2d`, `plot::PointList3d`, `plot::SparseMatrixplot`, `plot::Sum`	`RGB::MidnightBlue`
`plot::Listplot`, `plot::QQplot`, `plot::Scatterplot`	`RGB::Black`

Description

`PointColor` determines the color of points. The `RGB` library provides many pre-defined colors such as `RGB::Red` etc. See section Colors of this document for more information on colors.

Many graphical objects such as curves, surfaces etc. are approximated by a numerical mesh. With `PointsVisible` = `TRUE`, the points of this mesh become visible. These points do *not* react to `PointColor`.

`PointColor` cannot be animated.

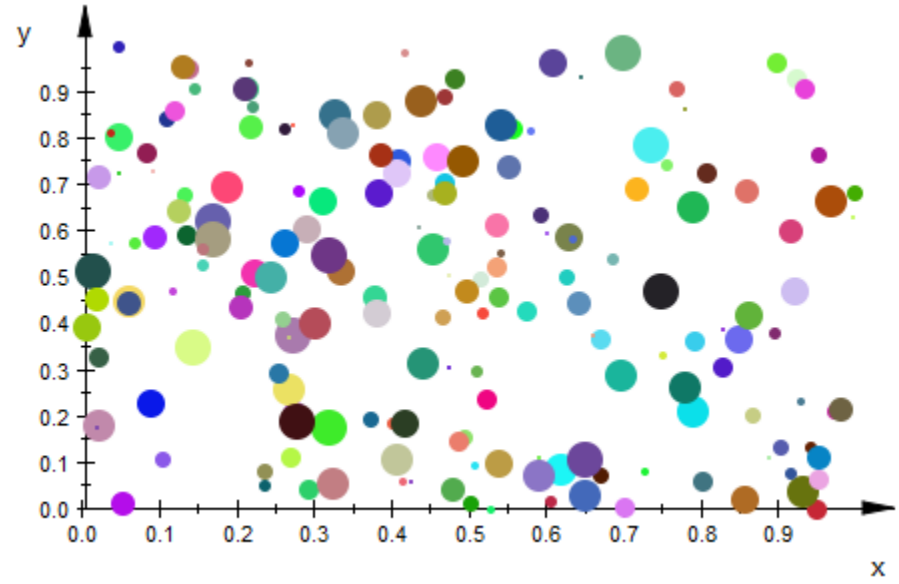
For points of type `plot::Point2d` and `plot::Point3d`, the point color can also be set by the attribute `Color`.

Examples

Example 1

We plot a cluster of random points with random sizes and random colors:

```
r := frandom:
plot(plot::Point2d([r(),r()], PointSize = 5*r()*unit::mm,
                   PointColor = [r(), r(), r()])
     $ i = 1 .. 200):
```




```
delete r:
```

Example 2

We can access the `PointColor` attribute from a point and change it:

```
p := plot::Point2d(1, 2):
p::PointColor := RGB::Black:
p::PointColor
```

```
[0.0, 0.0, 0.0]
```

```
delete p:
```

See Also

MuPAD Functions

`PointColor2` | `PointColorType` | `PointSize` | `PointStyle` | `PointsVisible`

PointColor2

Secondary point color for color blends

Value Summary

Optional	Color

Graphics Primitives

Objects	PointColor2 Default Values
`plot::SparseMatrixplot`	`RGB::Red`

Description

`PointColor2` sets the secondary point color in objects of type `plot::SparseMatrixplot`.

Objects of type `plot::SparseMatrixplot` color their points according to the attribute `PointColorType` = `Flat` or `PointColorType` = `Dichromatic`, respectively.

With `PointColorType` = `Flat`, all points in a `plot::SparseMatrixplot` object are displayed in the color given by `PointColor`.

With `PointColorType` = `Dichromatic`, the points are colored differently using a color blend from the color `PointColor` to the color `PointColor2`. The actual color of a point indicates the size of the matrix entry visualized by `plot::SparseMatrixplot`.

`PointColor` corresponds to small matrix entries, `PointColor2` corresponds to large matrix entries.

Examples

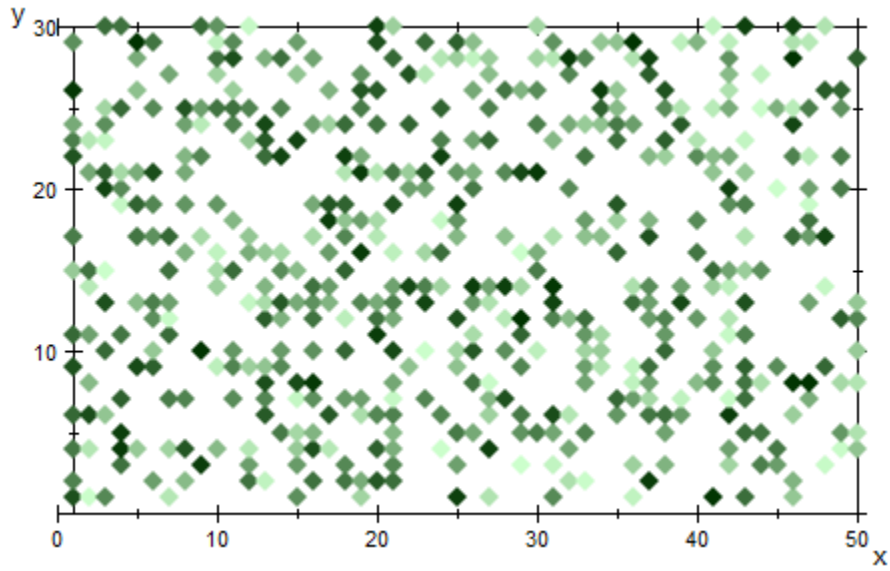
Example 1

We create a 30×50 matrix with 500 random entries.

```
smp := plot::SparseMatrixplot(
            matrix::random(30, 50, 500, frandom)):
```

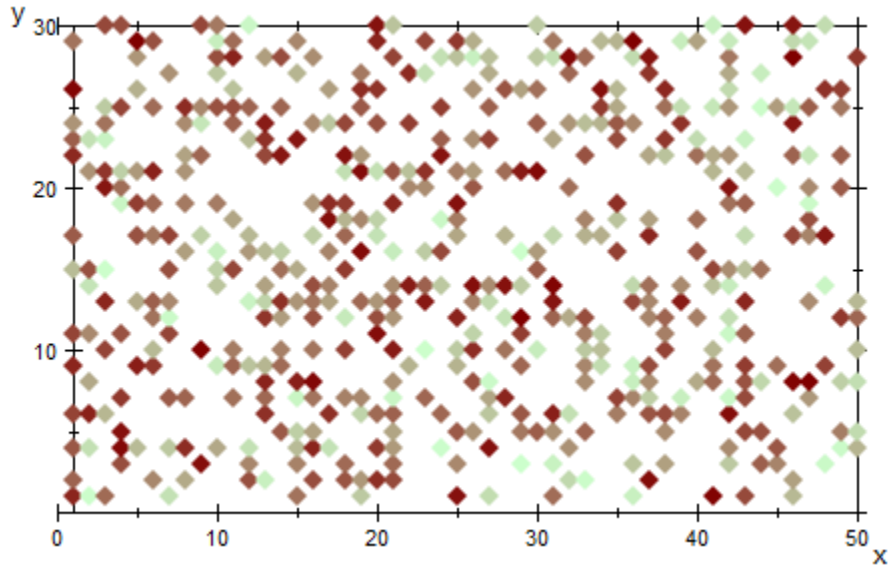
We use the color type `Dichromatic` and request a dark green for large matrix entries and a light green for small matrix entries:

```
smp::PointColorType := Dichromatic:
smp::PointColor := RGB::LightGreen:
smp::PointColor2 := RGB::DarkGreen:
smp::PointStyle := FilledDiamonds:
smp::PointSize := 2.5*unit::mm:
plot(smp):
```



The secondary color is changed to a dark red:

```
smp::PointColor2 := RGB::DarkRed:
plot(smp):
```




```
delete smp:
```

See Also

MuPAD Functions

PointColor | PointColorType | PointSize | PointStyle

PointColorType

Point coloring types

Value Summary

Optional	`Dichromatic`, or `Flat`

Graphics Primitives

Objects	PointColorType Default Values
`plot::SparseMatrixplot`	`Flat`

Description

`PointColorType` controls the type of point coloring used in objects of type `plot::SparseMatrixplot`.

With `PointColorType` = `Flat`, all points in a `plot::SparseMatrixplot` object are displayed in the color given by `PointColor`.

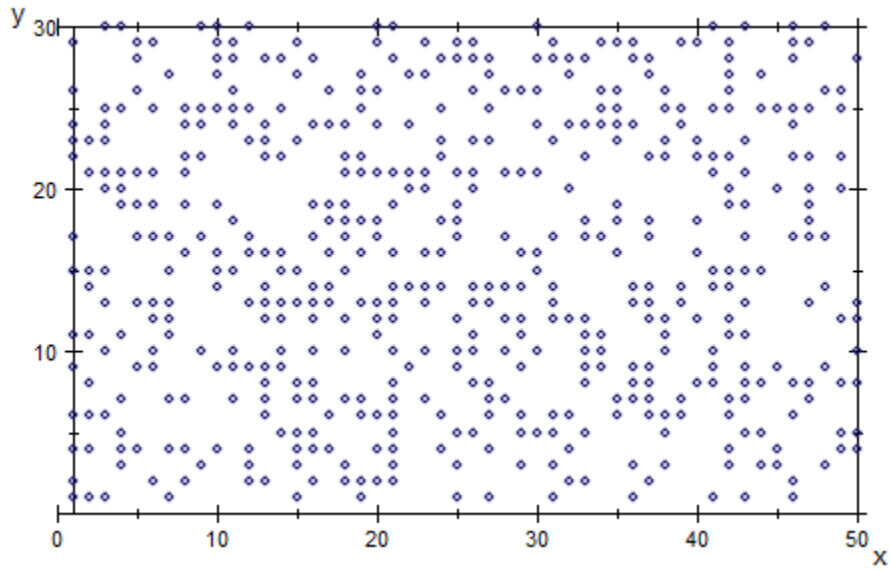
With `PointColorType` = `Dichromatic`, the points are colored differently using a color blend from the color `PointColor` to the color `PointColor2`. The actual color of a point indicates the size of the matrix entry visualized by `plot::SparseMatrixplot`.

Examples

Example 1

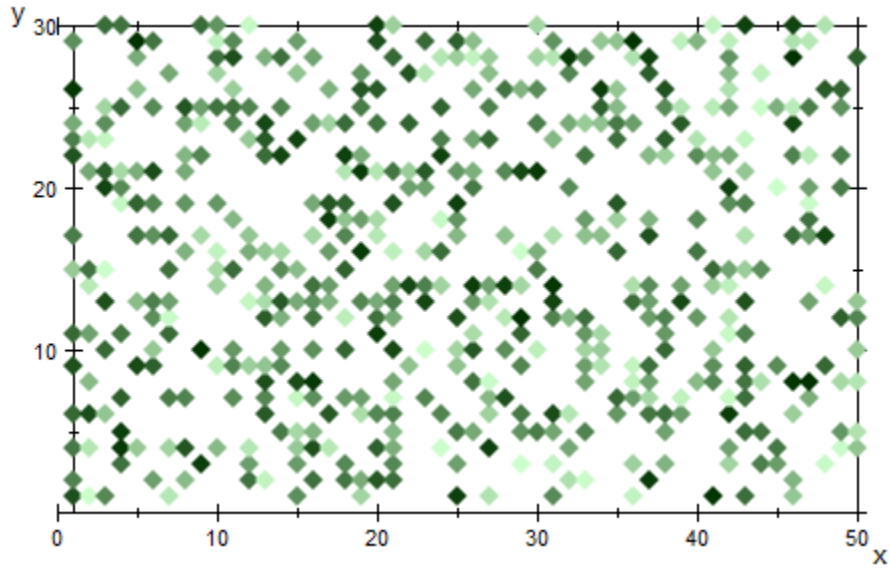
We create a 30×50 matrix with 500 random entries. With the default setting of `PointColorType` = `Flat`, all nonzero entries are displayed in the color given by `PointColor`:

```
smp := plot::SparseMatrixplot(
              matrix::random(30, 50, 500, frandom)):
plot(smp):
```



We change the color type to `Dichromatic` and request a dark green for large matrix entries and a light green for small matrix entries:

```
smp::PointColorType := Dichromatic:
smp::PointColor := RGB::LightGreen:
smp::PointColor2 := RGB::DarkGreen:
smp::PointStyle := FilledDiamonds:
smp::PointSize := 2.5*unit::mm:
plot(smp):
```




```
delete smp:
```

See Also

MuPAD Functions

PointColor | PointColor2 | PointSize | PointStyle

PointSize

Size of points

Value Summary

Inherited	Positive output size

Graphics Primitives

Objects	PointSize Default Values
`plot::Bars2d`, `plot::Bars3d`, `plot::Conformal`, `plot::Curve2d`, `plot::Curve3d`, `plot::Cylindrical`, `plot::Dodecahedron`, `plot::Function2d`, `plot::Function3d`, `plot::Hexahedron`, `plot::Histogram2d`, `plot::Icosahedron`, `plot::Integral`, `plot::Listplot`, `plot::Matrixplot`, `plot::Octahedron`, `plot::Ode2d`, `plot::Ode3d`, `plot::Point2d`, `plot::Point3d`, `plot::PointList2d`, `plot::PointList3d`, `plot::Polar`, `plot::Polygon2d`, `plot::Polygon3d`, `plot::Prism`, `plot::Pyramid`, `plot::QQplot`, `plot::Scatterplot`, `plot::Spherical`, `plot::Surface`, `plot::SurfaceSet`, `plot::SurfaceSTL`, `plot::Sweep`, `plot::Tetrahedron`, `plot::Turtle`, `plot::VectorField3d`, `plot::Waterman`, `plot::XRotate`, `plot::ZRotate`	`1.5`
`plot::Rootlocus`, `plot::SparseMatrixplot`	`1.0`

Objects	PointSize Default Values
`plot::Sequence`	2

Description

`PointSize` determines the physical size of points. The value should be specified as an absolute physical length including a length unit such as `PointSize = 1.5*unit::mm`. Numbers without a physical unit give the size in mm.

Typical points have a size of only a few pixels on the screen. Hence, the renderers cannot always react to small changes of the `PointSize`, because the actual size of the graphical points can attain only discrete values.

Depending on your hardware, there is a maximal size of the graphical points that can be rendered in 3D. If the `PointSize` is too large, the 3D renderer uses the maximal size that is supported.

Many graphical objects such as curves, surfaces etc. are approximated by a numerical mesh. With `PointsVisible` = `TRUE`, the points of this mesh become visible. These points react to `PointSize`.

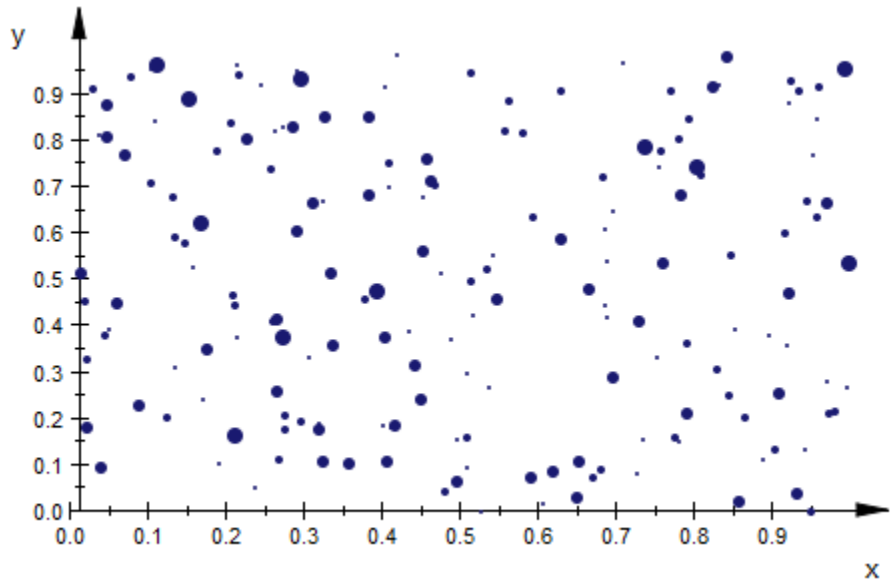
`PointSize` cannot be animated.

Examples

Example 1

We plot a cluster of points with random sizes within the unit square:

```
r := frandom:
plot(plot::Point2d(r(),r(), PointSize = 2*r()*unit::mm)
     $ i = 1 .. 200)
```

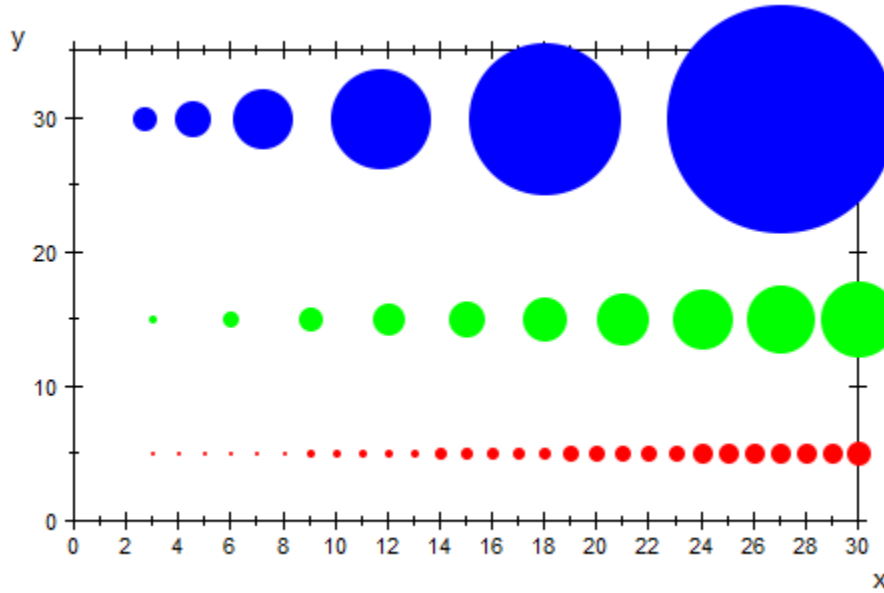



```
delete r:
```

Example 2

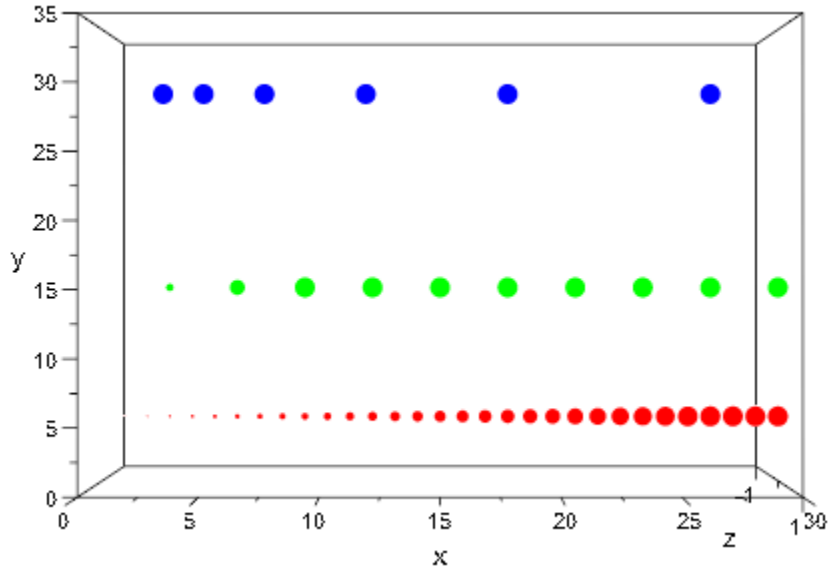
Due to pixelation, there is only a discrete number of `PointSize` values that the renderers can display faithfully. Further, note that the large points may protrude over the edges of the viewing box without being clipped:

```
plot(plot::Point2d([i, 5], PointSize = i*0.1*unit::mm,
                   Color = RGB::Red)
     $ i = 1 .. 30,
     plot::Point2d([3*i, 15], PointSize = i*unit::mm,
                   Color = RGB::Green)
     $ i = 1 .. 10,
     plot::Point2d([9*i , 30], PointSize = i*unit::cm,
                   Color = RGB::Blue)
     $ i in [0.3, 0.5, 0.8, 1.3, 2, 3],
     ViewingBox = [0 .. 30, 0 .. 35], Axes = Boxed)
```



Here are the same points in 3D. Note the threshold for `PointSize` beyond which the graphical points do not grow:

```
plot(plot::Point3d([i, 5, 0], PointSize = i*0.1*unit::mm,
                   Color = RGB::Red)
     $ i = 1 .. 30,
     plot::Point3d([3*i, 15, 0], PointSize = i*unit::mm,
                   Color = RGB::Green)
     $ i = 1 .. 10,
     plot::Point3d([9*i , 30, 0], PointSize = i*unit::cm,
                   Color = RGB::Blue)
     $ i in [0.3, 0.5, 0.8, 1.3, 2, 3],
     ViewingBox = [0 .. 30, 0 .. 35, -1 .. 1],
     Axes = Boxed, CameraDirection = [0, -10, 1000],
     YXRatio = 2/3)
```



Example 3

We can access the `PointSize` attribute from a point and change it:

```
p := plot::Point2d(1, 2):
p::PointSize := 4*unit::inch:
p::PointSize
```

$$\frac{508\ \text{mm}}{5}$$

```
delete p:
```

See Also

MuPAD Functions

`PointColor` | `PointStyle` | `PointsVisible`

PointStyle

Presentation style of points

Value Summary

Inherited	`Squares`, `FilledSquares`, `Circles`, `FilledCircles`, `Crosses`, `XCrosses`, `Diamonds`, `FilledDiamonds`, or `Stars` (2D),`FilledSquares` or `FilledCircles` (3D)

Graphics Primitives

Objects	PointStyle Default Values
`plot::Bars2d`, `plot::Bars3d`, `plot::Curve2d`, `plot::Curve3d`, `plot::Cylindrical`, `plot::Dodecahedron`, `plot::Function2d`, `plot::Function3d`, `plot::Hexahedron`, `plot::Histogram2d`, `plot::Icosahedron`, `plot::Implicit3d`, `plot::Integral`, `plot::Listplot`, `plot::Matrixplot`, `plot::Octahedron`, `plot::Ode2d`, `plot::Ode3d`, `plot::Point2d`, `plot::Point3d`, `plot::PointList2d`, `plot::PointList3d`, `plot::Polar`, `plot::Polygon2d`, `plot::Polygon3d`, `plot::Prism`, `plot::Pyramid`, `plot::QQplot`, `plot::Rootlocus`, `plot::Scatterplot`, `plot::Sequence`, `plot::Spherical`, `plot::Surface`, `plot::SurfaceSet`, `plot::SurfaceSTL`,	`FilledCircles`

Objects	PointStyle Default Values
`plot::Sweep`, `plot::Tetrahedron`, `plot::Turtle`, `plot::VectorField3d`, `plot::Waterman`, `plot::XRotate`, `plot::ZRotate`	
`plot::SparseMatrixplot`	`Diamonds`

Description

`PointStyle` determines the presentation style of points. The various styles are demonstrated in "Example 1" on page 24-2003.

In 3D, only two styles `FilledCircles` and `FilledSquares` are supported by the renderer.

Many graphical objects such as curves, surfaces etc. are approximated by a numerical mesh. With `PointsVisible` = `TRUE`, the points of this mesh become visible. These points react to `PointStyle`.

`PointStyle` cannot be animated.

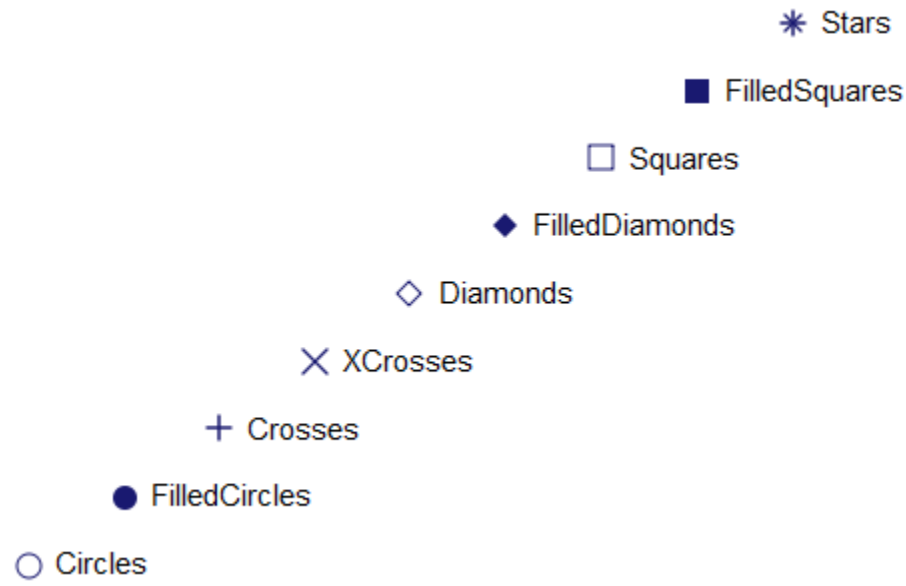
Examples

Example 1

We plot 2D points in all available styles:

```
styles := [Circles, FilledCircles,
           Crosses, XCrosses,
           Diamonds, FilledDiamonds,
           Squares, FilledSquares,
           Stars]:
points := null():
for i from 1 to nops(styles) do
  points := points,
            plot::Point2d([i, i], PointStyle = styles[i],
                          Title = expr2text(styles[i]),
```

```
                          TitlePosition = [i + 0.3, i - 0.15]):
end_for:
plot(points, PointSize = 3*unit::mm, Axes = None,
     TitleAlignment = Left):
```

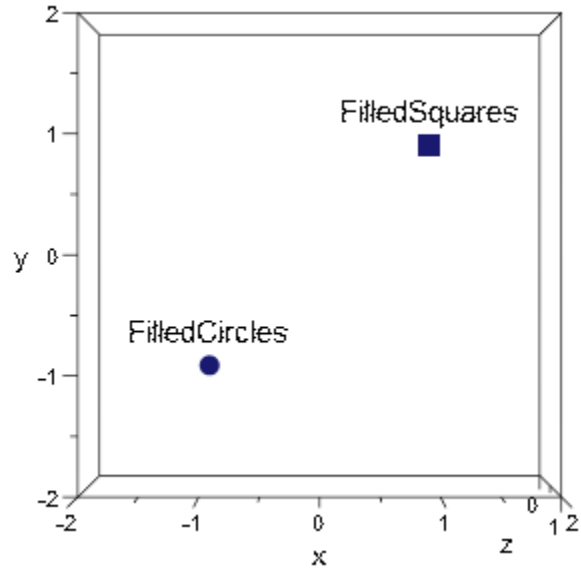



```
delete styles, points, i:
```

Example 2

In 3D, the renderer only supports the point styles `FilledCircles` and `FilledSquares`:

```
plot(plot::Point3d([-1, -1, 0], PointStyle = FilledCircles,
                   Title = "FilledCircles",
                   TitlePosition = [-1, -0.8, 0]),
     plot::Point3d([1, 1, 0], PointStyle = FilledSquares,
                   Title = "FilledSquares",
                   TitlePosition = [1, 1.2, 0]),
     PointSize = 3*unit::mm,
     ViewingBox = [-2..2, -2..2, 0..1],
     CameraDirection = [0, -1, 1000]):
```



Example 3

We can access the `PointStyle` attribute from a point and change it:

```
p := plot::Point2d(1, 2):
p::PointStyle := Diamonds:
p::PointStyle
```

```
Diamonds
```

```
delete p:
```

See Also

MuPAD Functions

`PointColor` | `PointSize` | `PointsVisible`

PointsVisible

Visibility of mesh points

Value Summary

Inherited	FALSE, or TRUE

Graphics Primitives

Objects	PointsVisible Default Values
plot::Conformal, plot::Curve2d, plot::Curve3d, plot::Cylindrical, plot::Dodecahedron, plot::Function2d, plot::Function3d, plot::Hexahedron, plot::Icosahedron, plot::Implicit3d, plot::Integral, plot::Octahedron, plot::Polar, plot::Polygon2d, plot::Polygon3d, plot::Prism, plot::Pyramid, plot::Rootlocus, plot::Spherical, plot::Sum, plot::Surface, plot::SurfaceSet, plot::SurfaceSTL, plot::Sweep, plot::Tetrahedron, plot::Tube, plot::Turtle, plot::Waterman, plot::XRotate, plot::ZRotate	FALSE
plot::Listplot, plot::Matrixplot, plot::Ode2d, plot::Ode3d, plot::QQplot, plot::Scatterplot, plot::Sequence, plot::SparseMatrixplot, plot::VectorField3d	TRUE

Description

`PointsVisible = TRUE/FALSE` enables/disables the plotting of mesh and submesh points.

The mesh points react to the attributes `PointSize` and `PointStyle`. However, they do *not* react to the attribute `PointColor`. Typically, mesh points are painted in the same color used for the line objects defined by the mesh points.

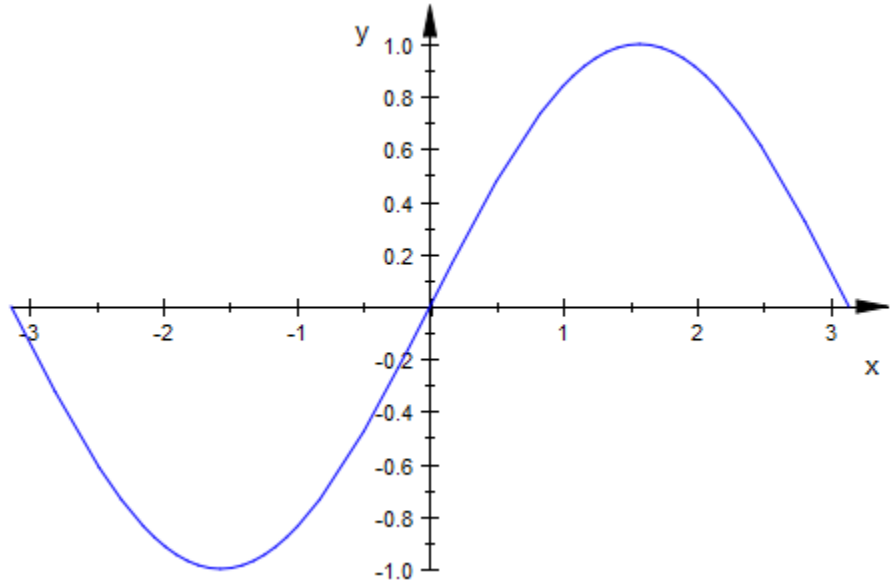
`PointsVisible` cannot be animated.

Examples

Example 1

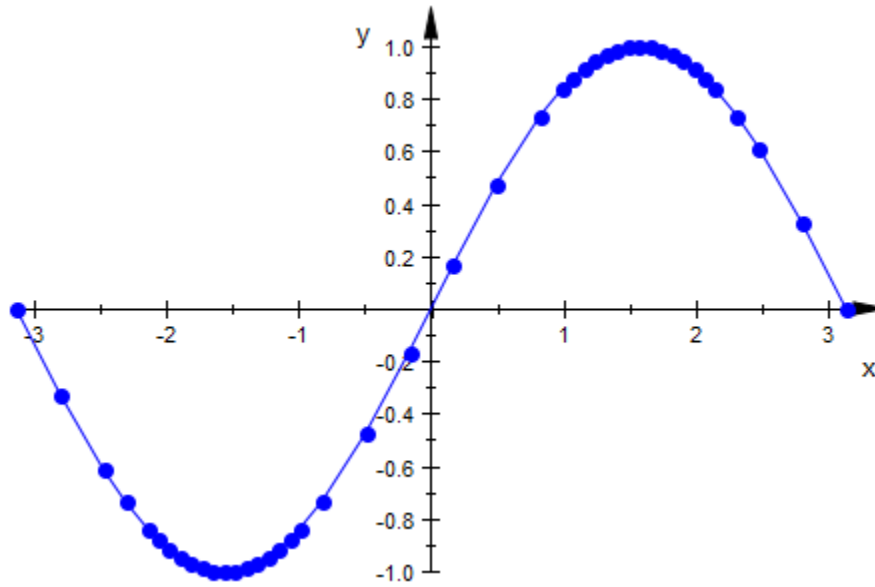
We plot the sine function on a rather coarse mesh using the `PointsVisible` default value `FALSE`:

```
f := plot::Function2d(sin(x), x = -PI .. PI, Mesh = 20):
plot(f):
```



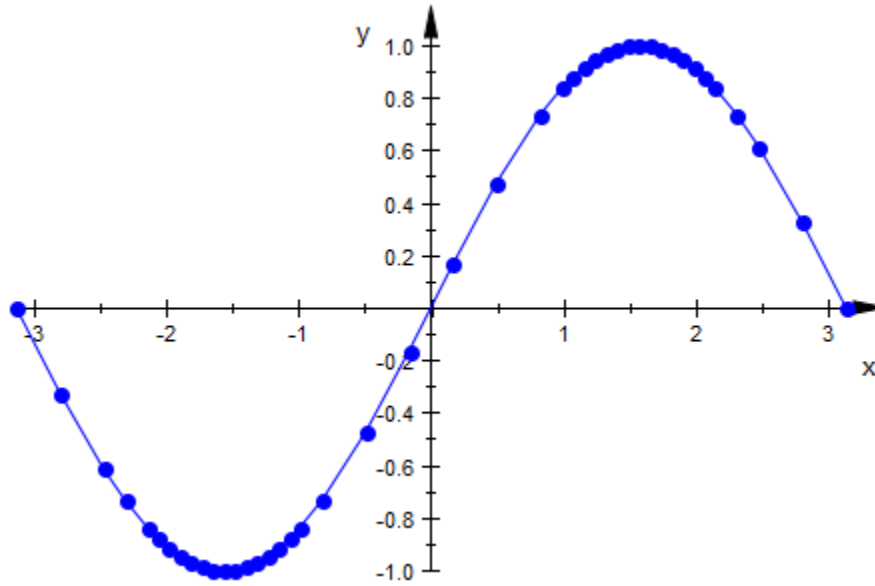
We use `PointsVisible` = `TRUE` to make the mesh points visible:

```
f::PointsVisible := TRUE:
plot(f, PointSize = 2*unit::mm)
```



We enable adaptive plotting:

```
f::AdaptiveMesh := 2:
plot(f, PointSize = 2*unit::mm)
```




```
delete f:
```

See Also

MuPAD Functions
`PointSize` | `PointStyle`

BarCenters, BarWidths

Position of bars

Value Summary

`BarCenters`, `BarWidths`	Optional	List of arithmetical expressions

Graphics Primitives

Objects	Default Values
`plot::Bars2d`	`BarWidths`: [[`1.0`]]

Description

`BarCenters` and `BarWidths` govern horizontal center positions and widths of the bars in 2D bar plots of Type `plot::Bars2d`.

A plot of type `plot::Bars2d` serves for visualizing and comparing discrete data samples by a 2D bar plot.

The data values define the vertical coordinates of the bars. The position along the horizontal axis and the horizontal width of the bars are controlled by the attributes `BarCenters` and `BarWidths`.

The value of the attribute `BarCenters` may be a list [x_1, x_2, …] of numerical values or expressions of the animation parameter. These values define the horizontal coordinates of the bar centers.

If several data samples are to be displayed simultaneously, the value of `BarCenters` may be a list of lists [[x_{11}, x_{12}, …], [x_{21}, x_{22}, …], …], where x_{ij} is the center position of the bar indicating the j-th data point in the i-th sample.

If the length of a list in the `BarCenters` attribute is smaller than the number of data in the corresponding sample, the center values are chosen automatically for the surplus data items.

If the length of the `BarCenters` list is larger than the number of corresponding data items, the surplus center values are ignored.

Setting `BarCenters` = `[`x_1`]`, the first bar is centered at $x = x_1$, while the standard distance between the bars is kept. Thus, `BarCenters` = `[`x_1`]` allows to shift the entire bar plot along the horizontal axis.

The value of the attribute `BarWidths` may be a numerical value or an expression of the animation parameter. This sets the horizontal width of *all* bars.

Alternatively, it may be a list of values [w_1, w_2, …] allowing to define differents widths of the bars. If several data samples are specified, each data sample uses the same list of `BarWidths` values.

Alternatively, the value of `BarWidths` may be a list of lists [[w_{11}, w_{12}, …], [w_{21}, w_{22}, …], …], where w_{ij} is the horizontal width of the bar indicating the j-th data point in the i-th sample.

If the length of a list in the `BarWidths` attribute is smaller than the number of data in the corresponding sample, the width values are chosen automatically for the surplus data items.

If the length of the `BarWidths` list is larger than the number of corresponding data items, the surplus width values are ignored.

The `BarWidths` attribute only has an effect in conjunction with the (default) `BarStyle` = `Boxes`.

If the attribute `DrawMode` = `Horizontal` is set in the `plot::Bars2d` object, the bars are drawn from left to right instead from bottom to top.

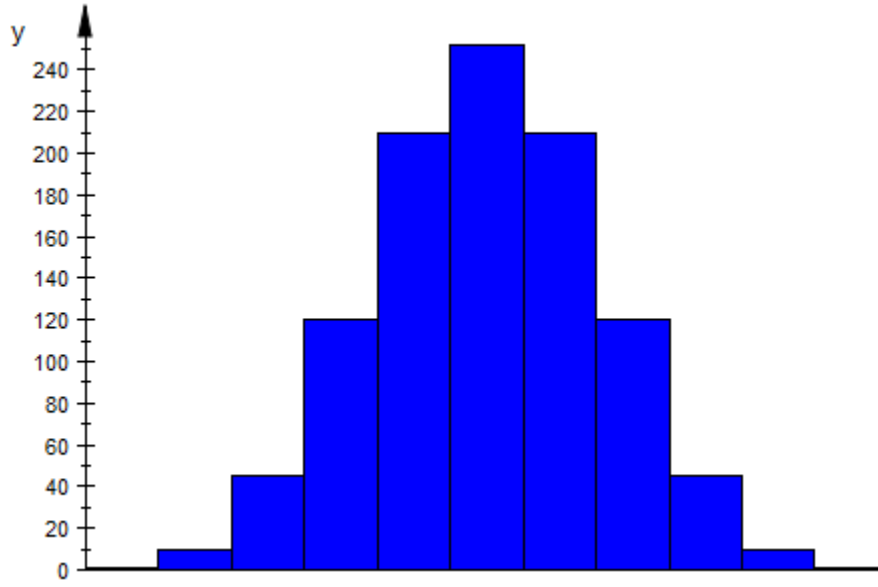
In this case, the attributes `BarCenters` and `BarWidths` refer to the vertical coordinates of the bars.

Examples

Example 1

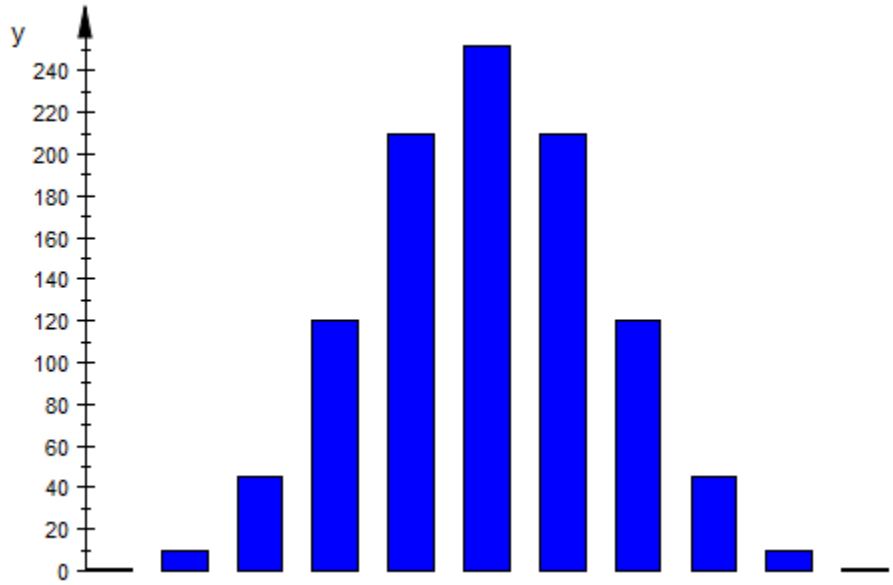
We display some discrete values as a bar plot:

```
data := [binomial(10, j) $ j = 0..10]:
plot(plot::Bars2d(data, BarCenters = [j $ j = 0..10])):
```



We reduce the widths of the bars:

```
plot(plot::Bars2d(data, BarCenters = [j $ j = 0..10],
                  BarWidths = 0.6)):
```

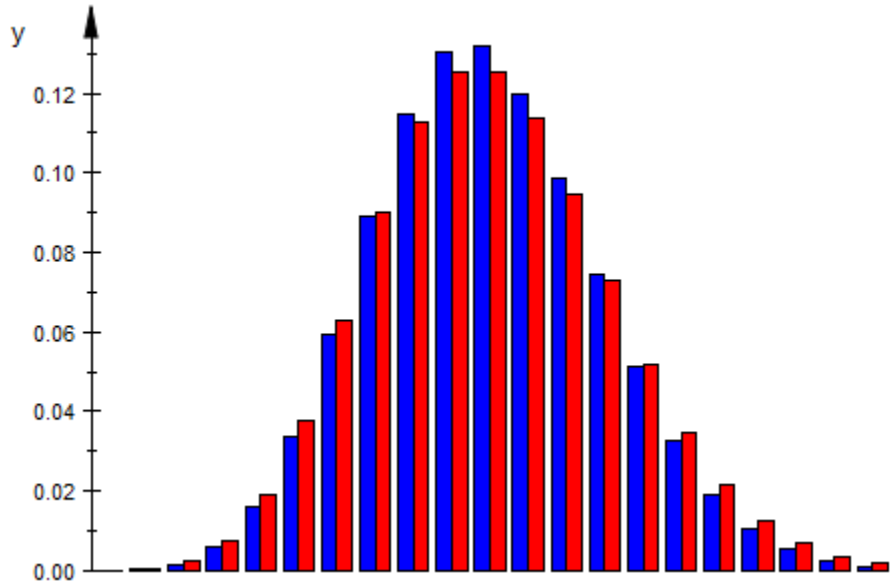



```
delete data:
```

Example 2

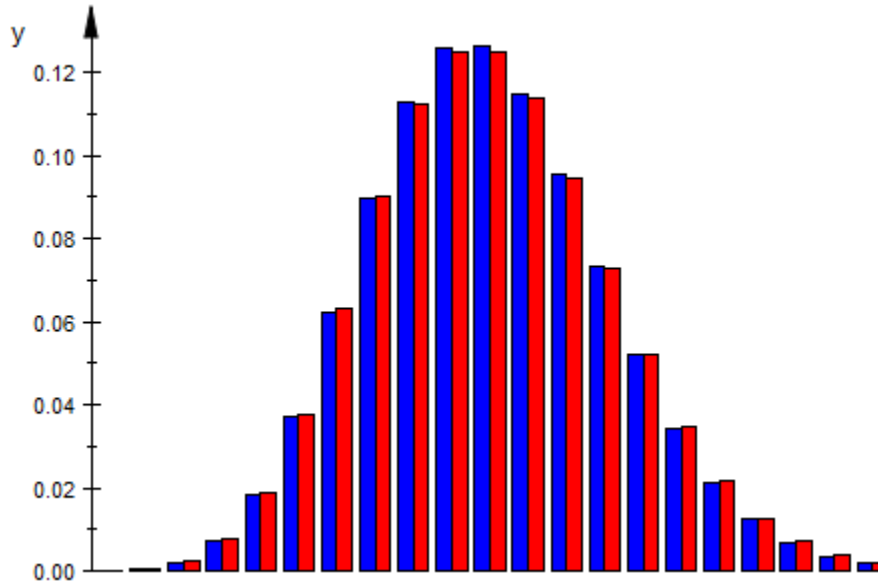
For large values of n and small values of p, the binomial distribution `stats::binomialPF(n, p)` is approximated by the Poisson distribution `stats::poissonPF(n*p)`. We demonstrate this fact by plotting the probability values of these distributions in one bar plot:

```
n := 100: p:= 0.1:
data1 := [stats::binomialPF(n, p)(j) $ j = 0..20]:
data2 := [stats::poissonPF(n*p)(j) $ j = 0..20]:
plot(plot::Bars2d([data1, data2],
           BarCenters = [[j $ j = 0..20], [j + 0.4 $ j = 0..20]],
           BarWidths = 0.4)):
```



The approximation is better for larger values of n. We reduce p accordingly to have the same value of $n\,p$ as in the previous plot:

```
n := 500: p:= 0.02:
data1 := [stats::binomialPF(n, p)(j) $ j = 0..20]:
data2 := [stats::poissonPF(n*p)(j) $ j = 0..20]:
plot(plot::Bars2d([data1, data2],
          BarCenters = [[j $ j = 0..20], [j + 0.4 $ j = 0..20]],
          BarWidths = 0.4)):
```




```
delete n, p, data1, data2:
```

See Also

MuPAD Functions
`BarStyle` | `DrawMode`

BarStyle, Shadows

Display style of bar plots

Value Summary

`BarStyle`	Optional	`Boxes`, `Lines`, `LinesPoints`, or `Points`
`Shadows`	Optional	`TRUE` or `FALSE`

Graphics Primitives

Objects	Default Values
`plot::Bars3d`	`BarStyle`: `Boxes`

`plot::Bars2d`	`BarStyle`: `Boxes`
	
	`Shadows`: `FALSE`

Description

`BarStyle` selects between bars drawn as boxes, as lines, just points, or lines with points. For box diagrams, `Shadows` can be used to have simple "shadows" drawn.

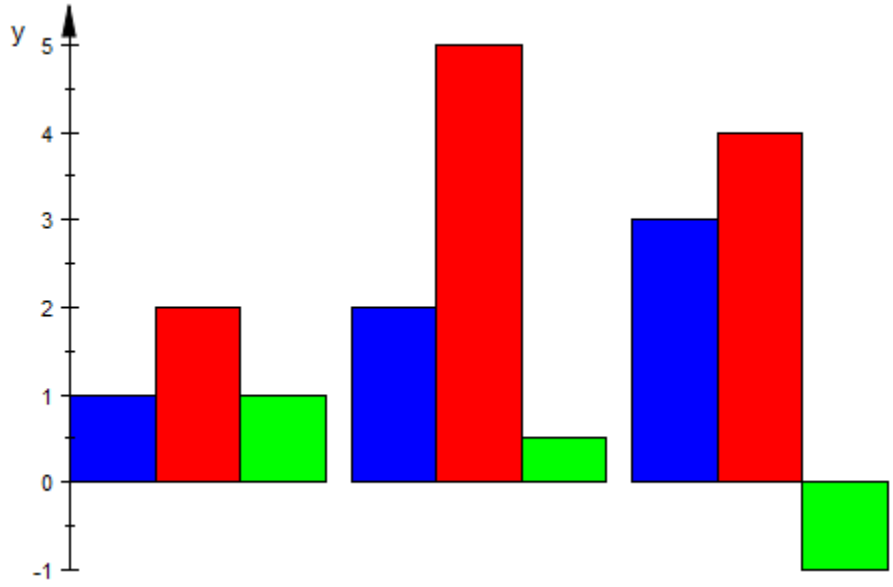
Bar plots can use different types of bars. The options are shown in the examples.

Examples

Example 1

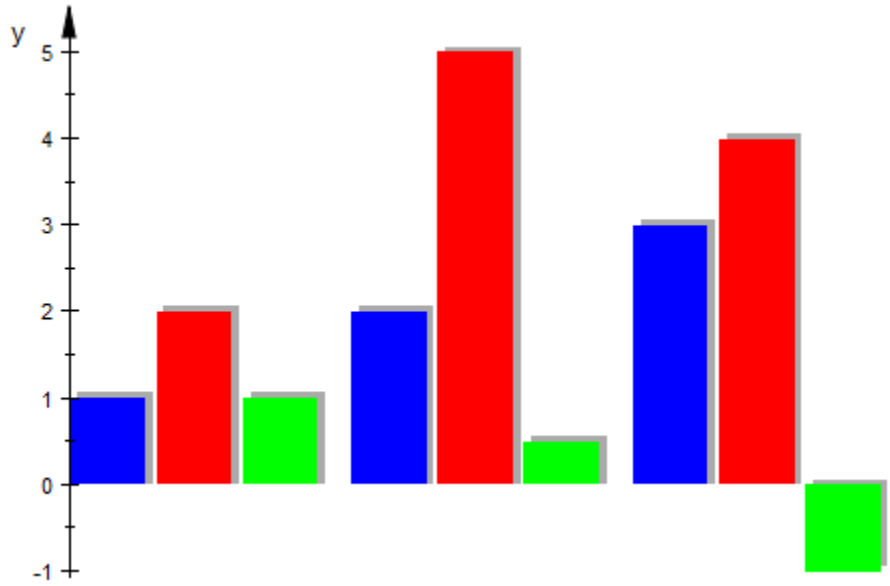
With only few data, the option `Boxes` is often the most useful one:

```
plot(plot::Bars2d([[1,2,3],[2,5,4],[1,0.5,-1]],
                  BarStyle = Boxes))
```



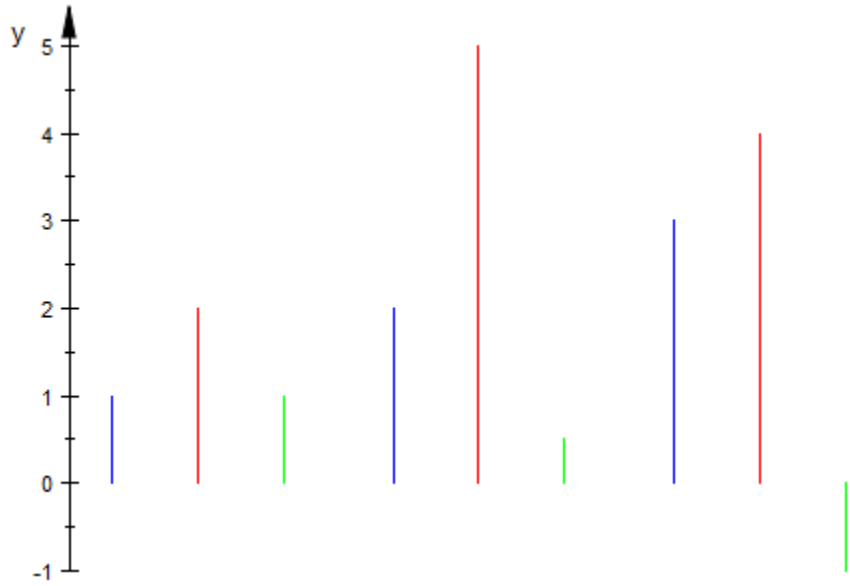
It can be combined with `Shadows = TRUE` and possibly `LinesVisible = FALSE` for a more pleasant display:

```
plot(plot::Bars2d([[1,2,3],[2,5,4],[1,0.5,-1]],
                  BarStyle = Boxes, Shadows = TRUE,
                  LinesVisible = FALSE))
```



Shadows are not displayed for the other bar styles:

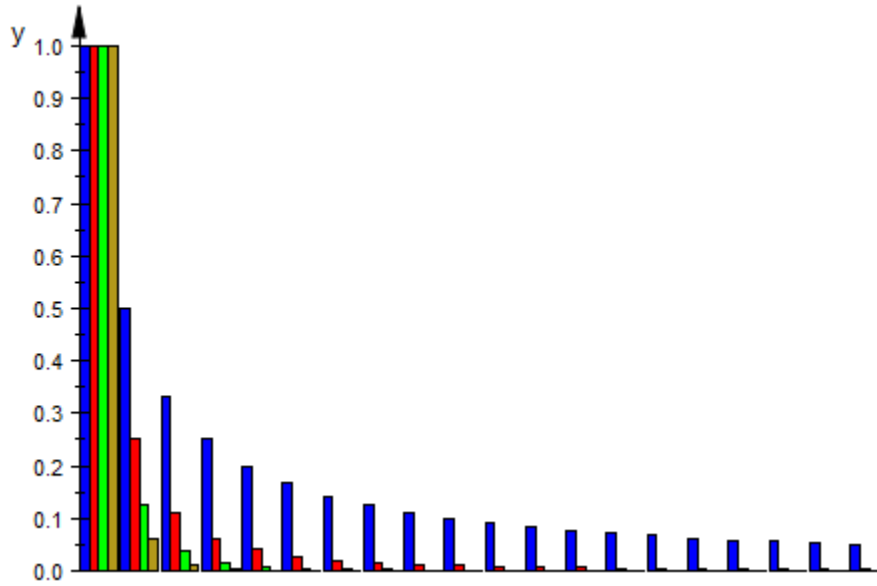
```
plot(plot::Bars2d([[1,2,3],[2,5,4],[1,0.5,-1]],
                  BarStyle = Lines, Shadows = TRUE))
```



Example 2

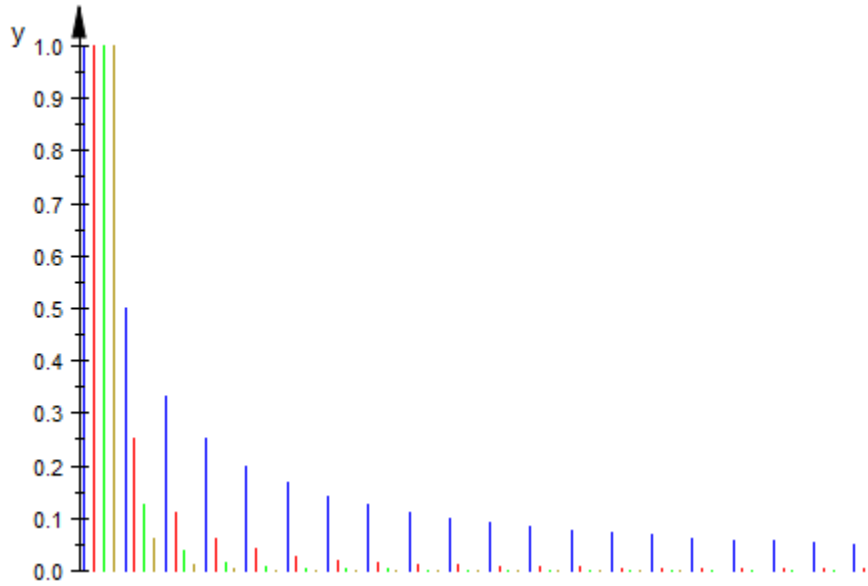
When more data is to be displayed, a bar plot may be less adequate:

```
b := plot::Bars2d([[1/i^k$i=1..20] $ k = 1..4]):
plot(b)
```

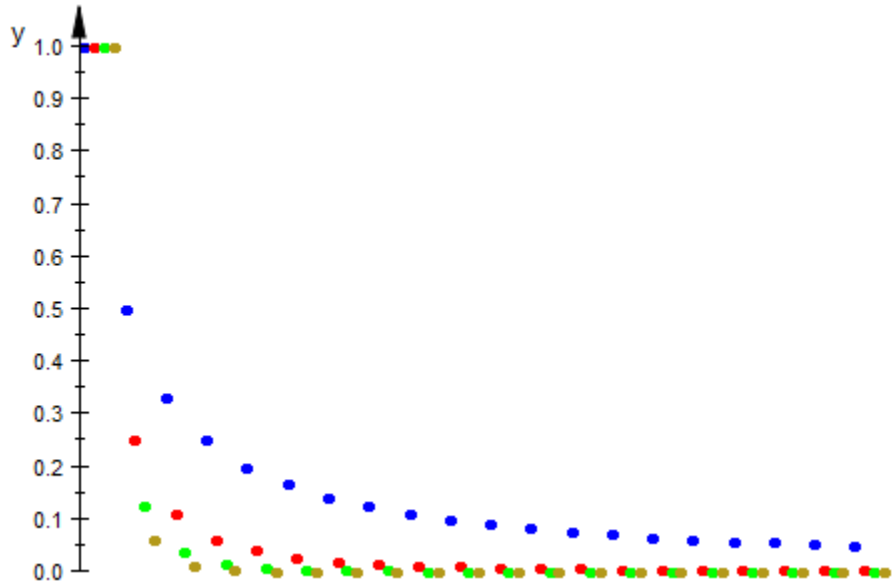


We demonstrate the alternatives without any further comment:

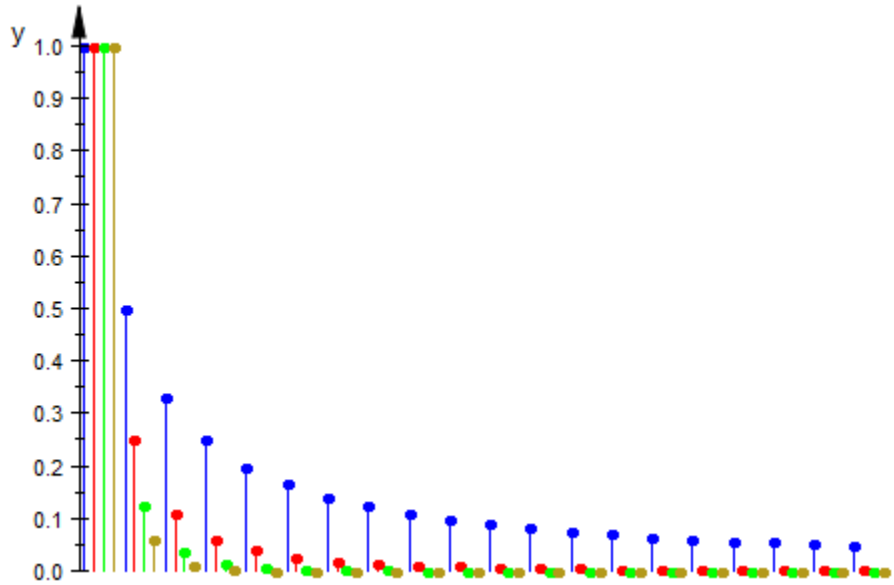
```
b::BarStyle := Lines:
plot(b)
```




```
b::BarStyle := Points:
plot(b)
```




```
b::BarStyle := LinesPoints:
plot(b)
```



See Also

MuPAD Functions
`Colors` | `FillPatterns`

Color

Main color

Value Summary

Library wrapper for "`Colors`, `FillColor`, `LightColor`, `LineColor`, and `PointColor`"	See below

Graphics Primitives

Objects	Color Default Values
`plot::Histogram2d`	`RGB::GeraniumLake`
`plot::Cylindrical`, `plot::Density`, `plot::Dodecahedron`, `plot::Function3d`, `plot::Hatch`, `plot::Hexahedron`, `plot::Icosahedron`, `plot::Implicit3d`, `plot::Matrixplot`, `plot::Octahedron`, `plot::Prism`, `plot::Pyramid`, `plot::Spherical`, `plot::Surface`, `plot::SurfaceSet`, `plot::SurfaceSTL`, `plot::Tetrahedron`, `plot::XRotate`, `plot::ZRotate`	`RGB::Red`
`plot::Box`, `plot::Cone`, `plot::Ellipsoid`, `plot::Parallelogram3d`, `plot::Plane`, `plot::Sphere`	`RGB::LightBlue`
`plot::Arrow2d`, `plot::Arrow3d`, `plot::Circle2d`, `plot::Circle3d`, `plot::Conformal`, `plot::Curve2d`, `plot::Curve3d`, `plot::Ellipse2d`,	`RGB::Blue`

Objects	Color Default Values
`plot::Function2d`, `plot::Implicit2d`, `plot::Line2d`, `plot::Line3d`, `plot::Lsys`, `plot::Parallelogram2d`, `plot::Polar`, `plot::Polygon2d`, `plot::Polygon3d`, `plot::Raster`, `plot::Rectangle`, `plot::Sequence`, `plot::Sum`, `plot::Turtle`, `plot::VectorField2d`, `plot::VectorField3d`	
`plot::Point2d`, `plot::Point3d`, `plot::PointList2d`, `plot::PointList3d`, `plot::SparseMatrixplot`	`RGB::MidnightBlue`
`plot::Sweep`	`RGB::Black.[0.25]`
`plot::Iteration`	`RGB::Grey50`
`plot::Bars2d`, `plot::Bars3d`, `plot::Boxplot`, `plot::Piechart2d`, `plot::Piechart3d`	
`plot::Waterman`	`RGB::SafetyOrange`
`plot::Integral`	`RGB::PaleBlue`

Description

`Color` refers to the "main color" of an object.

Depending on the object type, `Color` refers to the line color (e.g., `plot::Function2d`), the fill color (`plot::Surface`), the point color (`plot::Point2d`), the light color (`plot::PointLight`), or the one-and-only entry in `Colors` (`plot::Histogram2d`).

In general, the main color of an object is the first one available in the list

1. The first entry of `Colors`, if `Colors` contains exactly one entry.
2. `FillColor`
3. `LineColor`

4 `PointColor`

5 `LightColor`

The following object types deviate from this general rule and choose the line color as main color: `plot::Arc2d`, `plot::Arc3d`, `plot::Circle2d`, `plot::Circle3d`, `plot::Ellipse2d`, `plot::Ellipse3d`, `plot::Ode2d`, `plot::Ode3d`, `plot::Parallelogram2d`, `plot::Polygon2d`, `plot::Polygon3d`, and `plot::Rectangle`. `plot::Sequence` uses `PointColor` as the main color.

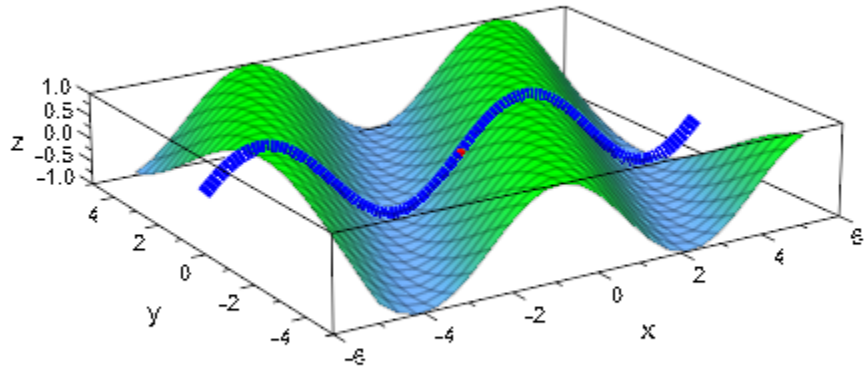
Note: `Color` is a *library attribute* and does not appear in the inspector.

Examples

Example 1

`Color` is useful for unified input of different object types:

```
plot(plot::Function3d(sin(x-y/2), Color = RGB::Green),
     plot::Point3d([0, 0, 0], Color = RGB::Red),
     plot::Curve3d([x, 0, sin(x)], x = -6..6,
                   LineWidth = 2*unit::mm,
                   Color = RGB::Blue),
     Scaling = Constrained)
```



See Also

MuPAD Functions

`Colors` | `FillColor` | `LineColor` | `PointColor`

Colors

List of colors to use

Value Summary

Optional	List of colors

Graphics Primitives

Objects	Colors Default Values
`plot::Bars2d`, `plot::Bars3d`, `plot::Ode2d`, `plot::Ode3d`	[`RGB::Blue`, `RGB::Red`, `RGB::Green`, `RGB::MuPADGold`, `RGB::Orange`, `RGB::Cyan`, `RGB::Magenta`, `RGB::LimeGreen`, `RGB::CadmiumYellowLight`, `RGB::AlizarinCrimson`, `RGB::Aqua`, `RGB::Lavender`, `RGB::SeaGreen`, `RGB::AureolineYellow`, `RGB::Banana`, `RGB::Beige`, `RGB::YellowGreen`, `RGB::Wheat`, `RGB::IndianRed`, `RGB::Black`]
`plot::MuPADCube`	[`RGB::Green`, `RGB::Blue`, `RGB::Red`, `RGB::Yellow`, `RGB::Antique`]
`plot::Boxplot`, `plot::Piechart2d`, `plot::Piechart3d`	[`RGB::Blue`, `RGB::Red`, `RGB::Green`, `RGB::MuPADGold`, `RGB::Orange`, `RGB::Cyan`, `RGB::Magenta`, `RGB::LimeGreen`, `RGB::CadmiumYellowLight`, `RGB::AlizarinCrimson`]

Description

`Colors` sets a list of colors to use for object parts.

Plot objects like `plot::Piechart3d` or `plot::MuPADCube` that use more than one color use `Colors` to have a configurable list of colors to use.

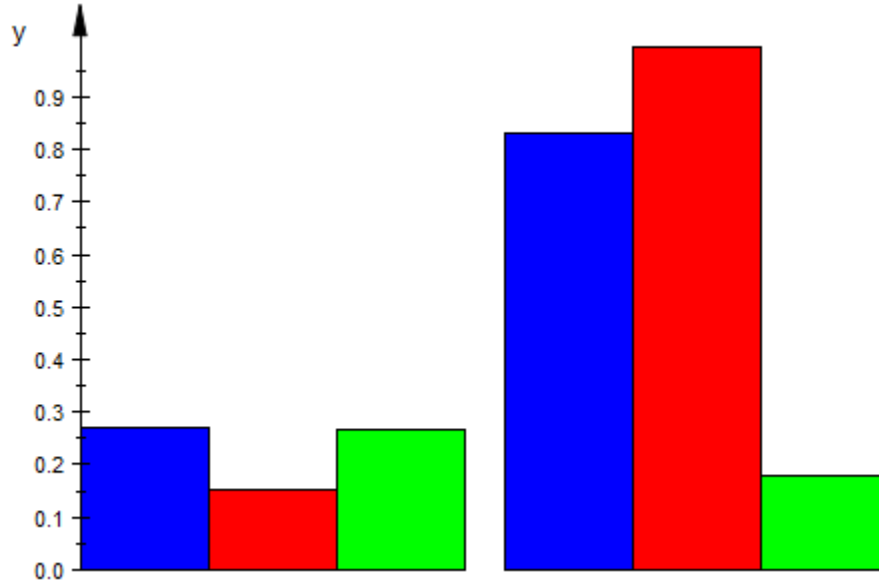
The length of the list in `Colors` need not be fixed, it just must not be empty. If the list contains more colors than needed, the remaining colors are simply not used; if the list contains fewer colors than needed, it will be used cyclically, i.e., as if it were repeated as often as necessary. Cf. “Example 2” on page 24-2030.

Examples

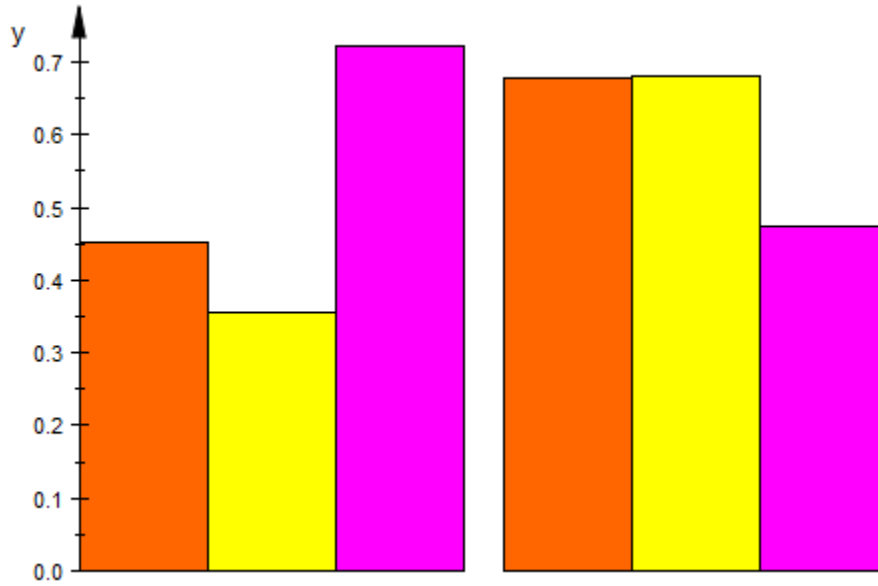
Example 1

Most of the statistical plots use `Colors` for the colors of their groups:

```
plot(plot::Bars2d([[frandom() $i=1..2] $ i = 1..3]))
```



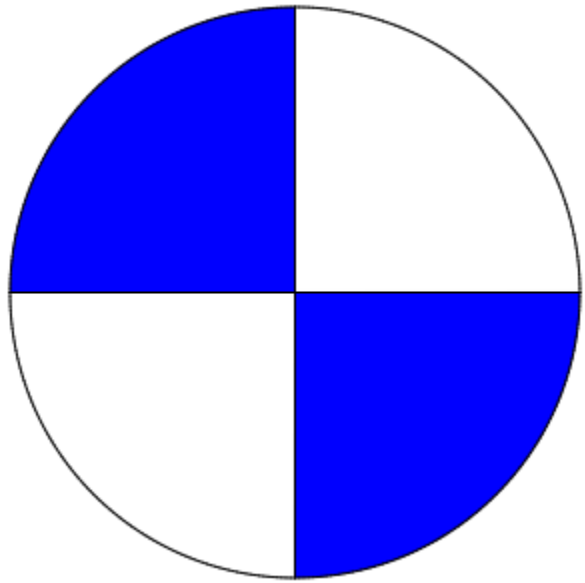

```
plot(plot::Bars2d([[frandom() $i=1..2] $ i = 1..3],
           Colors = [RGB::Orange, RGB::Yellow, RGB::Magenta]))
```



Example 2

If more colors are required than given in `Colors`, the given list is used cyclically:

```
plot(plot::Piechart2d([1, 1, 1, 1], Colors = [RGB::White, RGB::Blue]))
```



See Also

MuPAD Functions

`Color` | `FillColor` | `LineColor` | `PointColor`

FillColor, FillColor2

Color of areas and surfaces

Value Summary

`FillColor`, `FillColor2`	Inherited	Color

Graphics Primitives

Objects	Default Values
`plot::Histogram2d`	`FillColor`: `RGB::GeraniumLake`

`plot::Cylindrical`, `plot::Density`, `plot::Dodecahedron`, `plot::Function3d`, `plot::Hexahedron`, `plot::Icosahedron`, `plot::Implicit3d`, `plot::Matrixplot`, `plot::Octahedron`, `plot::Prism`, `plot::Pyramid`, `plot::Spherical`, `plot::Surface`, `plot::SurfaceSet`, `plot::SurfaceSTL`, `plot::Sweep`, `plot::Tetrahedron`, `plot::Tube`, `plot::XRotate`, `plot::ZRotate`	`FillColor`: `RGB::Red`
	
	`FillColor2`: `RGB::CornflowerBlue`

`plot::Box`, `plot::Circle3d`, `plot::Cone`, `plot::Cylinder`, `plot::Ellipsoid`, `plot::Plane`, `plot::Polygon3d`, `plot::Sphere`	`FillColor`: `RGB::LightBlue`
`plot::Arc2d`, `plot::Circle2d`, `plot::Ellipse2d`, `plot::Hatch`, `plot::Parallelogram2d`, `plot::Polygon2d`, `plot::Rectangle`, `plot::Sum`	`FillColor`: `RGB::Red`
`plot::Arc3d`, `plot::Ellipse3d`, `plot::Parallelogram3d`	`FillColor`: `RGB::LightBlue`

Objects	Default Values
	`FillColor2`: `RGB::CornflowerBlue`

`plot::Waterman`	`FillColor`: `RGB::SafetyOrange`
	
	`FillColor2`: `RGB::CornflowerBlue`

`plot::Integral`	`FillColor`: `RGB::PaleBlue`

Description

`FillColor` determines the color used to fill all types of areas and surfaces. `FillColor2` is used for color blends. `FillColors` is used for objects that need more than one color.

2D objects that have a notion of "area" and 3D objects that have a surface support `FillColor` to determine the primary color to show objects in. If `FillColorType` is set to `Dichromatic`, `FillColor2` sets the second color to blend to.

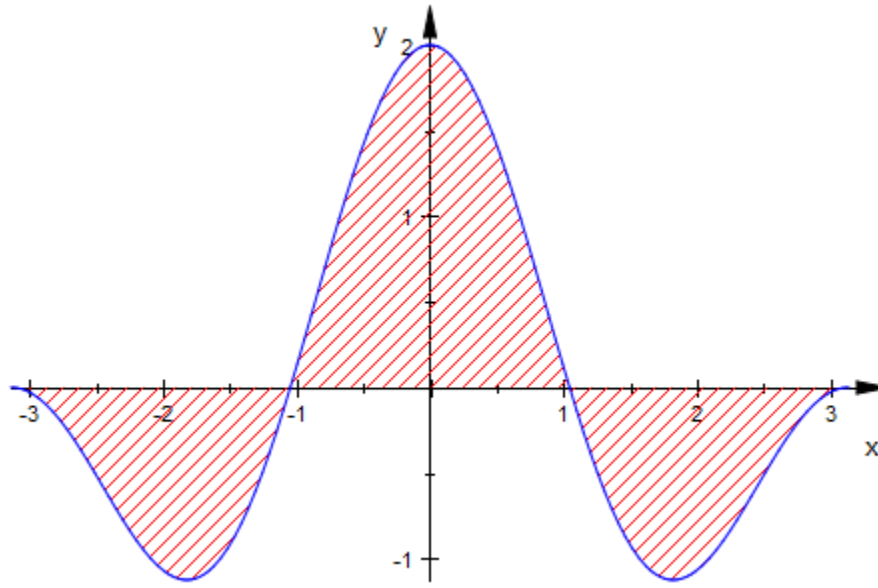
Functions and primitives displaying more than one object, such as `plot::Bars2d`, use `FillColors` for a list of colors used cyclically.

Examples

Example 1

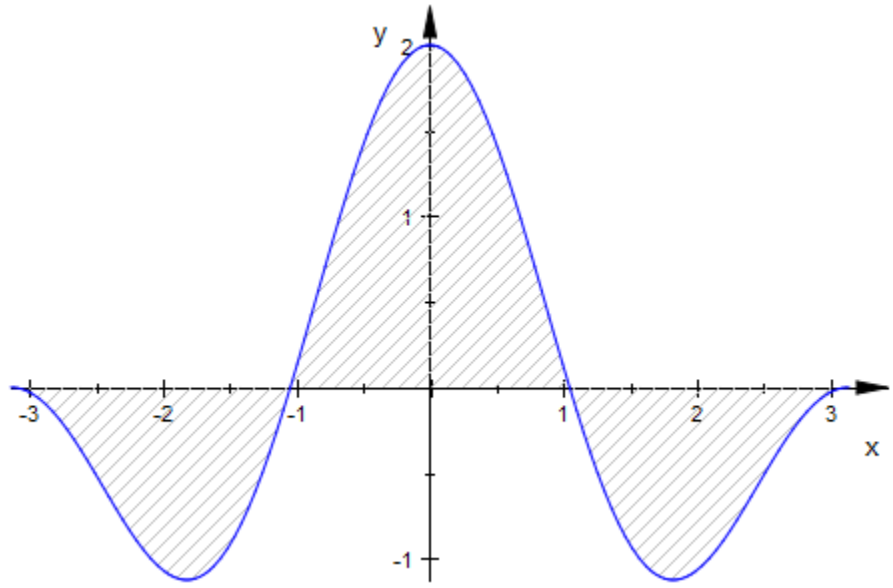
By default, `plot::Hatch` objects are hatched in `RGB ::Red`, the same color used by default for `plot::Function2d`:

```
f := plot::Function2d(cos(2*x)+cos(x), x=-PI..PI):
h := plot::Hatch(f):
plot(h, f)
```



To change the color of the hatch, simply set the `"FillColor"`-slot to some other value:

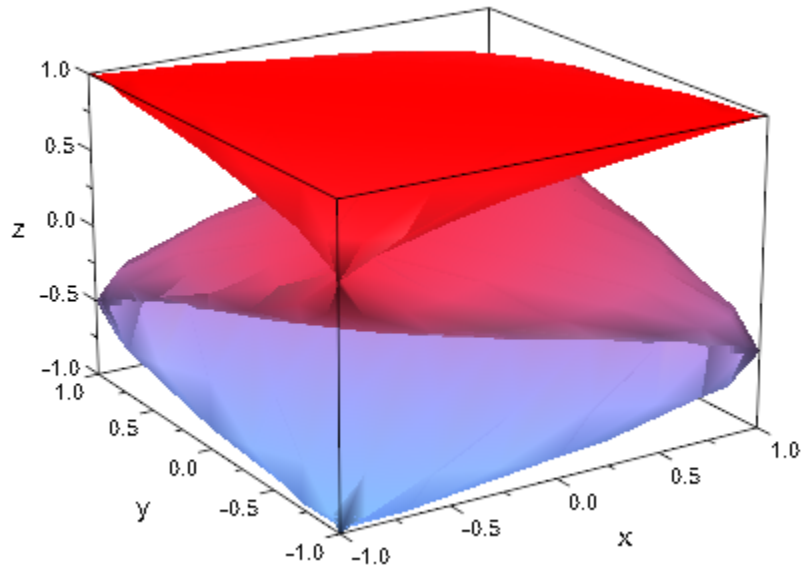
```
h::FillColor := RGB::Grey:
plot(h, f)
```



Example 2

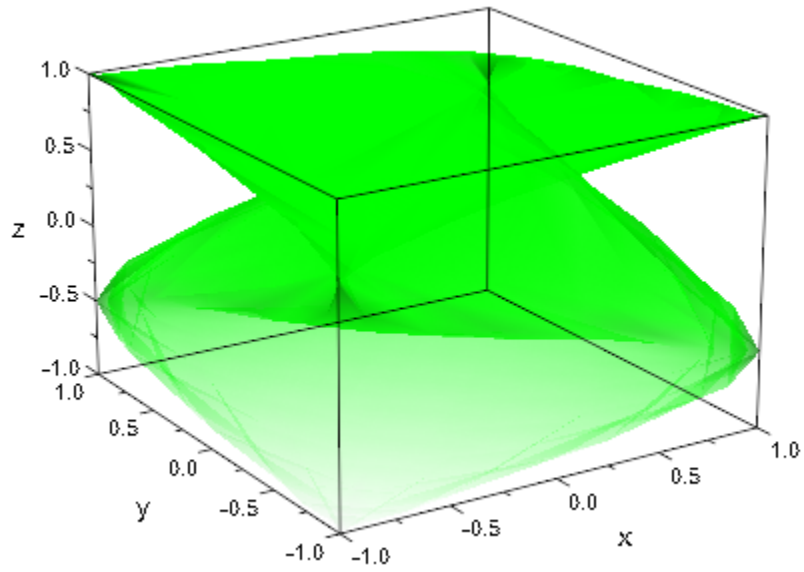
The default setting for a surface is to have a height-dependent coloring with a linear blend from `FillColor` to `FillColor2`:

```
s := plot::Surface([cos(2*u+v), sin(u+2*v), sin(u+v)],
                   u = 0..2*PI, v = 0..2*PI,
                   ULinesVisible = FALSE,
                   VLinesVisible = FALSE):
plot(s)
```



These colors can be manipulated in the usual way. As an example, we set the transition to a monochrome transition from opaque to transparent:

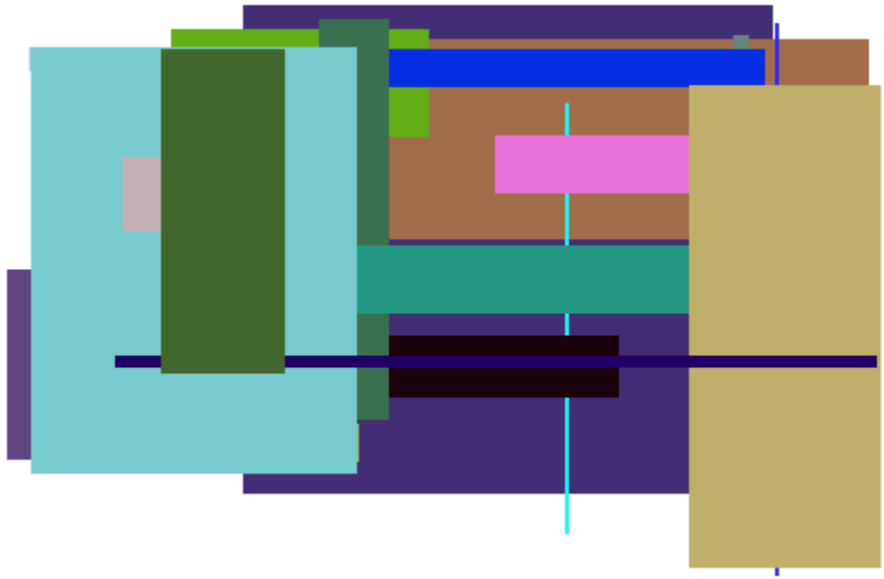
```
s::FillColor  := RGB::Green:
s::FillColor2 := s::FillColor . [0.0]:
plot(s)
```



Example 3

Using a utility function `randrange` that returns random ranges in [0, 1], we can plot random rectangles with random colors:

```
randrange := () -> _range(op(sort([frandom(), frandom()]))):
plot(plot::Rectangle(randrange(), randrange(),
                     LinesVisible = FALSE,
                     Filled = TRUE, FillPattern = Solid,
                     FillColor = [frandom(), frandom(), frandom()])
     $k=1..20,
     AxesVisible = FALSE)
```



See Also

MuPAD Functions

`FillColorType` | `Filled` | `FillPattern` | `LineColor` | `Shading`

FillColorDirection, FillColorDirectionX, FillColorDirectionY, FillColorDirectionZ

Direction of color transitions on surfaces

Value Summary

`FillColorDirection`	Library wrapper for "[`FillColorDirectionX`, `FillColorDirectionY`]" (2D), "[`FillColorDirectionX`, `FillColorDirectionY`, `FillColorDirectionZ`]" (3D)	See below
`FillColorDirectionX`, `FillColorDirectionY`, `FillColorDirectionZ`	Inherited	Real number

Graphics Primitives

Objects	Default Values
`plot::Arc3d`, `plot::Cylindrical`, `plot::Dodecahedron`, `plot::Ellipse3d`, `plot::Function3d`, `plot::Hexahedron`, `plot::Icosahedron`, `plot::Implicit3d`, `plot::Matrixplot`, `plot::Octahedron`, `plot::Parallelogram3d`, `plot::Prism`, `plot::Pyramid`, `plot::Spherical`, `plot::Surface`, `plot::SurfaceSet`, `plot::SurfaceSTL`, `plot::Sweep`, `plot::Tetrahedron`, `plot::Tube`, `plot::Waterman`, `plot::XRotate`, `plot::ZRotate`	`FillColorDirection`: [0, 0, 1] `FillColorDirectionX`, `FillColorDirectionY`: 0 `FillColorDirectionZ`: 1
`plot::Listplot`	`FillColorDirection`: [0, 0]

Description

`FillColorDirection` determines the direction in which the color transitions for `FillColorType = Dichromatic` etc. take place.

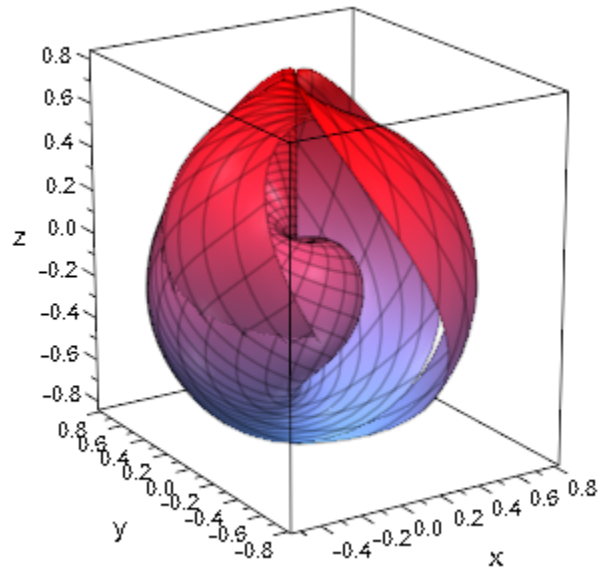
When setting `FillColorType` to some other value than `Flat` or `Functional`, MuPAD produces a "height-coloring." By default, this color method actually uses the height of a point. Using `FillColorDirection`, the axis along which the color method should be applied can be changed.

Examples

Example 1

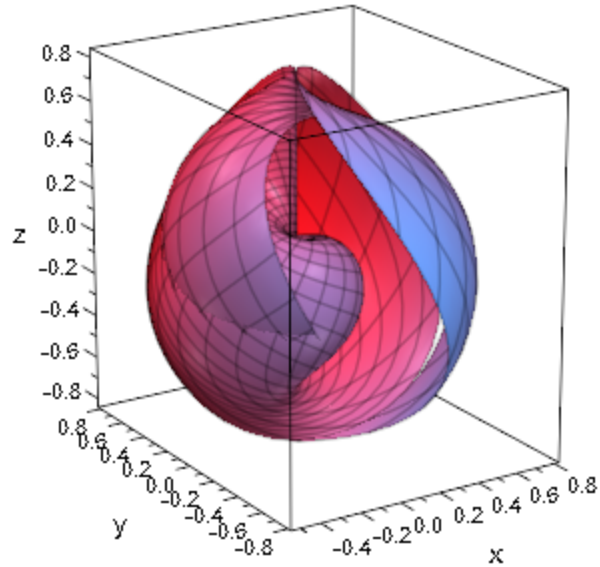
By default, MuPAD uses height coloring along the *z* axis for 3D objects:

```
s := plot::Spherical([sin(r),thet/(r+1)+1, thet*r],
                     r=0..1, thet=0..3*PI, Submesh=[2,2]):
plot(s)
```



By changing `FillColorDirection`, the color can be rotated on the object:

```
plot(s, FillColorDirection = [0, 1, 0])
```



See Also

MuPAD Functions

`FillColor` | `FillColor2` | `FillColorType` | `LineColorDirection`

FillColorTrue, FillColorFalse, FillColorUnknown

Color for “true” areas (inequality plot)

Value Summary

`FillColorFalse`, `FillColorTrue`, `FillColorUnknown`	Optional	Color

Graphics Primitives

Objects	Default Values

`plot::Inequality`	`FillColorTrue`: `RGB::Green`
	
	`FillColorFalse`: `RGB::Red`
	
	`FillColorUnknown`: `RGB::Black`

Description

`FillColorTrue`, `FillColorFalse`, and `FillColorUnknown` define the three colors use by `plot::Inequality` for the areas where the inequalities are fulfilled (true), violated (false) or the granularity is too small to decide (unknown).

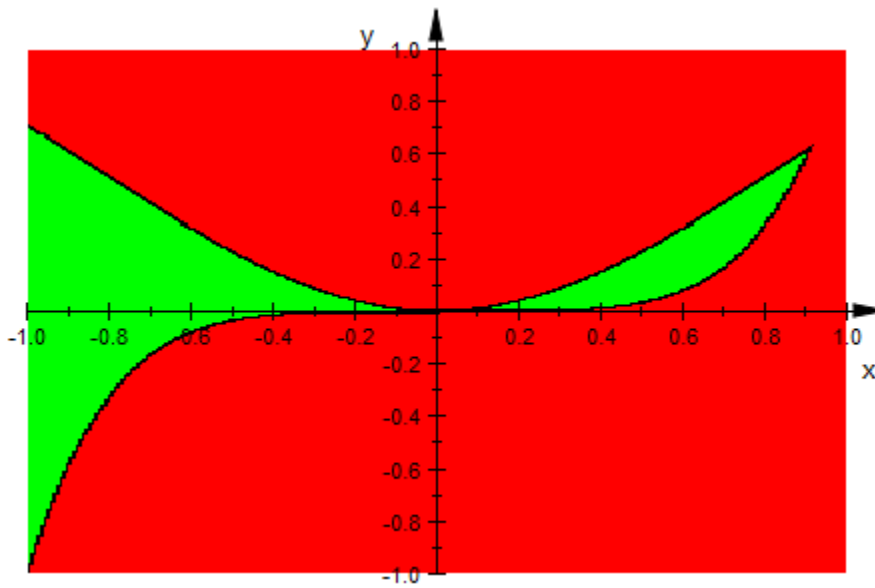
`plot::Inequality` divides the plot area into rectangles that are colored according to these three attributes. Rectangles over which the inequalities are true get the color set by `FillColorTrue`; rectangles over which at least one inequality is violated (i.e., false over the whole rectangle) use `FillColorFalse`. If neither of these two apply and the rectangle is already too small for subdivision (the settings for `XMesh` and `YMesh` control this), it will be painted in `FillColorUnknown`.

Examples

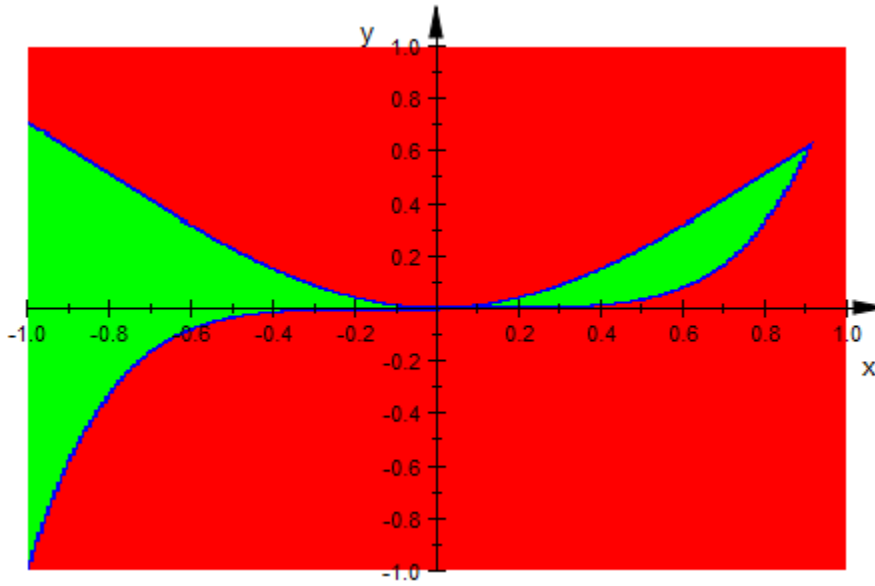
Example 1

We show the same inequality plot with different settings of these three attributes:

```
ineq := plot::Inequality([sin(x)^2>y, y>x^5], x=-1..1, y=-1..1):
plot(ineq)
```




```
ineq::FillColorTrue := RGB::Green:
ineq::FillColorFalse := RGB::Red:
ineq::FillColorUnknown := RGB::Blue:
plot(ineq)
```


y
1.0
0.8
0.6
0.4
0.2
-1.0
-0.8
-0.6
-0.4
-0.2
0.2
0.4
0.6
0.8
1.0
x
-0.2
-0.4
-0.6
-0.8
-1.0

FillColorType

Surface filling types

Value Summary

Inherited	`Dichromatic`, `Flat`, `Functional`, `Monochrome`, or `Rainbow`

Graphics Primitives

Objects	FillColorType Default Values
`plot::Cylindrical`, `plot::Density`, `plot::Dodecahedron`, `plot::Function3d`, `plot::Hexahedron`, `plot::Icosahedron`, `plot::Implicit3d`, `plot::Matrixplot`, `plot::Octahedron`, `plot::Prism`, `plot::Pyramid`, `plot::Spherical`, `plot::Surface`, `plot::SurfaceSet`, `plot::SurfaceSTL`, `plot::Sweep`, `plot::Tetrahedron`, `plot::Tube`, `plot::XRotate`, `plot::ZRotate`	`Dichromatic`
`plot::Arc3d`, `plot::Ellipse3d`, `plot::Parallelogram3d`, `plot::Waterman`	`Flat`

Description

`FillColorType` selects the type of surface fill color used.

With the exception of `Flat` and `Functional`, the coloring schemes depend on the height, i.e., the *z* value of points on the surface, in relation to the height of the whole

coordinate system. (Everything on this page relating to surfaces holds for objects of type `plot::Density`, too, with the values plotted replacing height information.)

By default, surfaces are drawn with a linear blend from `FillColor` to `FillColor2`. This behavior may be changed with `FillColorType`, using one of the following options:

- `Dichromatic`

  The default just described.
- `Flat`

  The surface is filled with `FillColor`. No blend is used.
- `Monochrome`

  The surface is filled with a blend from `FillColor` to a dimmed version of `FillColor`.
- `Rainbow`

  This setting is technically similar to `Dichromatic`, but the effect is vastly different, since interpolation takes place in HSV color space. This creates a rainbow effect, similar to a physical rainbow for suitable choices of colors.
- `Functional`

  Both `FillColor` and `FillColor2` are ignored; the color scheme is derived from `FillColorFunction`. See `FillColorFunction` for details (which depend on the object type). If no color function is given, the object will be rendered with `FillColorType` = `Flat`.

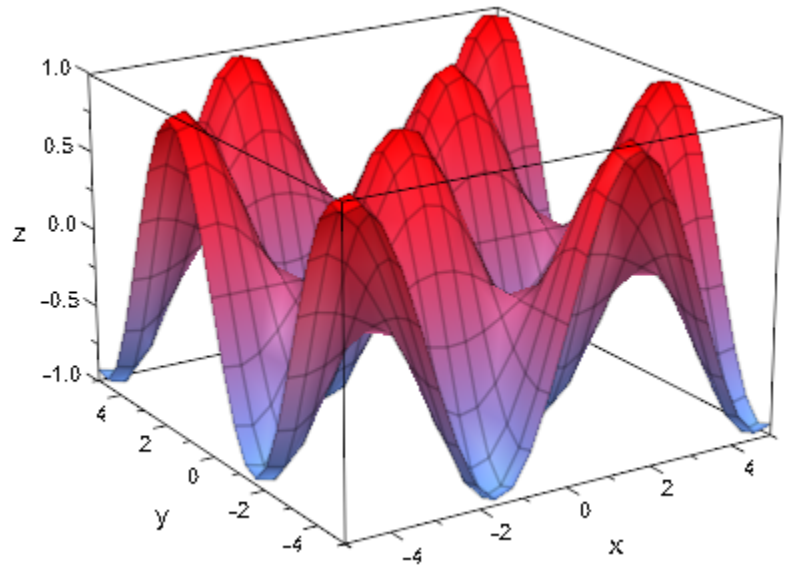
In this context, "a blend from *A* to *B*" means that color *A* is used at the top of the coordinate system (the part with the lowest *z* coordinate), color *B* is used at the bottom and in between each or the red, green, blues, and alpha channel are interpolated linearly.

Examples

Example 1

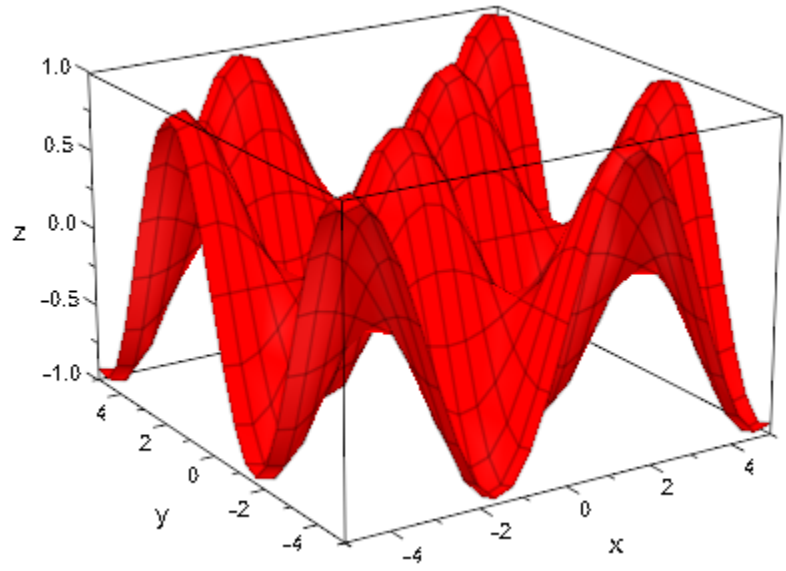
By default, function plots use `FillColorType = Dichromatic` with a color range from blue to red (as in a temperature scale):

```
plotfunc3d(sin(x)*sin(y))
```



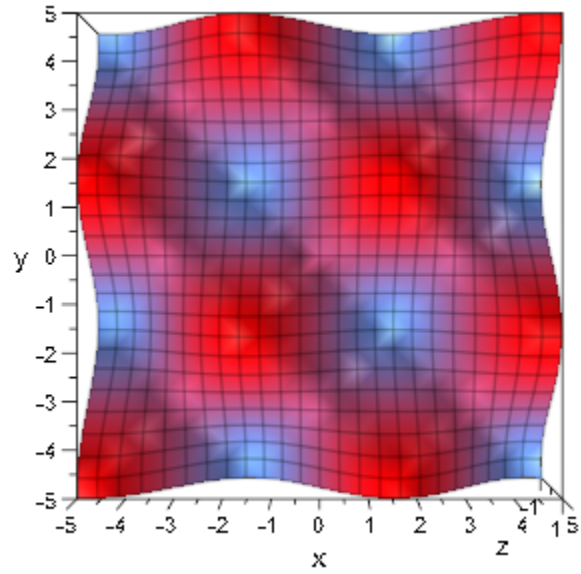
Using `FillColorType`, we color the graph completely in red:

```
plotfunc3d(sin(x)*sin(y), FillColorType = Flat)
```

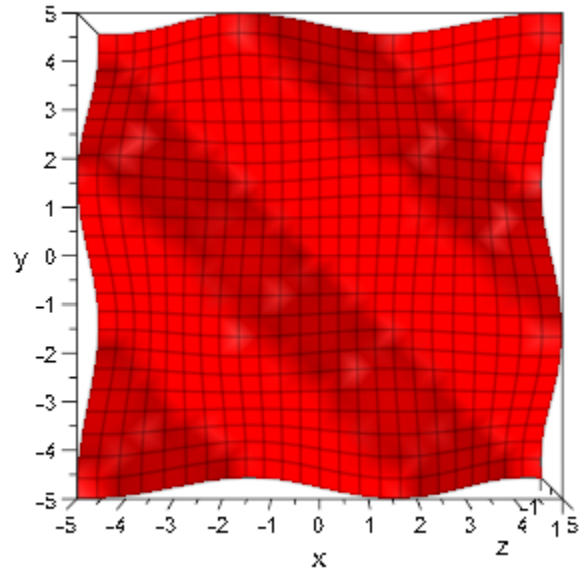


Note, however, that the coloring is a visual aid, e.g., when looking from above:

```
plotfunc3d(sin(x)*sin(y),
           CameraDirection = [0, 0, 1])
```



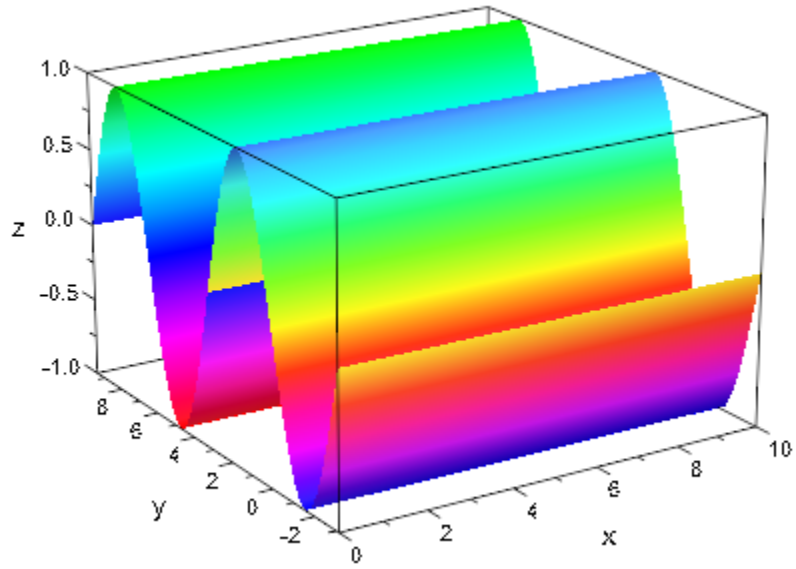

```
plotfunc3d(sin(x)*sin(y), FillColorType = Flat,
           CameraDirection = [0, 0, 1])
```



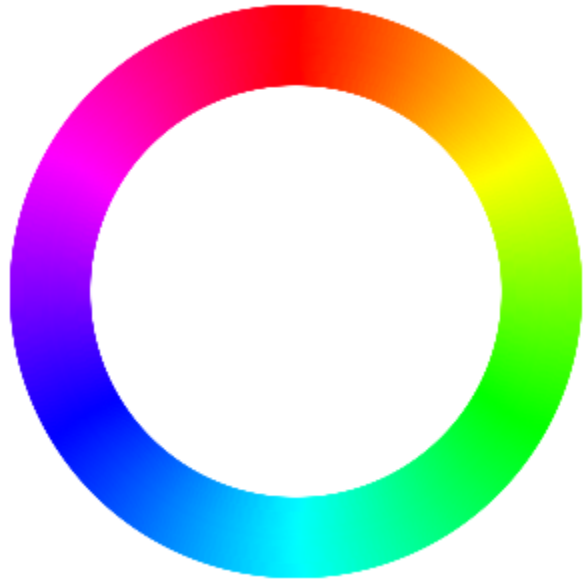
Example 2

In MuPAD, rainbow coloring does react to `FillColor` and `FillColor2`. The following plot uses different color settings to show this effect:

```
plot(plot::Function3d(sin(y), x = 0..10, y = -PI..PI,
                      FillColor = RGB::BlueLight,
                      FillColor2 = RGB::Blue),
     plot::Function3d(sin(y), x = 0..10, y = PI..3*PI,
                      FillColor = RGB::Green,
                      FillColor2 = RGB::Red),
     FillColorType = Rainbow,
     XLinesVisible = FALSE, YLinesVisible = FALSE)
```

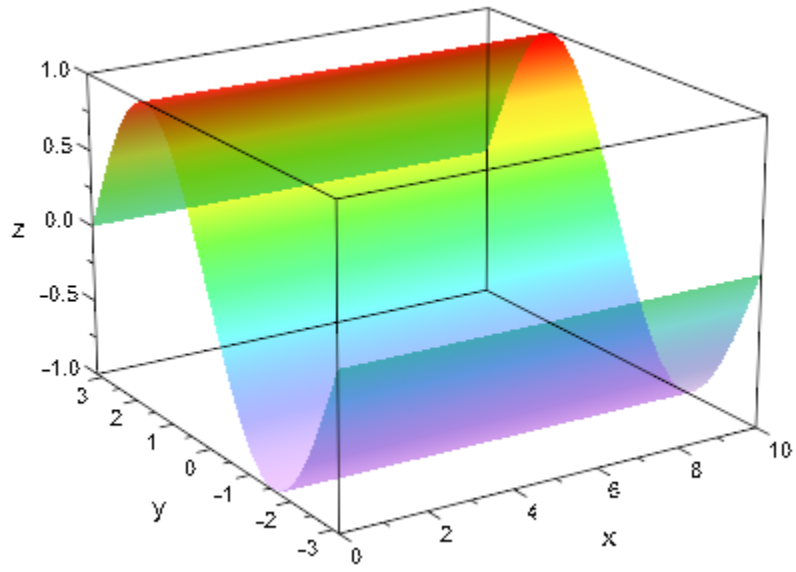


What is happening here technically is that MuPAD performs a linear interpolation in HSV color space, i.e., the *longest* path round the following color circle is followed, with saturation and value (roughly speaking, whiteness and blackness) interpolated linearly:



The opacity of colors is treated the same way in both the `Rainbow` and `Dichromatic` settings of `FillColorType`, by linear interpolation:

```
plot((f:=plot::Function3d(sin(y), x = 0..10, y = -PI..PI,
     FillColorType = Rainbow, FillColor2 = RGB::VioletDark.[0.2],
     XLinesVisible = FALSE, YLinesVisible = FALSE)))
```



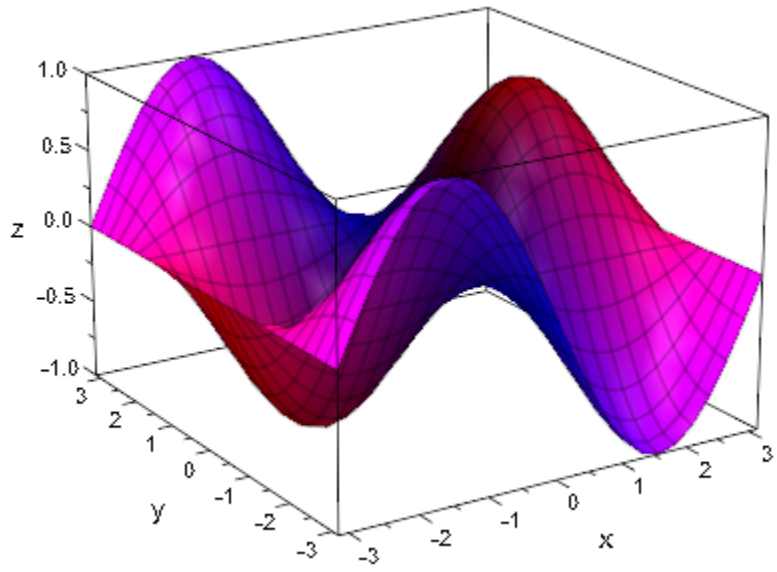
Example 3

Setting a `FillColorFunction` for an object automatically sets `FillColorType` to `Functional`:

```
colorfunc := (x, y) -> [abs(x)/PI, 0, abs(y)/PI]:
f := plot::Function3d(sin(x)*cos(y), x = -PI..PI, y = -PI..PI,
                      FillColorFunction = colorfunc):
f::FillColorType
```

```
Functional
```

```
plot(f)
```




```
delete colorfunc, f:
```

See Also

MuPAD Functions

`FillColor` | `FillColor2` | `Filled` | `FillPattern` | `LineColorType` | `Shading`

Filled

Filled or transparent areas and surfaces

Value Summary

Inherited	FALSE, or TRUE

Graphics Primitives

Objects	Filled Default Values
plot::Bars2d, plot::Bars3d, plot::Box, plot::Boxplot, plot::Cone, plot::Cylinder, plot::Cylindrical, plot::Dodecahedron, plot::Function3d, plot::Hexahedron, plot::Histogram2d, plot::Icosahedron, plot::Implicit3d, plot::Integral, plot::Matrixplot, plot::Octahedron, plot::Parallelogram3d, plot::Piechart2d, plot::Piechart3d, plot::Plane, plot::Prism, plot::Pyramid, plot::Spherical, plot::Surface, plot::SurfaceSet, plot::SurfaceSTL, plot::Sweep, plot::Tetrahedron, plot::Tube, plot::Waterman, plot::XRotate, plot::ZRotate	TRUE
plot::Arc2d, plot::Arc3d, plot::Circle2d, plot::Circle3d, plot::Ellipse2d, plot::Ellipse3d, plot::Parallelogram2d, plot::Polygon2d, plot::Polygon3d, plot::Rectangle, plot::Sum	FALSE

Description

`Filled` controls whether areas and surfaces are filled or transparent.

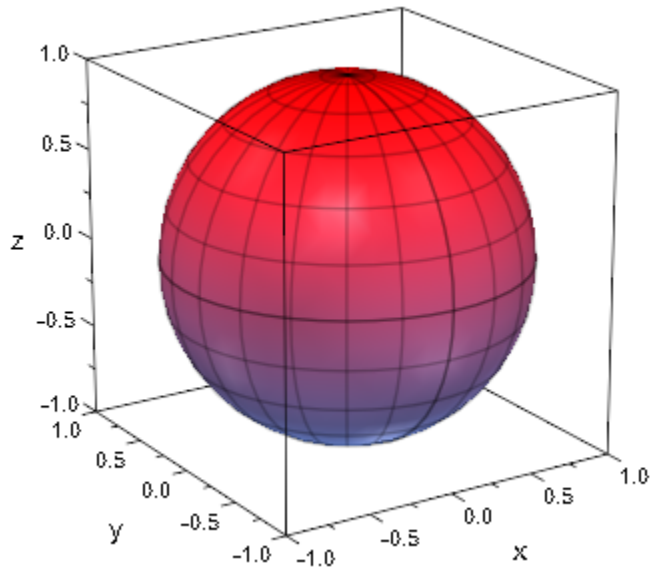
With `Filled = FALSE`, areas and surfaces are not filled. This means that, e.g., a surface plot is reduced to a wire frame model.

Examples

Example 1

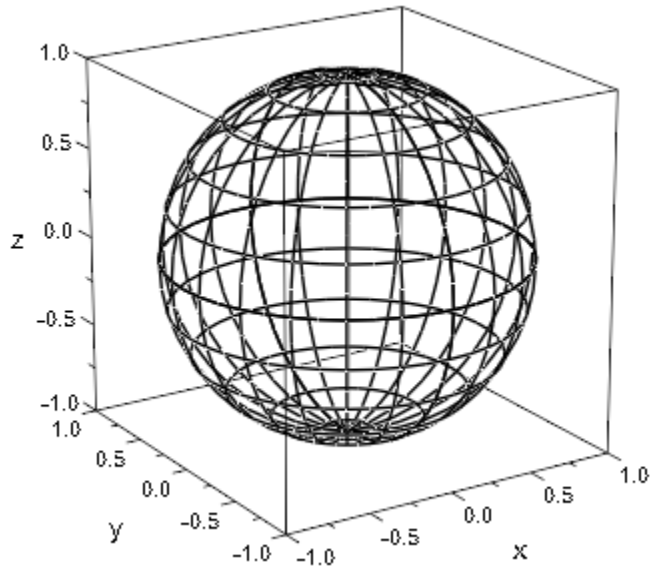
The following parametrization of a sphere uses a mesh similar to the graticule (longitudes and latitudes) of geography:

```
globe := plot::Surface([sin(u)*cos(v), cos(u)*cos(v), sin(v)],
                       u = 0..2*PI, v = 0..2*PI,
                       Mesh = [12, 12], Submesh = [3, 3]):
plot(globe, Scaling = Constrained)
```



To get a wire frame model, we set `Filled = FALSE`:

```
plot(globe, Filled = FALSE, LineColor = RGB::Black,
     Scaling = Constrained)
```



See Also

MuPAD Functions

`Colors` | `FillColor` | `FillColor2` | `FillColorType` | `FillPattern` | `FillStyle` | `LinesVisible` | `ULinesVisible` | `VLinesVisible`

FillPattern, FillPatterns

Type of area filling

Value Summary

`FillPattern`	Inherited	`CrossedLines`, `DiagonalLines`, `FDiagonalLines`, `HorizontalLines`, `Solid`, `VerticalLines`, or `XCrossedLines`
`FillPatterns`	Optional	`Solid`, `HorizontalLines`, `VerticalLines`, `DiagonalLines`, `FDiagonalLines`, `CrossedLines`, or `XCrossedLines`

Graphics Primitives

Objects	Default Values
`plot::Arc2d`, `plot::Boxplot`, `plot::Circle2d`, `plot::Ellipse2d`, `plot::Hatch`, `plot::Parallelogram2d`, `plot::Polygon2d`, `plot::Rectangle`	`FillPattern`: `DiagonalLines`
`plot::Bars2d`, `plot::Histogram2d`, `plot::Inequality`, `plot::Integral`, `plot::Piechart2d`, `plot::Sum`	`FillPattern`, `FillPatterns`: `Solid`

Description

`FillPattern` determines the style of area filling used: lines, grids, or a solid fill.

`FillPatterns` is used for objects with more than one type of area to fill.

Areas can be filled in various ways. You can have horizontal, vertical, or diagonal lines (`HorizontalLines`, `VerticalLines`, `DiagonalLines`, `FDiagonalLines`), a horizontal/vertical grid (`CrossedLines`), a diagonal grid (`XCrossedLines`), or a solid fill (`Solid`).

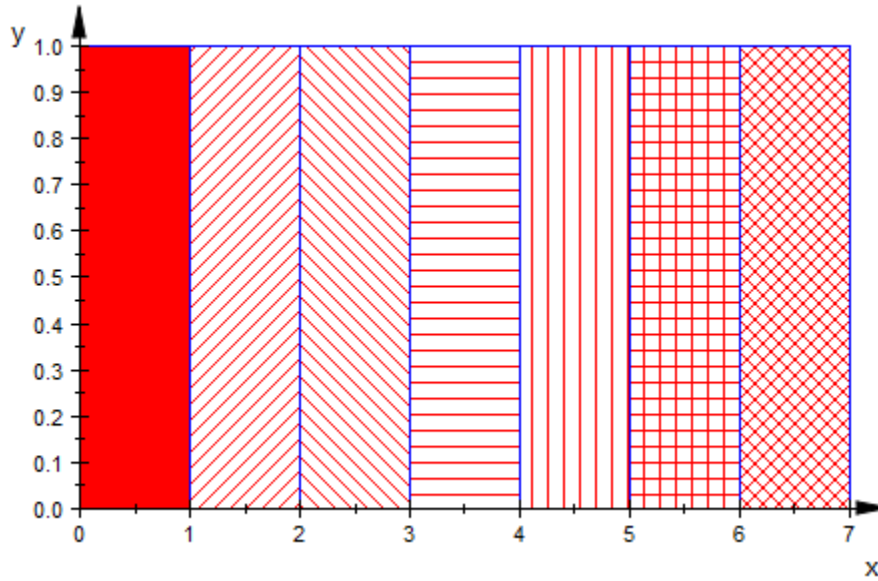
For types like `plot::Bars2d`, `FillPatterns` is a list of fill patterns used cyclically, in this case for the groups of data plotted.

Examples

Example 1

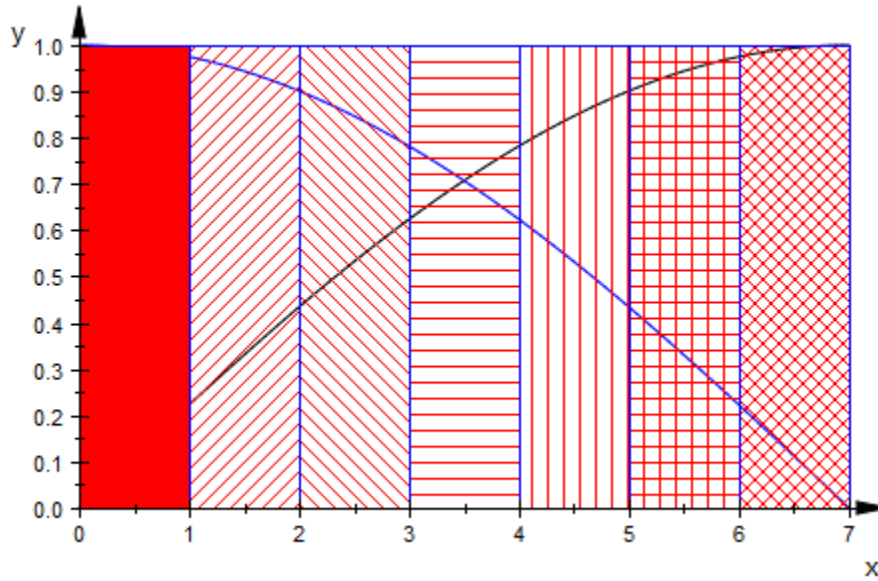
The fill patterns look like this:

```
plot(
 plot::Rectangle(0..1, 0..1, FillPattern = Solid),
 plot::Rectangle(1..2, 0..1, FillPattern = DiagonalLines),
 plot::Rectangle(2..3, 0..1, FillPattern = FDiagonalLines),
 plot::Rectangle(3..4, 0..1, FillPattern = HorizontalLines),
 plot::Rectangle(4..5, 0..1, FillPattern = VerticalLines),
 plot::Rectangle(5..6, 0..1, FillPattern = CrossedLines),
 plot::Rectangle(6..7, 0..1, FillPattern = XCrossedLines),
 Filled = TRUE, AxesInFront = TRUE
)
```



Except for `Solid`, the fill patterns let objects below be seen:

```
plot(
 plot::Function2d(sin(x*PI/14), x = 0..7, Color = RGB::Black),
 plot::Function2d(cos(x*PI/14), x = 0..7, Color = RGB::Blue),
 plot::Rectangle(0..1, 0..1, FillPattern = Solid),
 plot::Rectangle(1..2, 0..1, FillPattern = DiagonalLines),
 plot::Rectangle(2..3, 0..1, FillPattern = FDiagonalLines),
 plot::Rectangle(3..4, 0..1, FillPattern = HorizontalLines),
 plot::Rectangle(4..5, 0..1, FillPattern = VerticalLines),
 plot::Rectangle(5..6, 0..1, FillPattern = CrossedLines),
 plot::Rectangle(6..7, 0..1, FillPattern = XCrossedLines),
 Filled = TRUE, AxesInFront = TRUE
)
```



See Also

MuPAD Functions

`Color` | `Colors` | `FillColor` | `FillColorType` | `Filled`

FillStyle

Definition of inside/outside

Value Summary

Inherited	`EvenOdd`, or `Winding`

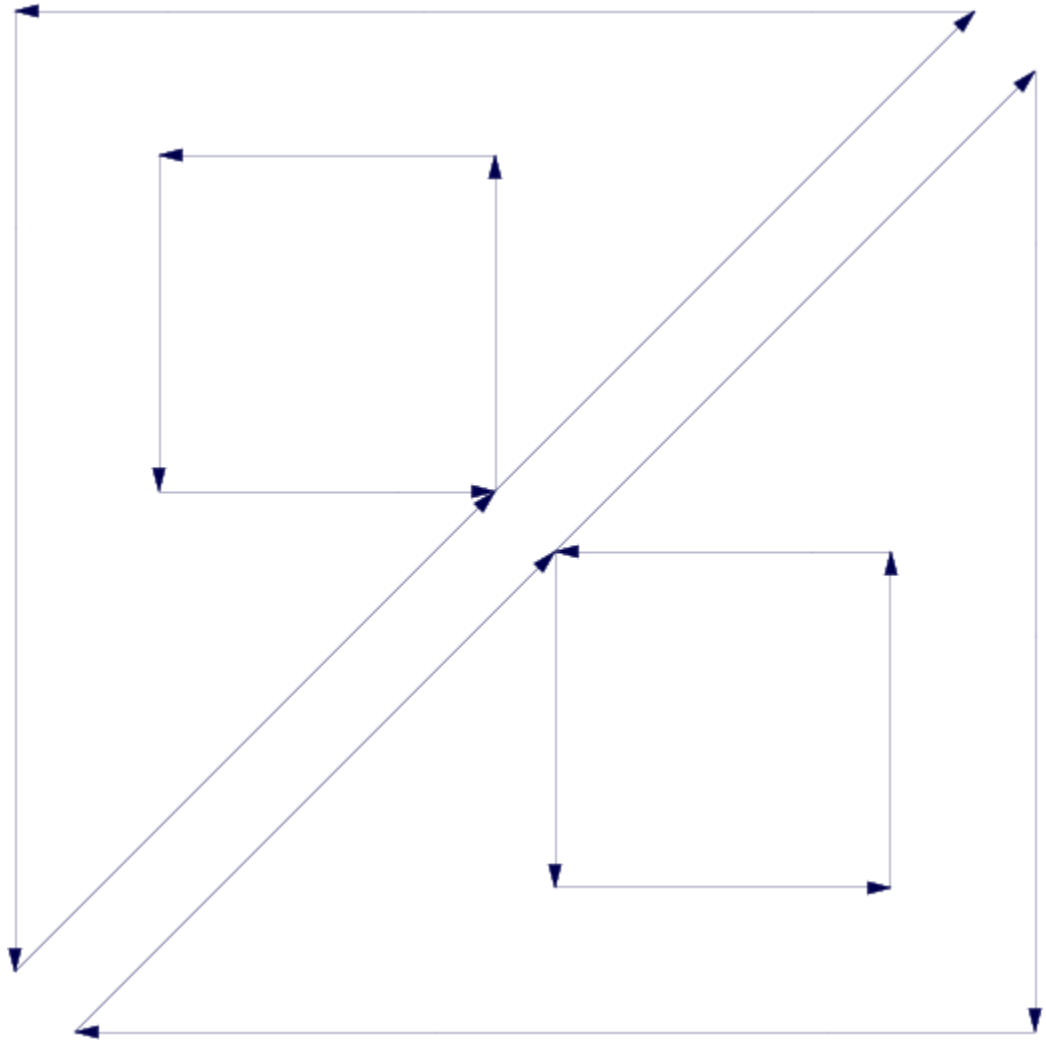
Graphics Primitives

Objects	FillStyle Default Values
`plot::Polygon2d`	`EvenOdd`

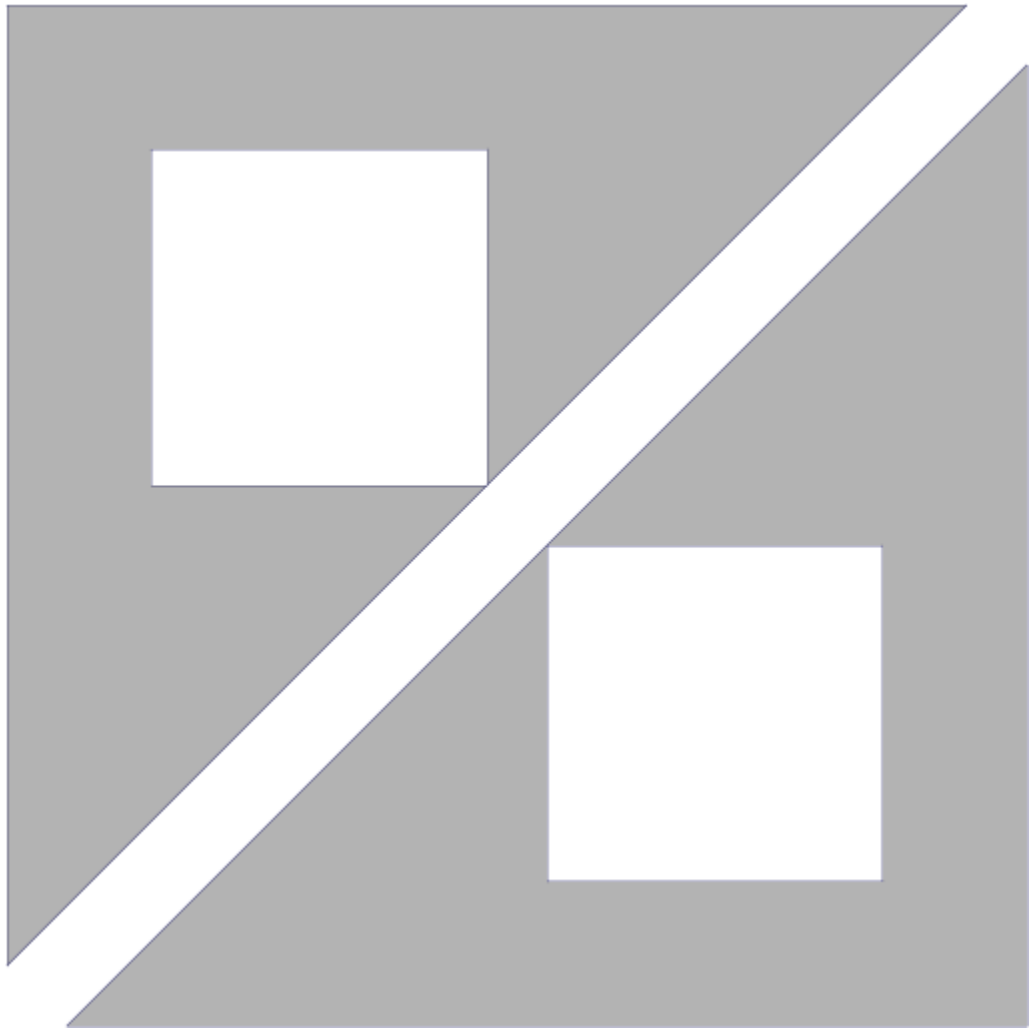
Description

For self-intersecting closed curves, `FillStyle` determines how holes are detected/defined.

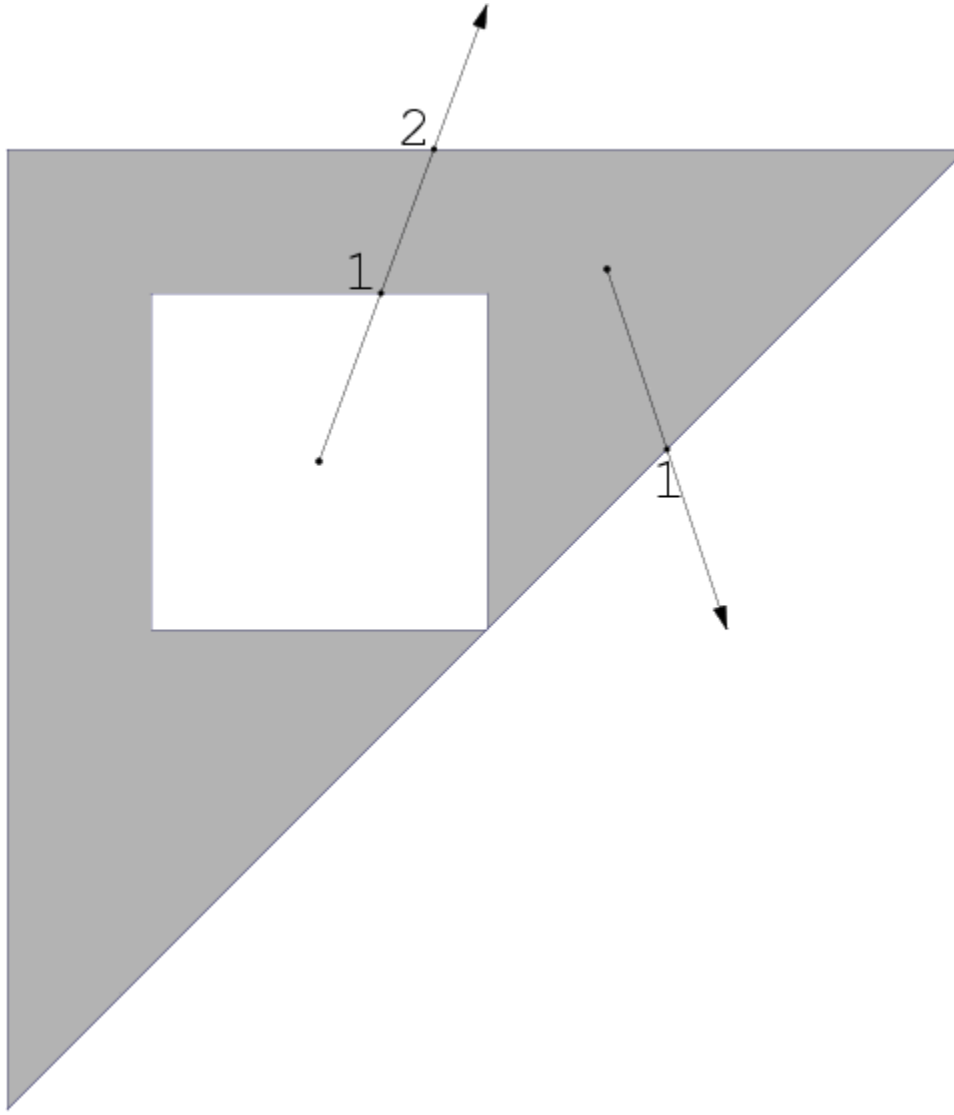
Closed curves have an inside and an outside. With self-intersecting curves, the inside may have holes which are considered outside and not filled. To explain the difference between `EvenOdd` and `Winding`, we use the following two polygons which differ only in the order the inner points are visited:



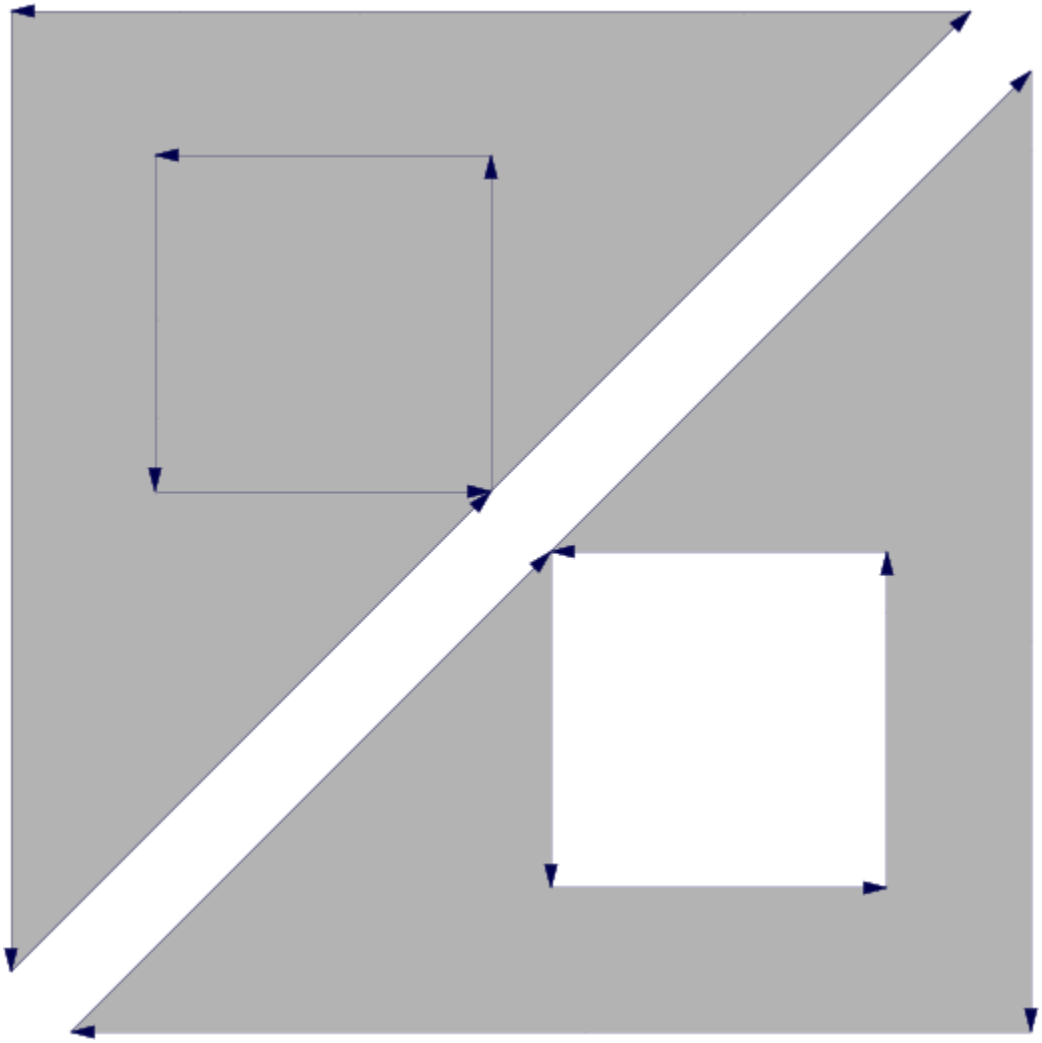
If plotted with `FillStyle = EvenOdd`, there is no difference between the two:



This is because for `FillStyle = EvenOdd`, a point is considered "inside" if a ray starting from the point and extending to infinity has an *odd* number of intersections with the polygon:

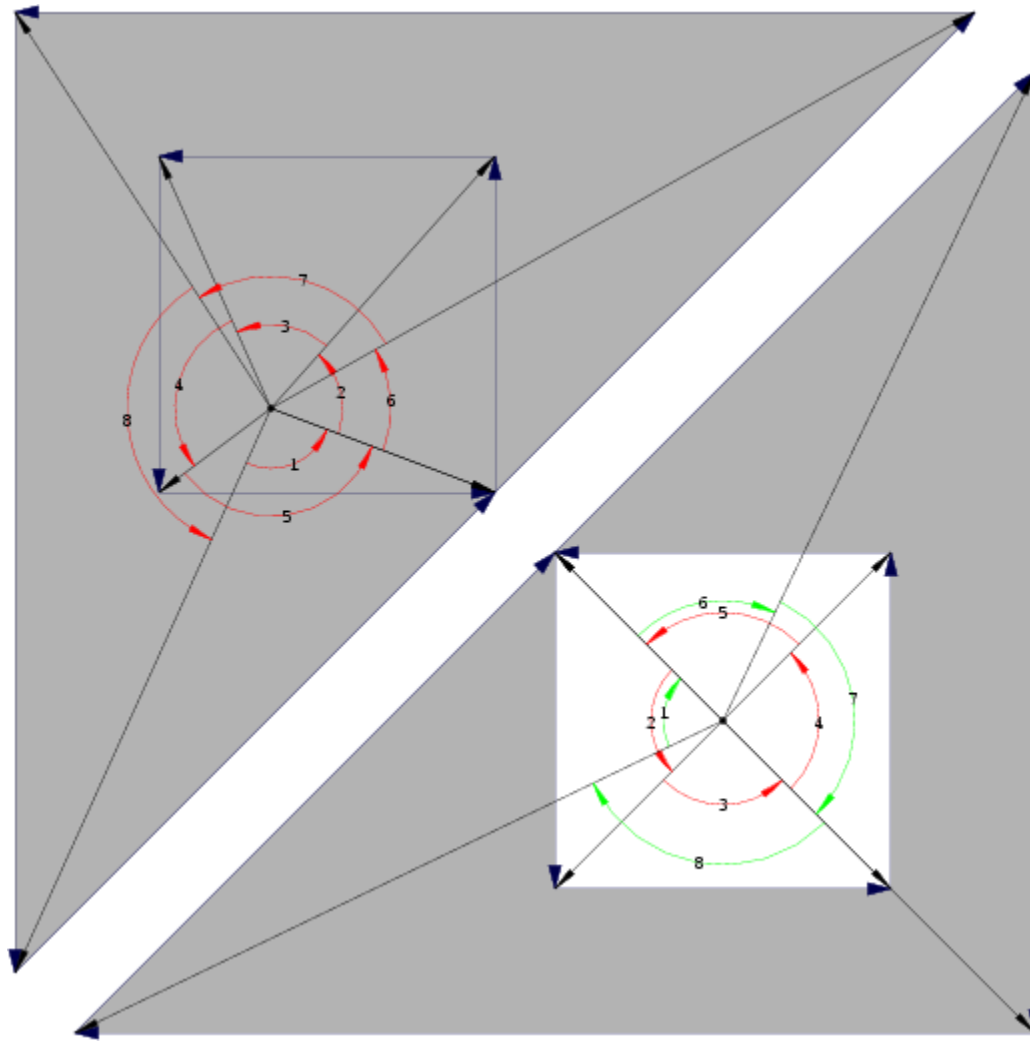


With `FillStyle = Winding`, however, the triangles look different from one another:

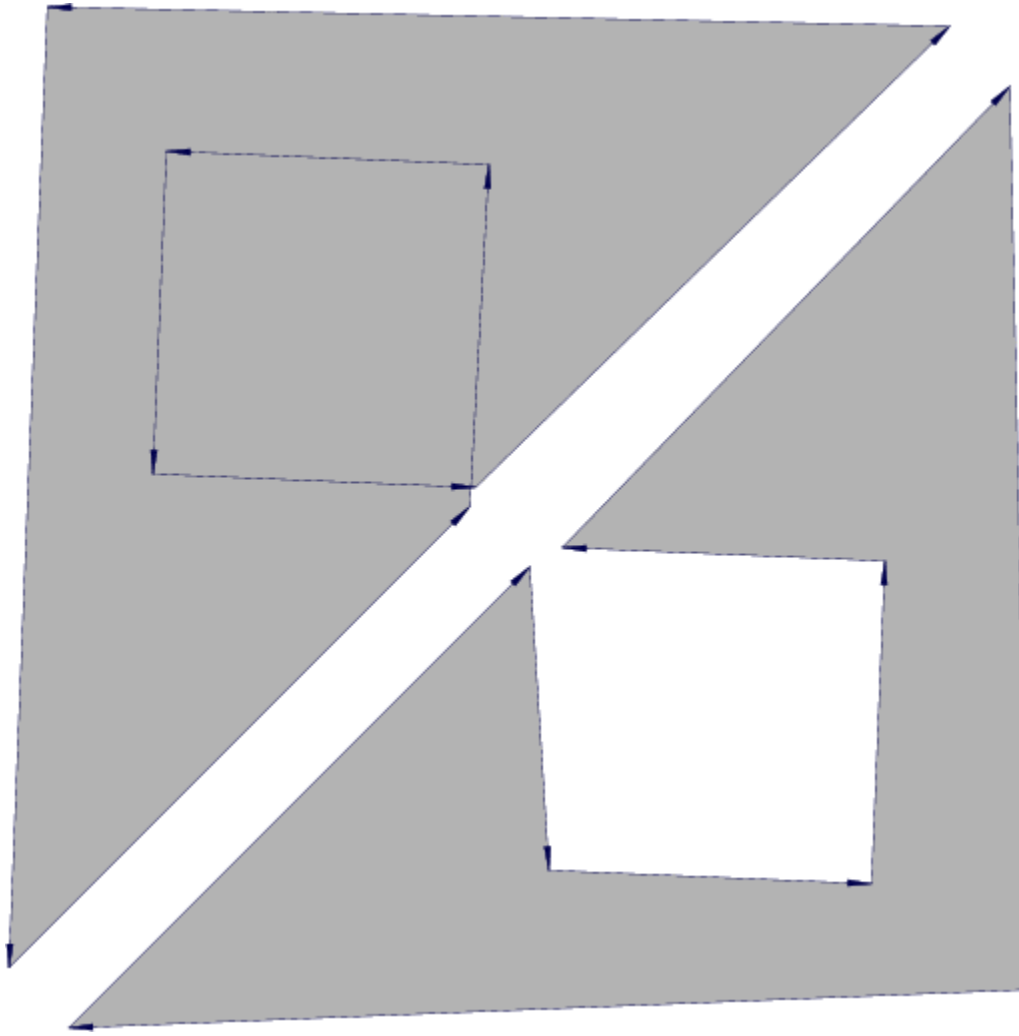


For `FillStyle = Winding`, the *winding number* of a point must be nonzero for this point to be “inside”. The winding number is the number of times the polygon line actually “runs around” the point. It can be determined by sequentially looking at all the edges, summing up the angles under which neighboring edges are seen (take care of the sign of he angle!) and dividing by 2 π. In our example, a point in the square in the upper triangle

has a winding number of 2, while one in the square in the lower triangle has a winding number of 0:



`FillStyle = Winding` is similar to a complete filling of the polygon area, but it is stable under small displacements of the polygon points:



See Also

MuPAD Functions
`Filled`

GroupStyle

Grouping options in 2D bar plots

Value Summary

Optional	`MultipleBars`, or `SingleBars`

Graphics Primitives

Objects	GroupStyle Default Values
`plot::Bars2d`	`MultipleBars`

Description

`GroupStyle` determines whether a bar plot visualizes the data of different groups by separate bars or by single bars that are split into colored regions.

2D `bar plots` can group the bars in various ways. With the default setting `GroupStyle` = `MultipleBars`, data that are split into several groups are displayed by separate bars for each value in each group. With `GroupStyle` = `SingleBars`, corresponding data items in different groups are stacked up to one single bar. It is split into differently colored parts that correspond to the different groups.

With `SingleBars`, all data must be nonnegative.

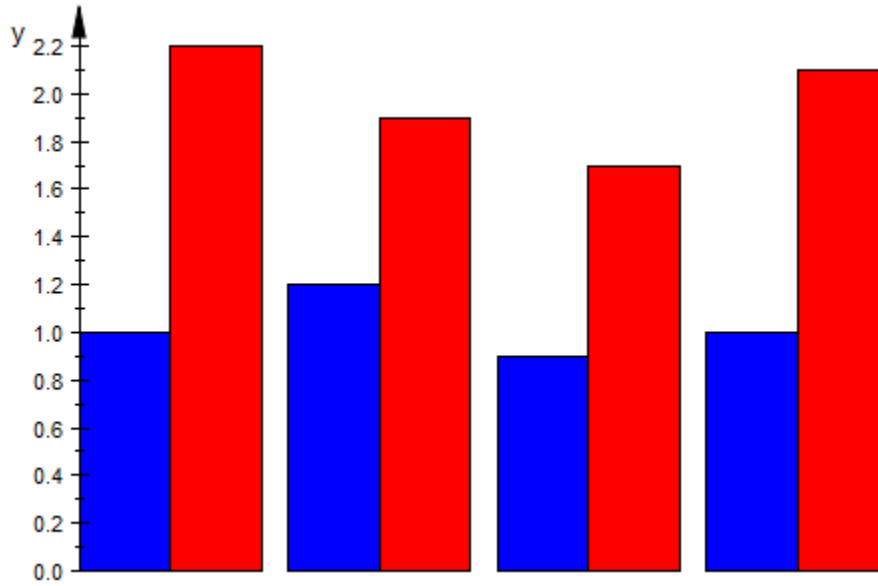
`SingleBars` has no effect if the data of only one group are given. If you wish to visualize the data in a single bar, you have to turn each data item into a separate group. Cf. “Example 2” on page 24-2073.

Examples

Example 1

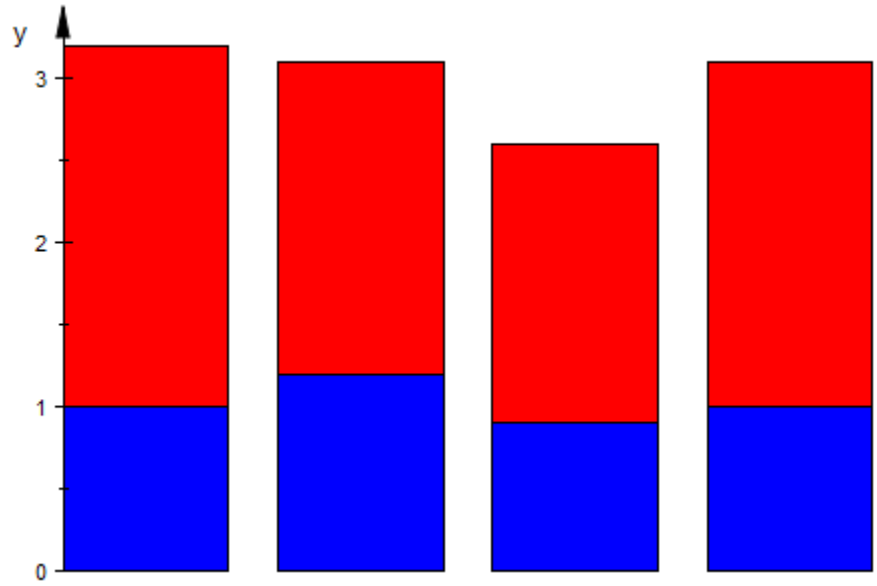
The attribute `GroupStyle` can have the values `MultipleBars` or `SingleBars`:

```
group1 := [1.0, 1.2, 0.9, 1.0]:
group2 := [2.2, 1.9, 1.7, 2.1]:
data:= [group1, group2]:
plot(plot::Bars2d(data, GroupStyle = MultipleBars))
```



With `SingleBars`, corresponding data items in the different groups are collected in a single bar:

```
plot(plot::Bars2d(data, GroupStyle = SingleBars))
```

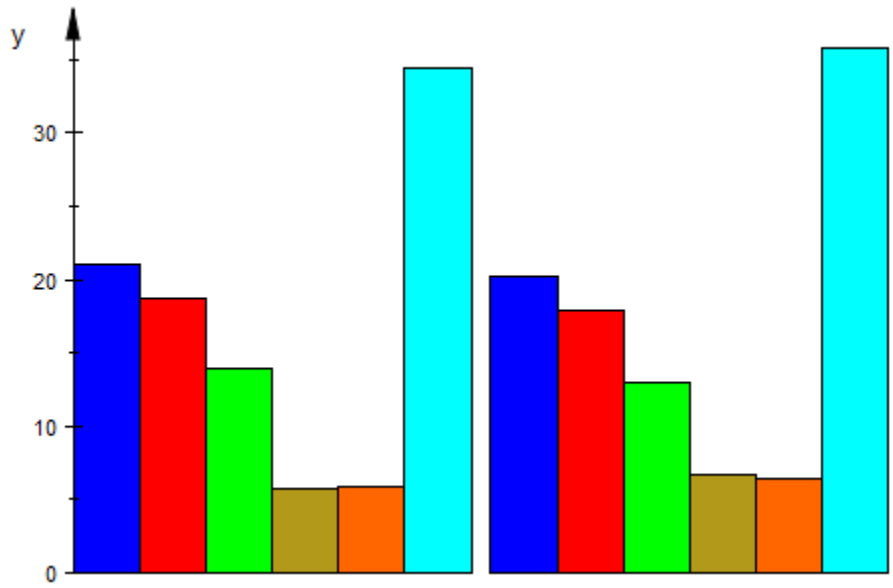


The following data are the australian market shares (in percent) of major car producers in the years 2004 and 2005:

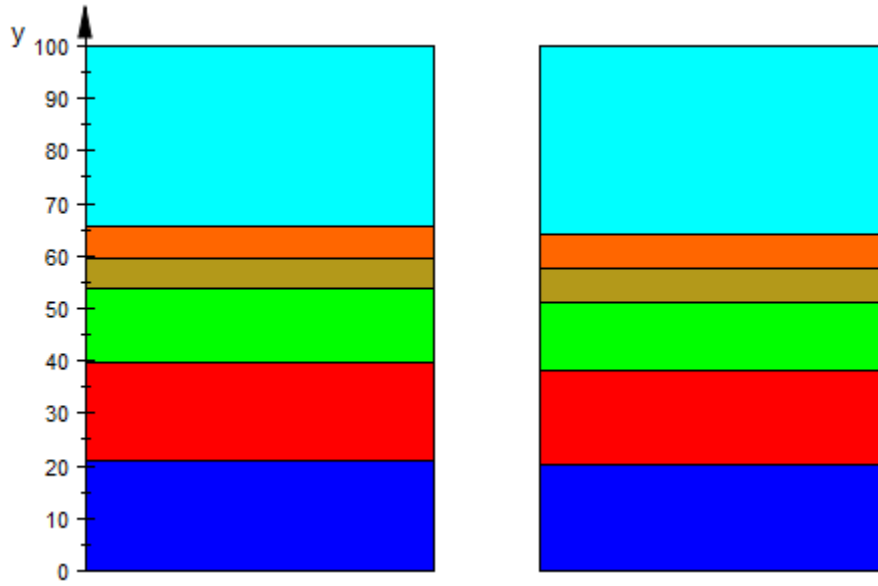
```
           // 2004  2005
Toyota:=     [21.1, 20.2]:
Holden_GM:=  [18.7, 17.9]:
Ford:=       [14.0, 13.0]:
Mazda:=      [ 5.8,  6.7]:
Mitsubishi:= [ 5.9,  6.4]:
Others:=     [34.5, 35.8]:
data:= [Toyota, Holden_GM, Ford, Mazda, Mitsubishi, Others]:
```

We visualize the change of the market shares by bar plots using different group styles:

```
plot(plot::Bars2d(data, GroupStyle = MultipleBars))
```




```
plot(plot::Bars2d(data, GroupStyle = SingleBars))
```

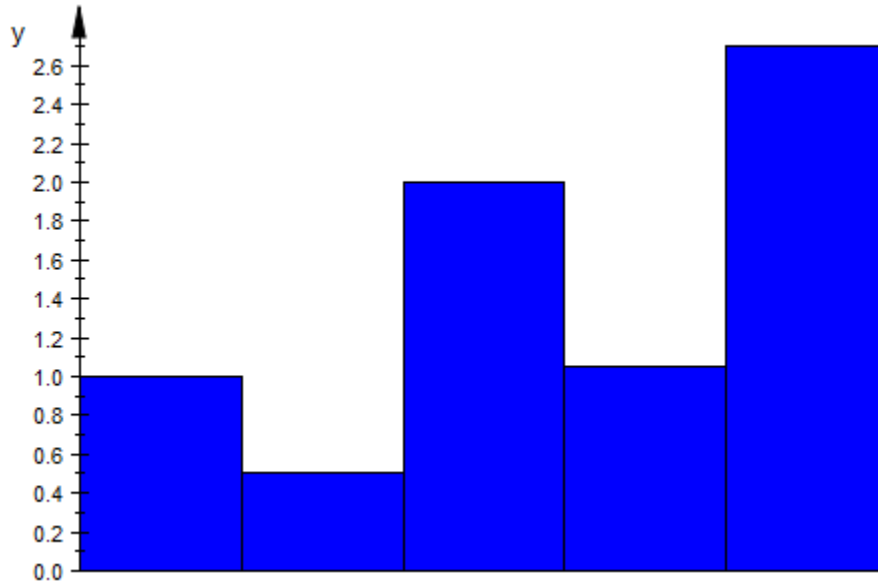



```
delete group1, group2, data, Toyota, Holden_GM,
       Ford, Mazda, Mitsubishi, Others:
```

Example 2

The option `SingleBars` has no effect when the data of only one group are given:

```
group:= [1, 0.5, 2, PI/3, 2.7]:
plot(plot::Bars2d(group, GroupStyle = SingleBars))
```



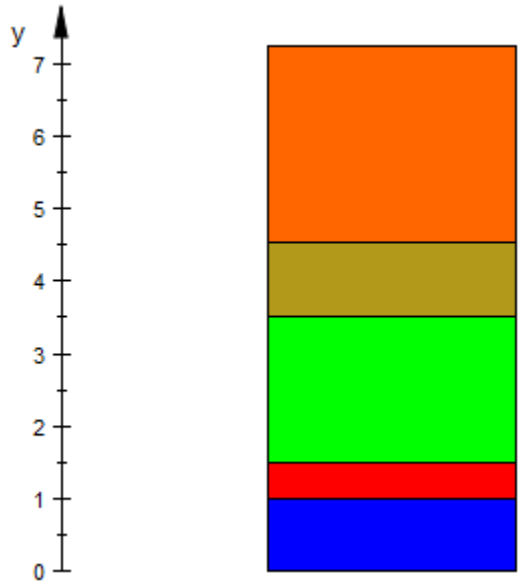
Each value is turned into a separate group:

```
groups:= [[x] $ x in group]
```

$$\left[[1], [0.5], [2], \left[\frac{\pi}{3}\right], [2.7]\right]$$

Now, `SingleBars` has an effect:

```
plot(plot::Bars2d(groups, GroupStyle = SingleBars,
           BarCenters = [0.4], BarWidths = [0.3]),
           ViewingBox = [0..1, Automatic])
```




```
delete group, groups:
```

See Also

MuPAD Functions
`BarStyle` | `Colors` | `FillPatterns`

InterpolationStyle

Interpolation via linear or cubic splines

Value Summary

Optional	`Cubic`, or `Linear`

Graphics Primitives

Objects	InterpolationStyle Default Values
`plot::Listplot`, `plot::Matrixplot`	`Linear`

Description

`InterpolationStyle` determines whether discrete data are interpolated linearly or via cubic splines.

With the default setting `InterpolationStyle = Linear`, the curve connecting the data points in a plot of type `plot::Listplot` consists of line segments. Similarly, the surface generated from the matrix data in `plot::Matrixplot` consists of linear segments (triangles).

These plot objects do not react to the attribute `Submesh` when using linear interpolation.

With `InterpolationStyle = Cubic`, the curve connecting the data points in a plot of type `plot::Listplot` is the graph of the cubic spline function interpolating the data points. Similarly, the surface generated by `plot::Matrixplot` is the graph of the cubic spline function interpolating the matrix data.

The spline functions can be rendered smoothly by setting appropriate values for the attribute `Submesh`.

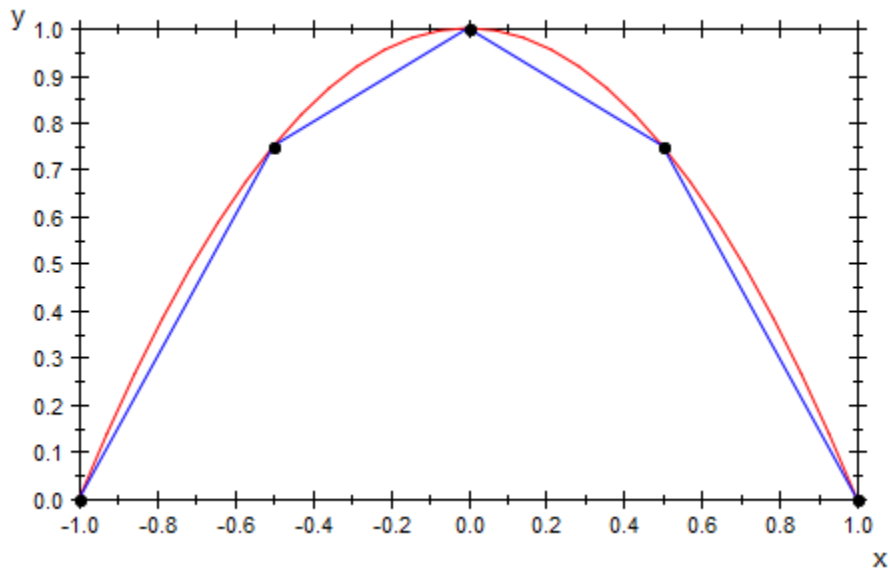
For large amounts of data, rendering with cubic spline interpolation may be much more costly than linear interpolation.

Examples

Example 1

We sample the function $f(x) = \frac{1}{(1+x^2)}$ at various points and store the data in a list. The data are displayed via `plot::Listplot` with different interpolation styles:

```
L := [1 - (i/2)^2 $ i = -2..2]:
plot(plot::Listplot(L, x = -1..1, InterpolationStyle = Cubic,
                    Color = RGB::Red),
     plot::Listplot(L, x = -1..1, InterpolationStyle = Linear,
                    Color = RGB::Blue)):
```

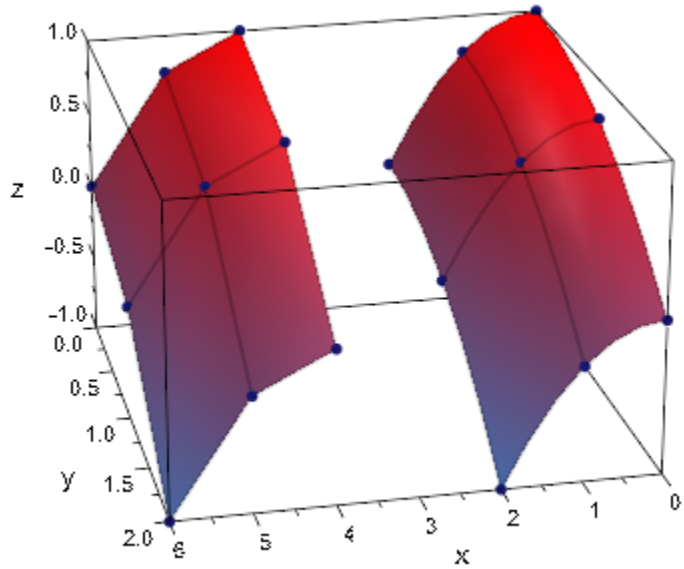



```
delete L:
```

Example 2

We sample the function $f(x, y) = 1 - x^2 - y^2$ at various points and store the data in a matrix. The matrix data are displayed as a matrix plot with different interpolation styles:

```
A := matrix([[1 - (i/2)^2 - (j/2)^2 $ j = 0..2] $ i = 0..2]):
plot(plot::Matrixplot(A, x = 0..2, y = 0..2,
                      InterpolationStyle = Cubic),
     plot::Matrixplot(A, x = 4..6, y = 0..2,
                      InterpolationStyle = Linear),
     CameraDirection = [10, 15, 9]):
```




```
delete A:
```

See Also

MuPAD Functions
Submesh

Shading

Smooth color blend of surfaces

Value Summary

Inherited	`Flat`, or `Smooth`

Graphics Primitives

Objects	Shading Default Values
`plot::Cone`, `plot::Dodecahedron`, `plot::Ellipsoid`, `plot::Function3d`, `plot::Hexahedron`, `plot::Icosahedron`, `plot::Implicit3d`, `plot::Matrixplot`, `plot::Octahedron`, `plot::Prism`, `plot::Pyramid`, `plot::Sphere`, `plot::Surface`, `plot::SurfaceSet`, `plot::SurfaceSTL`, `plot::Tetrahedron`, `plot::Tube`, `plot::Waterman`, `plot::XRotate`, `plot::ZRotate`	`Smooth`

Description

Using `Shading`, a smooth color blend of triangulated surfaces can be (de-)activated.

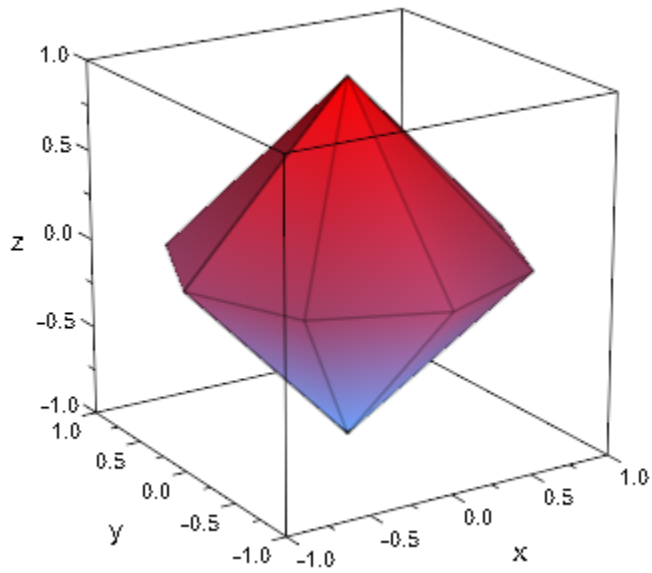
Most surfaces in 3D are triangulated for display. The triangles are then drawn using Gouraud-shading to achieve a smooth visual effect. Using `Shading = Flat`, you can instruct the viewer to display the plain triangles.

Examples

Example 1

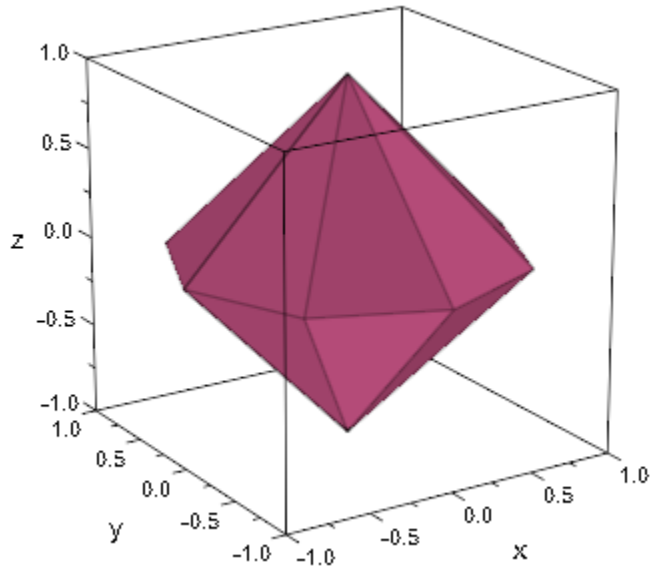
Reducing the mesh density of a surface usually has more effect on its outer rim than on the display of the middle:

```
plot(plot::Spherical([1, u, v], u=0..PI, v=0..2*PI,
                     UMesh=5, VMesh=5))
```



Setting `Shading = Flat`, you can see the triangles from which the sphere is constructed:

```
plot(plot::Spherical([1, u, v], u=0..PI, v=0..2*PI,
                     UMesh=5, VMesh=5),
     Shading = Flat)
```



See Also

MuPAD Functions

`Lighting` | `UMesh` | `USubmesh` | `VMesh` | `VSubmesh`

UseNormals

Use predefined normals?

Value Summary

Optional	TRUE or FALSE

Graphics Primitives

Objects	UseNormals Default Values
`plot::SurfaceSet`, `plot::SurfaceSTL`	TRUE

Description

`UseNormals` controls whether predefined normals of triangulation data are used when plotting a surface.

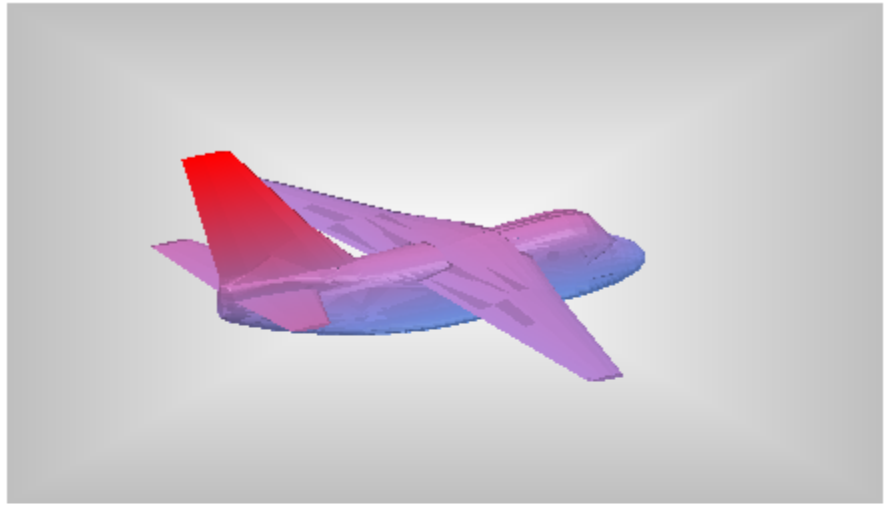
With `UseNormals = FALSE`, the predefined normals given in the triangulation data (`MeshList`) of a `plot::SurfaceSet` or in an STL file imported by a `plot::SurfaceSTL` are ignored when plotting this surface. This may reduce the data volume of the graphical object and the computing time as well. However, it usually leads to a somewhat less brilliant image.

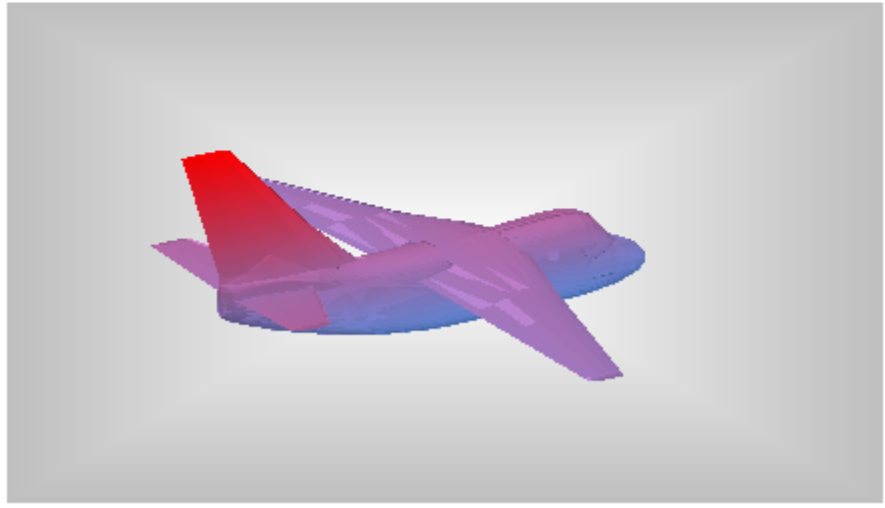
Examples

Example 1

By default, the normals defined in STL files are used when plotting the corresponding MuPAD object. For comparison, an STL graphics is plotted with and without using the normals provided by the STL file:

```
plot(plot::Scene3d(plot::SurfaceSTL("skin.stl")),
     plot::Scene3d(plot::SurfaceSTL("skin.stl",
                                    UseNormals = FALSE)),
     Width = 120*unit::mm, Height = 140*unit::mm,
     Layout = Vertical, BackgroundStyle = Pyramid,
     Axes = None):
```





See Also

MuPAD Functions
`Filled` | `MeshList` | `MeshListNormals`

TipAngle

Opening angle of arrow heads

Value Summary

Inherited	Positive real number

Graphics Primitives

Objects	TipAngle Default Values
`plot::Arrow2d`, `plot::Arrow3d`, `plot::Streamlines2d`	`(2*PI)/15`
`plot::VectorField2d`	`0.6283185307`

Description

`TipAngle` determines the opening angle of arrow heads in radians.

`TipAngle` determines the opening angle of the tips of arrows of type `plot::Arrow2d` and `plot::Arrow3d`. Also the arrow tips in a vector field of type `plot::VectorField2d` are controlled by `TipAngle`. The opening angle must be specified in radians. Values for `TipAngle` between 0 and π are reasonable.

The tip angle is the geometric angle of the arrow heads as visible in the graphical output. It is invariant under scaling and zooming.

The values of `TipAngle` cannot be animated.

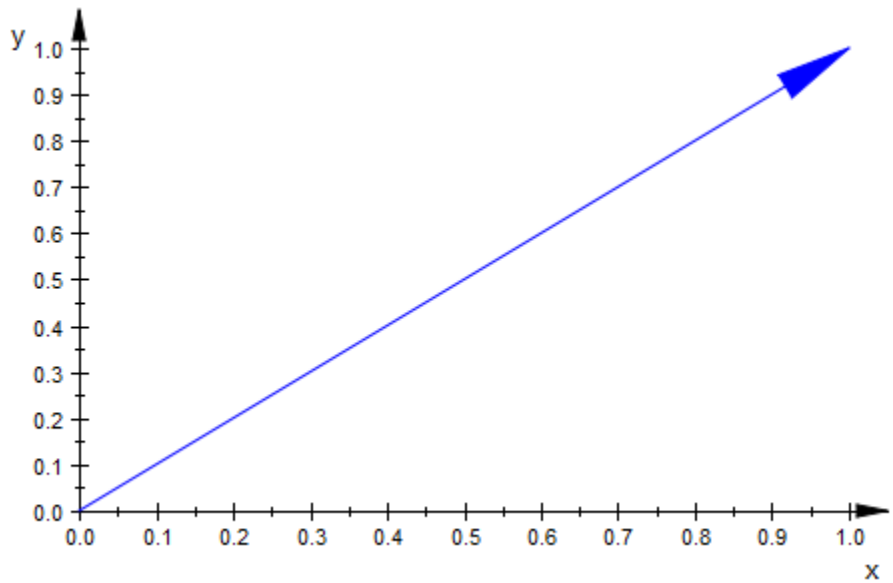
The attribute `TipStyle` sets the presentation style of arrow tips. `TipLength` sets the physical tip length.

Examples

Example 1

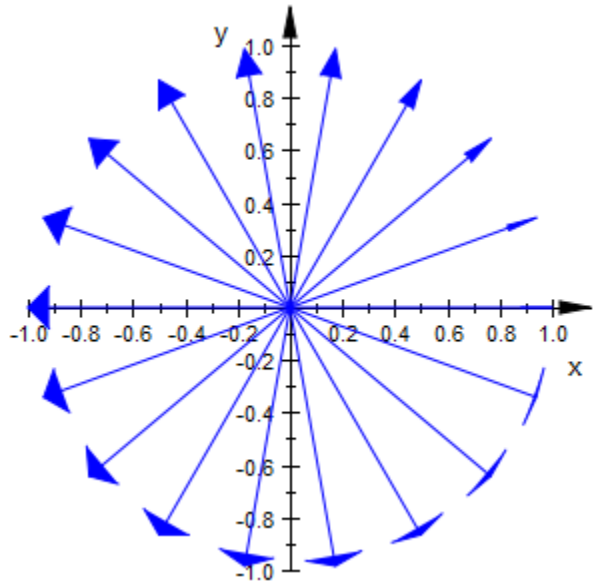
We create an arrow whose arrow tip has an angle of 20 degrees. This corresponds to $\frac{\pi}{9}$ radians:

```
plot(plot::Arrow2d([0, 0], [1, 1], TipAngle = PI/9,
                   TipLength = 10*unit::mm)):
```



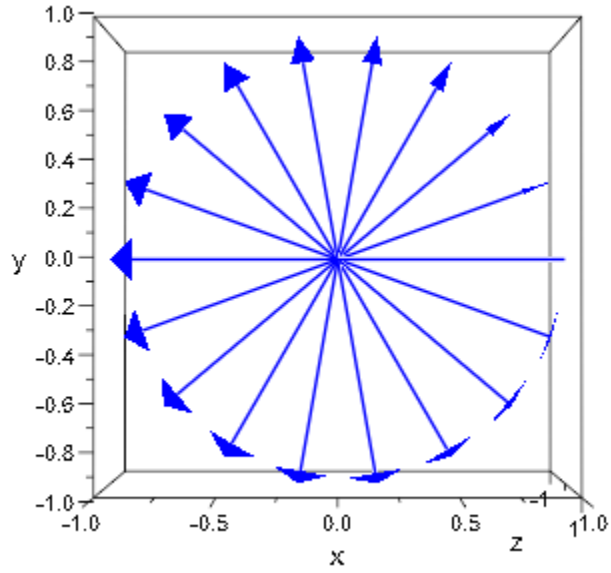
We create several arrows with different tip angles. The angle is increased by 10 degrees from one arrow to the next:

```
plot(plot::Arrow2d([0, 0], [cos(a*2*PI/18), sin(a*2*PI/18)],
                   TipAngle = a*PI/18) $ a = 0 .. 17,
     Scaling = Constrained):
```



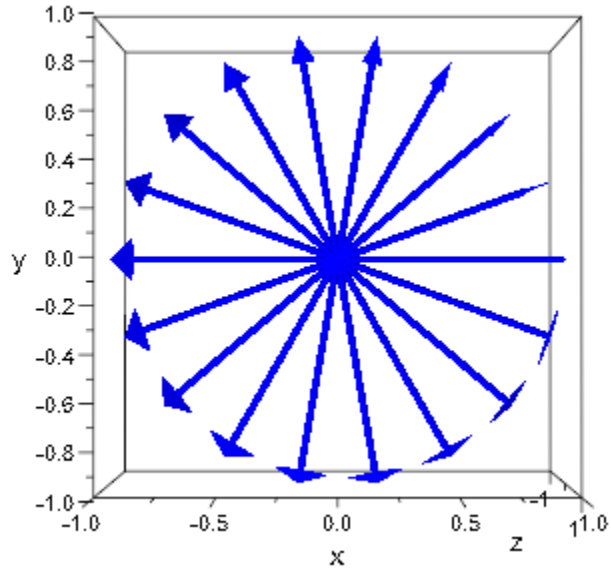
Here are corresponding arrows in 3D:

```
plot(plot::Arrow3d([0, 0, 0],
                   [cos(a*2*PI/18), sin(a*2*PI/18), 0],
                   TipAngle = a*PI/18) $ a = 0 .. 17,
     Scaling = Constrained,
     CameraDirection = [0, -10, 1000]):
```



We use `Tubular` = `TRUE`:

```
plot(plot::Arrow3d([0, 0, 0],
                   [cos(a*2*PI/18), sin(a*2*PI/18), 0],
                   TipAngle = a*PI/18) $ a = 0 .. 17,
     Tubular = TRUE, Scaling = Constrained,
     CameraDirection = [0, -10, 1000]):
```



See Also

MuPAD Functions

`TipLength | TipStyle | TubeDiameter | Tubular`

TipLength

Length of arrow heads

Value Summary

Inherited	Non-negative output size

Graphics Primitives

Objects	TipLength Default Values
`plot::Arrow2d`, `plot::Arrow3d`	`4`
`plot::VectorField2d`	`1.5`
`plot::Streamlines2d`	`0`

Description

`TipLength` determines the physical length of arrow heads

`TipLength` determines the length of the tips of arrows of type `plot::Arrow2d` and `plot::Arrow3d`. Also the arrow tips in a vector field of type `plot::VectorField2d` are controlled by `TipLength`. The value should be specified as an absolute physical length including a length unit such as `TipLength = 2.5*unit::mm`. Numbers without a physical unit give the size in mm.

The tip length is the physical length of the arrow heads as visible in the graphical output. It is invariant under scaling and zooming.

The values of `TipLength` cannot be animated.

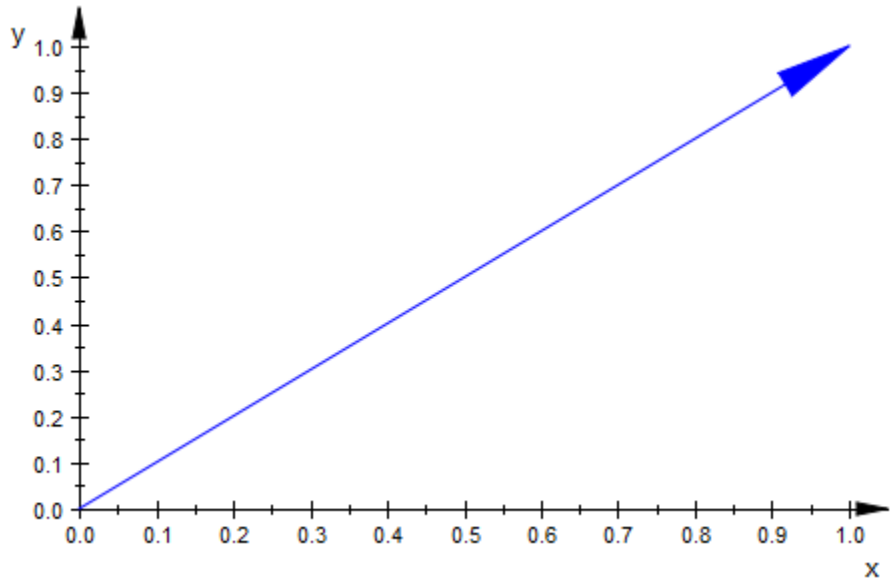
The attribute `TipStyle` sets the presentation style of arrow tips. `TipAngle` sets the opening angle of the tips.

Examples

Example 1

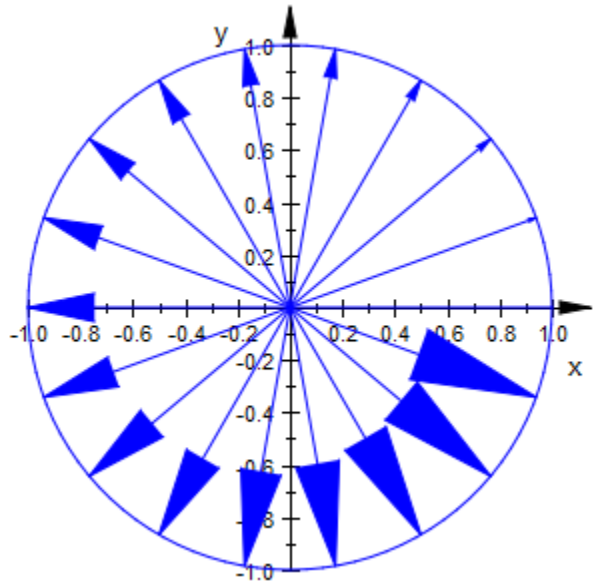
We create an arrow whose tip has physical length of 10 mm:

```
plot(plot::Arrow2d([0, 0], [1, 1], TipAngle = PI/9,
                   TipLength = 10*unit::mm)):
```



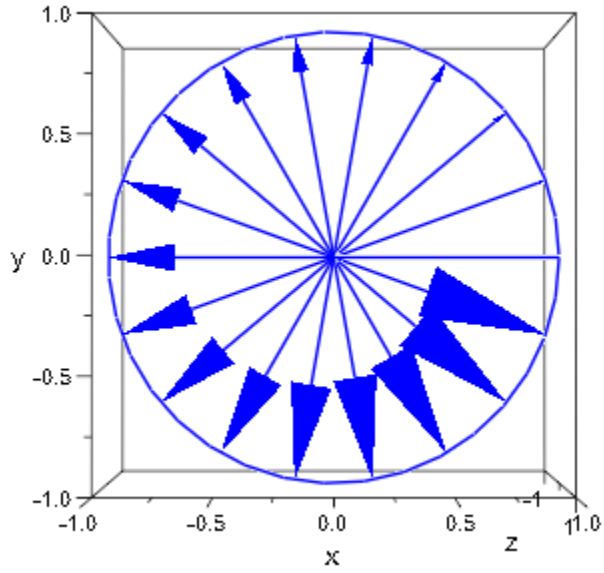
We create several arrows with different tip lengths. The length is increased by 0.7 mm from one arrow to the next:

```
plot(plot::Arrow2d([0, 0], [cos(a*2*PI/18), sin(a*2*PI/18)],
                   TipLength = a*unit::mm) $ a = 0 .. 17,
     plot::Circle2d(1, [0, 0]),
     Scaling = Constrained):
```



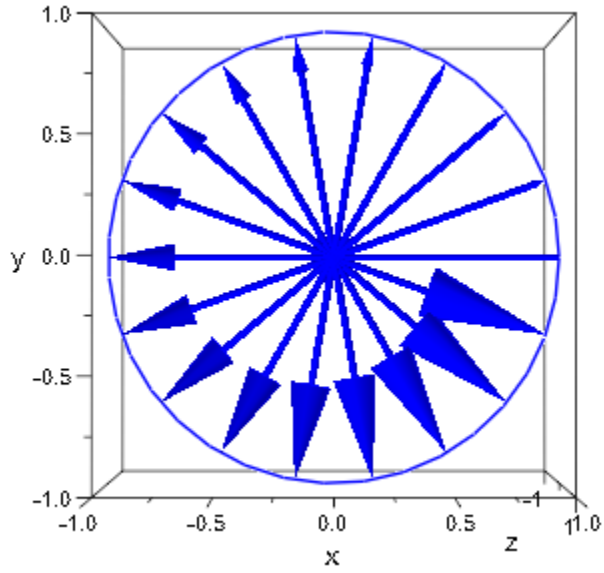
Here are corresponding arrows in 3D:

```
plot(plot::Arrow3d([0, 0, 0],
                   [cos(a*2*PI/18), sin(a*2*PI/18), 0],
                   TipLength = a*unit::mm) $ a = 0 .. 17,
     plot::Circle3d(1, [0, 0, 0], [0, 0, 1]),
     Scaling = Constrained,
     CameraDirection = [0, -10, 1000]):
```



We use `Tubular` = `TRUE`:

```
plot(plot::Arrow3d([0, 0, 0],
                   [cos(a*2*PI/18), sin(a*2*PI/18), 0],
                   TipLength = a*unit::mm) $ a = 0 .. 17,
     plot::Circle3d(1, [0, 0, 0], [0, 0, 1]),
     Tubular = TRUE, Scaling = Constrained,
     CameraDirection = [0, -10, 1000]):
```



See Also

MuPAD Functions

`TipAngle` | `TipStyle` | `TubeDiameter` | `Tubular`

TipStyle

Presentation style of arrow heads

Value Summary

Inherited	`Closed`, `Filled`, or `Open`

Graphics Primitives

Objects	TipStyle Default Values
`plot::Arrow2d`, `plot::Arrow3d`, `plot::Streamlines2d`	`Filled`
`plot::VectorField2d`	`Open`

Description

`TipStyle` governs the appearance of arrow heads.

`TipStyle` determines how the tips of arrows of type `plot::Arrow2d` and `plot::Arrow3d` look. Also the arrow tips in a vector field of type `plot::VectorField2d` are controlled by `TipStyle`.

With `TipStyle` = `Open`, the tips are given by two lines.

With `TipStyle` = `Closed`, the tips are given by a triangle.

With `TipStyle` = `Filled`, the tips are given by a filled triangle.

`TipStyle` cannot be animated.

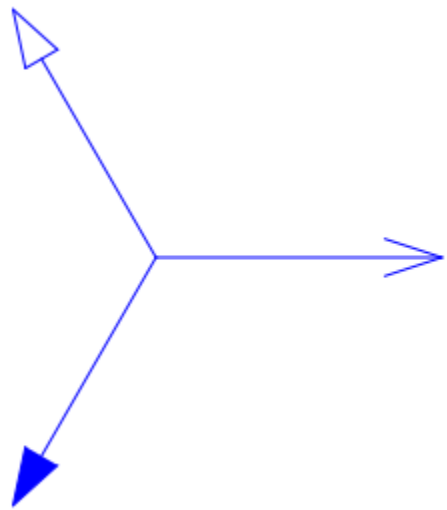
The opening angle and the physical length of the arrow tips are set by the attributes `TipAngle` and `TipLength`, respectively.

Examples

Example 1

We draw arrows with different tip styles:

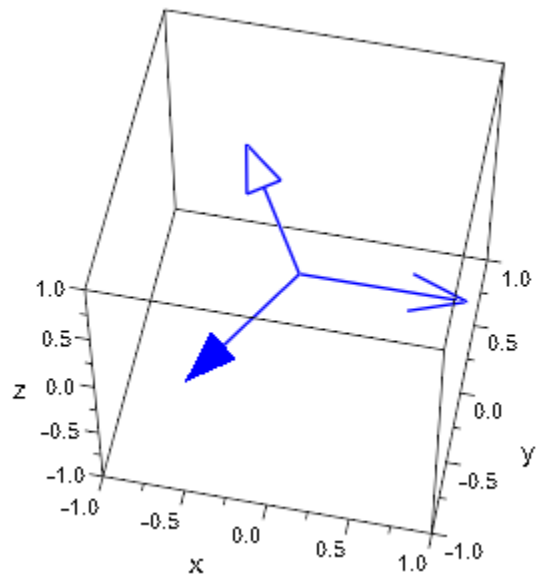
```
plot(plot::Arrow2d([0, 0], [1, 0],
                   TipStyle = Open),
     plot::Arrow2d([0, 0], [cos(2*PI/3), sin(2*PI/3)],
                   TipStyle = Closed),
     plot::Arrow2d([0, 0], [cos(4*PI/3), sin(4*PI/3)],
                   TipStyle = Filled),
     Axes = None, ViewingBox = [-1..1, -1..1],
     TipLength = 8*unit::mm, TipAngle = PI/5,
     Scaling = Constrained):
```



Here are corresponding arrows in 3D:

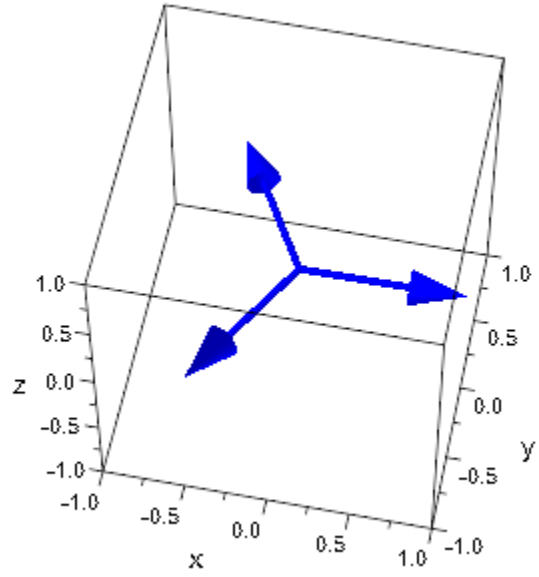
```
plot(plot::Arrow3d([0, 0, 0], [1, 0, 0],
                   TipStyle = Open),
```

```
plot::Arrow3d([0, 0, 0], [cos(2*PI/3), sin(2*PI/3), 0],
              TipStyle = Closed),
plot::Arrow3d([0, 0, 0], [cos(4*PI/3), sin(4*PI/3), 0],
              TipStyle = Filled),
ViewingBox = [-1..1, -1..1, -1..1],
TipLength = 8*unit::mm, TipAngle = PI/5,
Scaling = Constrained, CameraDirection = [2, -10, 15]):
```



We use `Tubular` = `TRUE`:

```
plot(plot::Arrow3d([0, 0, 0], [1, 0, 0],
                   TipStyle = Open),
     plot::Arrow3d([0, 0, 0], [cos(2*PI/3), sin(2*PI/3), 0],
                   TipStyle = Closed),
     plot::Arrow3d([0, 0, 0], [cos(4*PI/3), sin(4*PI/3), 0],
                   TipStyle = Filled),
     TipLength = 8*unit::mm, TipAngle = PI/5,
     Tubular = TRUE, ViewingBox = [-1..1, -1..1, -1..1],
     Scaling = Constrained, CameraDirection = [2, -10, 15]):
```



See Also

MuPAD Functions

`TipAngle` | `TipLength` | `TubeDiameter` | `Tubular`

TubeDiameter

Diameter of tubular arrows and lines. , and coordinate axes

Value Summary

Inherited	Positive output size

Graphics Primitives

Objects	TubeDiameter Default Values
`plot::Arrow3d`, `plot::Line3d`	`1.0`

Description

`TubeDiameter` governs the size of tubular arrows and lines in 3D.

Arrows of type `plot::Arrow3d` as well as lines of type `plot::Line3d` can be rendered as 3D tubes by setting the attribute `Tubular` = `TRUE`.

The attribute `TubeDiameter` determines the diameter of tubular arrows and lines. Its value should be specified as an absolute physical length including a length unit such as `TubeDiameter = 2.5*unit::mm`. Numbers without a physical unit give the size in mm.

Tubular arrows have a tip that is rendered as a little cone. The size of these cones is adjusted when the diameter of the arrow shaft changes.

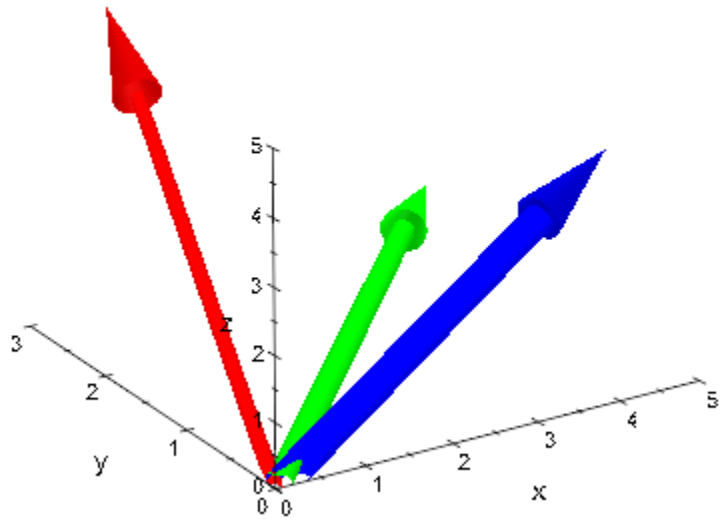
The attribute `TubeDiameter` is not available in 2D.

Examples

Example 1

We draw some tubular arrows with different tube diameters:

```
plot(plot::Arrow3d([0, 0, 0], [1, 3, 5], Color = RGB::Red,
                   TubeDiameter = 2.0*unit::mm),
     plot::Arrow3d([0, 0, 0], [5, 3, 1], Color = RGB::Green,
                   TubeDiameter = 3.0*unit::mm),
     plot::Arrow3d([0, 0, 0], [5, 1, 3], Color = RGB::Blue,
                   TubeDiameter = 4.0*unit::mm),
     Tubular = TRUE, TipLength = 18*unit::mm, Axes = Origin):
```



See Also

MuPAD Functions
`TipAngle` | `TipLength`

Tubular

Display 3D arrows and lines as tubes?

Value Summary

Inherited	FALSE, or TRUE

Graphics Primitives

Objects	Tubular Default Values
`plot::Arrow3d`, `plot::Line3d`	FALSE

Description

With `Tubular` = `TRUE`, arrows of type `plot::Arrow3d` and lines of type `plot::Line3d` are rendered as tubes (cylinders). The diameter is set by the attribute `TubeDiameter`.

Tubular arrows have a tip that is rendered as a little cone. The tip is determined by the attributes `TipLength` and `TipAngle`.

With `Tubular` = `FALSE`, arrows and lines are displayed as simple lines.

The attribute `Tubular` is not available in 2D.

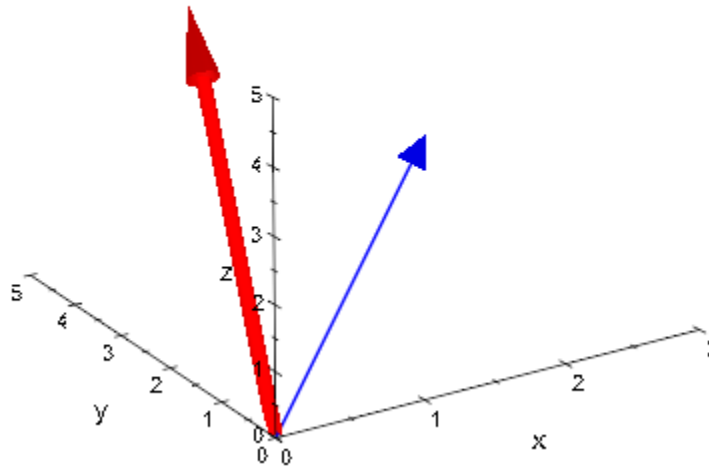
Examples

Example 1

We draw a tubular arrow together with an arrow displayed as a line:

```
plot(plot::Arrow3d([0, 0, 0], [1/2, 3, 5], Tubular = TRUE,
                   TubeDiameter = 2*unit::mm,
                   Color = RGB::Red),
```

```
plot::Arrow3d([0, 0, 0], [3, 5, 1], Tubular = FALSE,
              Color = RGB::Blue),
TipLength = 12*unit::mm, Axes = Origin):
```



See Also

MuPAD Functions

`TipAngle` | `TipLength` | `TubeDiameter`

polylib – Manipulating Polynomials

polylib::coeffRing
polylib::cyclotomic
polylib::decompose
polylib::discrim
polylib::divisors
polylib::Dpoly
polylib::elemSym
polylib::makerat
polylib::minpoly
polylib::Poly
polylib::primitiveElement
polylib::primpart
polylib::randpoly
polylib::realroots
polylib::representByElemSym
polylib::resultant
polylib::sortMonomials
polylib::splitfield
polylib::sqrfree
polylib::subresultant
polylib::support

polylib::coeffRing

Coefficient ring of a polynomial

Syntax

```
polylib::coeffRing(P)
```

```
polylib::coeffRing(p)
```

Description

`polylib::coeffRing(p)` returns the coefficient ring of `p`.

`polylib::coeffRing`(p) allows to query in a uniform way the coefficient ring of the polynomial `p` or the polynomial domain `P`.

`P` can be any polynomial domain (`Dom::UnivariatePolynomial`x, `Dom::DistributedPolynomial`[x,y], ...).

`P` can also be of the form `polylib::Poly([x,y],K)`. If `K` is `Expr` or `IntMod(n)`, then the corresponding domains `Dom::ExpressionField()` or `Dom::IntegerMod(p)` is returned. See `poly` for the details about `Expr` and `IntMod(n)`.

`p` can be a kernel polynomial (`DOM_POLY`), or an element of one of the above domains

Examples

Example 1

We define a polynomial ring over the ring of integers modulo 4, and query for its coefficient ring:

```
P := Dom::UnivariatePolynomial(x, Dom::IntegerMod(4)):
polylib::coeffRing(P)
```

```
Dom::IntegerMod(4)
```

The coefficient ring of the elements of this domain can be queried the same way:

```
polylib::coeffRing(P(x))
```

Dom::IntegerMod(4)

```
polylib::coeffRing(Dom::Matrix(Dom::IntegerMod(3)))
```

Dom::IntegerMod(3)

Example 2

When no coefficient ring is specified, `poly` currently constructs kernel polynomials over the fake domain Expr instead of the mathematically equivalent field `Dom::ExpressionField()` of arbitrary expression (this happens to be more efficient with the current kernels):

```
extop(poly(x))
```

x, $[x]$, Expr

```
polylib::coeffRing(poly(x))
```

Dom::ExpressionField()

This makes it possible to plug the result right away as coefficient ring of some other domain:

```
Dom::UnivariatePolynomial(x, polylib::coeffRing(poly(x)))
```

Dom::UnivariatePolynomial(x, Dom::ExpressionField(), LexOrder)

Parameters

P

A polynomial domain

p

A polynomial

Return Values

Domain

polylib::cyclotomic

Cyclotomic polynomials

Syntax

```
polylib::cyclotomic(n, x)
```

Description

`polylib::cyclotomic(n, x)` computes the n-th cyclotomic polynomial, expressed in the indeterminate x.

The n-th cyclotomic polynomial is defined to be the minimal polynomial of any n-th primitive root of unity over the field of rational numbers.

Examples

Example 1

We compute the 20th cyclotomic polynomial.

```
polylib::cyclotomic(20, z);
```

$$\mathrm{poly}(z^8 - z^6 + z^4 - z^2 + 1, [z])$$

Parameters

n

Positive integer

x

Identifier

Return Values

Polynomial over `Expr` in the indeterminate `x`.

See Also

MuPAD Functions
`numlib::phi`

polylib::decompose

Functional decomposition of a polynomial

Syntax

```
polylib::decompose(p)
```

```
polylib::decompose(p, x)
```

Description

`polylib::decompose(p,x)` returns a sequence of polynomials q_1, …, q_n such that $p(x) = q_1(\ldots q_n(x) \ldots)$.

The second argument may be left out if the polynomial is univariate, as in “Example 1” on page 25-7.

If a polynomial has several decompositions, it is not specified which of them is returned.

Examples

Example 1

In the simplest case, an univariate polynomial is decomposed with respect to its only variable:

```
polylib::decompose(x^4+x^2+1)
```

$$x^2+x+1,\ x^2$$

Example 2

If there are several variables, a main variable must be specified:

```
polylib::decompose(y*x^4+y,y);
```

$$y\,x^4 + y$$

Parameters

p

Polynomial or polynomial expression

x

One of the indeterminates of the polynomial `p`

Return Values

If a decomposition is possible, `polylib::decompose` returns it as an expression sequence, each element being of the same type as the input. If no decomposition is possible, the input is returned.

Overloaded By

p

References

A description of the algorithm behind `polylib::decompose` can be found in *Barton and Zippel*, Polynomial decomposition algorithms, Journal of Symbolic Computation, 1 (1985), pp. 159–168.

See Also

MuPAD Functions
`factor`

polylib::discrim

Discriminant of a polynomial

Syntax

```
polylib::discrim(p, x)
```

Description

`polylib::discrim(p, x)` returns the discriminant of the polynomial `p` with respect to the variable `x`.

The function `normal` is applied to the discriminant before returning it.

Examples

Example 1

We compute the discriminant of the general quadratic equation:

```
polylib::discrim(a*x^2 + b*x + c, x);
```

$$b^2 - 4\,a\,c$$

Parameters

x

Indeterminante

p

Polynomial or polynomial expression

Return Values

`polylib::discrim` returns an element of the coefficient ring of `p`. If the coefficient ring is `Expr` or `IntMod(n)`, an expression is returned.

Overloaded By

p

Algorithms

The discriminant of `p` with respect to the variable `x` is defined as:

$$\frac{(-1)^{\frac{d\,(d-1)}{2}}\,\mathrm{res}_x(p,\,p')}{c},$$

where d is the degree and c is the leading coefficient of p.

See Also

MuPAD Functions
`polylib::resultant`

polylib::divisors

Divisors of a polynomial, polynomial expression, or Factored element

Syntax

```
polylib::divisors(p)
```

```
polylib::divisors(f)
```

```
polylib::divisors(e)
```

Description

`polylib::divisors(p)` computes the set of all monic divisors of the polynomial or polynomial expression `p`.

`polylib::divisors(f)` returns all monic divisors of a pre-factored polynomial. Cf. “Example 3” on page 25-12.

`polylib::divisors` works on polynomials of category `Cat::Polynomial` as well. Cf. “Example 4” on page 25-12.

Examples

Example 1

If the argument is a polynomial, a set of polynomials is returned:

```
polylib::divisors(poly(x^2 - 2*x + 1))
```

$$\{\mathrm{poly}(1, [x]),\ \mathrm{poly}(x-1, [x]),\ \mathrm{poly}(x^2-2x+1, [x])\}$$

Example 2

If the argument is a polynomial expression, a set of polynomial expressions is returned:

```
polylib::divisors(x^2 - 1)
```

$$\{1, x^2 - 1, x - 1, x + 1\}$$

Example 3

If the argument is of type `Factored` (a `factor` return value) a set of polynomials is returned:

```
p := factor(poly(x^2 - 1));
polylib::divisors(p)
```

$$\mathrm{poly}(x - 1, [x])\ \mathrm{poly}(x + 1, [x])$$

$$\{\mathrm{poly}(1, [x]), \mathrm{poly}(x - 1, [x]), \mathrm{poly}(x + 1, [x]), \mathrm{poly}(x^2 - 1, [x])\}$$

The polynomials in the resulting set have the same type as the polynomials in the `Factored` element:

```
p := factor(x^2 - 1);
polylib::divisors(p)
```

$$(x - 1)\ (x + 1)$$

$$\{1, x^2 - 1, x - 1, x + 1\}$$

Example 4

`polylib::divisors` works on polynomials from category `Cat::Polynomial` as well:

```
P := Dom::Polynomial(Dom::IntegerMod(7)):
polylib::divisors(P(x^3 + 2*x^2 + 1))
```

$$\{1 \bmod 7, (1 \bmod 7)\ x^3 + (2 \bmod 7)\ x^2 + (1 \bmod 7)\}$$

Parameters

p

A polynomial or polynomial expression

f

`Factored` (return value of `factor`)

e

Element of a domain of category `Cat::Polynomial`

Return Values

`polylib::divisors` returns a set of polynomials. The polynomials are from the same type as the polynomials in the argument.

See Also

MuPAD Categories
`Cat::Polynomial`

MuPAD Domains
`Dom::MultivariatePolynomial` | `Dom::Polynomial` | `Dom::UnivariatePolynomial` | `DOM_POLY`

MuPAD Functions
`factor` | `irreducible` | `numlib::divisors` | `polylib::sqrfree`

polylib::Dpoly

Differential operator for polynomials

Syntax

```
polylib::Dpoly(f)
```

```
polylib::Dpoly(indexlist, f)
```

Description

If `f` is a polynomial in the indeterminates `x1` through xn, `polylib::Dpoly([i1,..,ik], f)` computes the *k*-th partial derivative $\frac{\partial}{\partial x_{ik}} \frac{\partial}{\partial \ldots} \frac{\partial}{\partial x_{i1}} f$.

`polylib::Dpoly(f)` returns the derivative of `f` with respect to its only variable for an univariate polynomial `f`.

If some element of `indexlist` is greater than the number of indeterminates of `f`, the zero polynomial is returned.

`polylib::Dpoly([], p)` returns `p`.

If the coefficients of the polynomial are elements of a domain `d`, then this domain must have the method `"intmult"` (`d::intmult(e,i)`) that must calculate the integer multiple of a domain element `e` and a positive integer `i`.

Examples

Example 1

We differentiate a univariate polynomial with respect to its only indeterminate. In this case, we may leave out the first argument.

```
polylib::Dpoly(poly(2*x^2 + x + 1));
```

$$\mathrm{poly}(4x+1, [x])$$

Example 2

Now we differentiate a bivariate polynomial, and must specify the indeterminate in this case.

```
polylib::Dpoly([1], poly(x^2*y + 3*x + y, [x, y]));
```

$$\mathrm{poly}(2xy+3, [x, y])$$

Example 3

It is also possible to compute second or higher partial derivatives.

```
polylib::Dpoly([1, 2], poly(x^2*y + 3*x + y, [x, y]));
```

$$\mathrm{poly}(2x, [x, y])$$

Parameters

`f`

Polynomial

`indexlist`

List of positive integers

Return Values

`polylib::Dpoly` returns a polynomial in the same indeterminates and over the same coefficient ring as the input.

Overloaded By

f

See Also

MuPAD Functions
`D | diff`

polylib::elemSym

Elementary symmetric polynomials

Syntax

```
polylib::elemSym(l, k)
```

Description

`polylib::elemSym([x1,...,xn], k)` returns the `k`-th elementary symmetric polynomial in the given variables `x1` through `xn`.

A given list `l` is a valid first argument only if its elements can be used as indeterminates of a polynomial .

Examples

Example 1

The first elementary symmetric polynomial is just the sum of its variables:

```
polylib::elemSym([x,y,z], 1);
```

$$\mathrm{poly}(x + y + z, [x, y, z])$$

Example 2

Indeterminates may also be e.g. trigonometric functions:

```
polylib::elemSym([sin(u),cos(u), exp(u)], 2);
```

$$\mathrm{poly}(\sin(u)\cos(u) + \sin(u)\,e^{u} + \cos(u)\,e^{u}, [\sin(u), \cos(u), e^{u}])$$

Parameters

l

List of indeterminatess

k

Positive integer

Return Values

Result is a polynomial over the coefficient ring `Expr`. If `k` is greater than the number of operands of `l`, `undefined` is returned.

References

For more information about elementary symmetric polynomials, see v.d. Waerden, Algebra, vol. 1.

See Also

MuPAD Functions
`polylib::representByElemSym`

polylib::makerat

Convert expression into rational function over a suitable field

Syntax

```
polylib::makerat(a, <maxd>)
```

```
polylib::makerat(l, <maxd>)
```

Description

`polylib::makerat(a)` returns two polynomials *f* and *g* over the rationals and a list of substitutions such that applying the substitutions to the rational function $\frac{f}{g}$ gives `a`.

`polylib::makerat(l)` does the same for every element of the list `l` and returns lists of resulting *f*'s and *g*'s.

`polylib::makerat(a, maxd)` replaces *d*-th roots of integers by elements of some algebraic extension field over the rationals if $d \le maxd$, and returns polynomials *f* and *g* over that extension field.

`polylib::makerat(a)` replaces all irrational subexpressions (except identifiers) in `a` by newly created identifiers, thereby producing a rational function over the rationals. It returns the numerator and denominator of that rational function as polynomials over `Expr`, and the substitutions to be made to get back the numerator and denominator of the original input `a`.

`polylib::makerat(l)` replaces all irrational subexpressions in all elements of `l` by newly created identifiers.

Every subexpression is replaced by the same identifier every time it occurs.

All indeterminates of the input and all of the new identifiers become indeterminates of the result, unless a second argument `maxd` is given.

The imaginary unit I is handled in a special way: it is replaced by the element `` `#I` `` of the algebraic extension field with minimal polynomial `` `\#I`^2 + 1 ``. If I occurs in the input, the result consists of polynomials over that extension field.

If a second argument `maxd` is given, d-th roots of rationals are replaced by elements of a suitable field extension of the rationals if $d \le maxd$. In the same way, nested fractional powers of rationals are replaced unless the denominator of some exponent exceeds `maxd`. In this case, the returned result consists of polynomials over a tower of extension fields over the rationals.

Examples

Example 1

In the simplest case (integer polynomial), the numerator equals the input, the denominator equals 1, and no substitutions are necessary:

```
polylib::makerat(x^2+3)
```

$$\text{poly}(x^2 + 3, [x]),\ \text{poly}(1, [x]),\ []$$

Example 2

Transcendental expressions are replaced by new identifiers. The result indicates on which variables the generated identifiers depend:

```
polylib::makerat(sin(u)/x)
```

$$\text{poly}(\text{X5}, [\text{X5}, x]),\ \text{poly}(x, [\text{X5}, x]),\ [\text{X5} = \sin(u)]$$

Example 3

Floating point numbers are considered transcendental:

```
polylib::makerat(0.27*x)
```

$$\text{poly}(\text{X8}\ x, [\text{X8}, x]),\ \text{poly}(1, [\text{X8}, x]),\ [\text{X8} = 0.27]$$

Example 4

By default, radicals are treated like transcendental subexpressions:

```
polylib::makerat(sqrt(2)/x)
```

$$\text{poly}(X11, [X11, x]),\ \text{poly}(x, [X11, x]),\ [X11 = \sqrt{2}]$$

Example 5

If a sufficiently large second argument is given, radicals are replaced by elements of algebraic extensions:

```
polylib::makerat(sqrt(2)/x, 2)
```

$$\text{poly}(X14, [x], \text{Dom::AlgebraicExtension}(\text{Dom::Rational}, X14^2 - 2 = 0, X14)),$$

$$\text{poly}(x, [x], \text{Dom::AlgebraicExtension}(\text{Dom::Rational}, X14^2 - 2 = 0, X14)),\ [X14 = \sqrt{2}]$$

Parameters

`a`

Polynomial over `Expr` or arithmetical expression

`l`

List or set of polynomials over `Expr` or arithmetical expressions

`maxd`

Positive integer

Return Values

`polylib::makerat` returns an expression sequence consisting of three operands:

- The first operand represents the numerator (or the list/set of numerators, respectively). It is a single polynomial if the call was `polylib::makerat(a)`, otherwise it is a set or list of polynomials (the same type as the input). The

polynomial(s) may have more indeterminates than the input. The coefficient ring is either `Expr` or a `Dom::AlgebraicExtension`.

- The second operand represents the denominator (or the list/set of denominators, respectively). It is of the same type as the first operand.
- The third operand is a list of equations.

See Also

MuPAD Functions

`rationalize`

polylib::minpoly

Approximate minimal polynomial

Syntax

```
polylib::minpoly(a, n, x)
```

Description

`polylib::minpoly(a, n, x)` computes a univariate polynomial `f` in the variable `x` of degree `n` with integer coefficients such that `a` equals a root of `f` up to the precision given by `DIGITS`, and such that the sum of squares of its coefficients is minimal among all polynomials with this property.

Environment Interactions

`polylib::minpoly` is sensitive to the environment variable `DIGITS`.

Examples

Example 1

We compute a polynomial of degree 4 that has a root close to PI (up to 6 decimal digits) and small integer coefficients:

```
DIGITS:=6: polylib::minpoly(PI, 4, x); delete DIGITS:
```

$$\mathrm{poly}(x^4 - 5\,x^3 + 5\,x^2 + 9\,x - 20, [x])$$

If the root has to be even closer to PI, bigger coefficients are needed:

```
DIGITS:=20: polylib::minpoly(PI, 4, x); delete DIGITS
```

$$\mathrm{poly}(-108\,x^4 - 1717\,x^3 + 6952\,x^2 - 258\,x - 4045, [x])$$

Parameters

a

Arithmetical expression that can be converted to a floating point number

n

Positive integer

x

Identifier

Return Values

`polylib::minpoly` returns a polynomial in `x`. Its coefficient ring is `Expr`, all of its coefficients are integers.

Algorithms

The problem reduces to finding a shortest integer vector in the lattice $\left\{ e_i + a^i e_{n+1} \;\middle|\; 0 \le i \le n \right\}$, where e_i denotes the vector with $e_i[j] = \delta_{i,j}$ (Kronecker symbol).
This problem is solved using the algorithm of Lenstra/Lenstra/Lovasz.

References

Lenstra/Lenstra/Lovasz, Factoring polynomials with rational coefficients, Math. Ann. 261(1982), pp. 515–534.

See Also

MuPAD Functions
`interpolate` | `lllint` | `stats::linReg`

polylib::Poly

Domain of polynomials

Syntax

Domain Creation

```
polylib::Poly([x1, …], <R>)
```

Description

`polylib::Poly([x1,...,xn], R)` creates the ring of polynomials in the unknowns `x1` through `xn` over the coefficient ring `R`. If the argument `R` is missing, `Expr` is used.

`polylib::Poly` is a facade domain; it has no domain elements. It serves only as a coefficient ring for polynomials.

The attempt to create an element of `polylib::Poly` results in a `DOM_POLY`.

The arithmetical operations of the domain are realized by the corresponding kernel methods.

Examples

Example 1

`polylib::Poly` can be used for defining polynomials in `x` whose coefficients are polynomials in `y`. Such polynomials must not be confused with bivariate polynomials in `x` and `y`.

```
delete x,y: e:= x*(y^2*2 + y) + 3*y:
poly(e, [x, y]);  poly(e, [x], polylib::Poly([y]))
```

$$\mathrm{poly}(2\,x\,y^2 + x\,y + 3\,y, [x, y])$$

$$\text{poly}((y+2\,y^2)\,x+3\,y,\,[x],\,\text{polylib::Poly}([y],\,\text{Expr}))$$

Parameters

x_1

Unknown

R

Admissible coefficient ring for polynomials. See `poly`.

Entries

"zero"	the zero polynomial
"one"	the constant polynomial one
"indets"	list of unknowns
"coeffRing"	the coefficient ring `R`

See Also

MuPAD Domains

`Dom::DistributedPolynomial`

polylib::primitiveElement

Primitive element for tower of field extensions

Syntax

```
polylib::primitiveElement(F, G)
```

Description

For given field extensions $F = K(\alpha)$ and $G = F(\beta)$, `polylib::primitiveElement(F, G)` returns a list consisting of a simple algebraic extension of K that is K-isomorphic to G, a symbol for a primitive element of that extension, and the images of α and β under some fixed K-isomorphism.

It is presumed that the extension is separable. Otherwise, it may happen that the algorithm does not terminate.

Examples

Example 1

Since the rational numbers are perfect, extensions of them can always be handled:

```
F := Dom::AlgebraicExtension(Dom::Rational, sqrt2^2 - 2):
G := Dom::AlgebraicExtension(F, sqrt3^2 - 3):
```

Now $G = Q(\sqrt{2}, \sqrt{3})$, and we use `polylib::primitiveElement` to find a primitive element for `G`:

```
polylib::primitiveElement(F, G)
```

$$\left[\text{Dom::AlgebraicExtension(Dom::Rational, } -10\,X1^2 + X1^4 + 1 = 0,\ X1),\ X1,\ \frac{X1^3}{2} - \frac{9\,X1}{2},\ \frac{11\,X1}{2} - \frac{X1^3}{2}\right]$$

This means that a primitive element X_1 of the extension is determined by its minimal polynomial X_1^4 - 10 X_1^2 + 1. The last two operands of the list are field elements whose squares are 2 and 3, respectively.

Example 2

The function works also for subdomains of `Dom::AlgebraicExtension`, e.g., Galois fields.

```
F := Dom::GaloisField(7, 2):
G := Dom::GaloisField(F, 2):
polylib::primitiveElement(F, G)
```

$$[\text{Dom::AlgebraicExtension}(\text{Dom::IntegerMod}(7),\ 3\ X5 - X5^2 + 2\ X5^3 + X5^4 - 1 = 0,\ X5),$$

$$X5,\ -3\ X5^3 + 3\ X5^2 - 3\ X5 - 2,\ X5]$$

Parameters

`F`

A field created by `Dom::AlgebraicExtension`

`G`

A field created by `Dom::AlgebraicExtension` with ground field `F`

Return Values

List consisting of four operands:

- a field `H` of type `Dom::AlgebraicExtension` over the same ground field as `F`;
- an identifier that equals the entry `H::variable`;
- an object of type `H` that satisfies the minimal polynomial for α;
- an object of type `H` that satisfies the minimal polynomial for β.

Algorithms

The chosen primitive element is $\alpha + s\,\beta$, where s is a positive integer.

See Also

MuPAD Domains
`Dom::AlgebraicExtension`

MuPAD Functions
`polylib::splitfield`

polylib::primpart

Primitive part of a polynomial

Syntax

```
polylib::primpart(f)
```

```
polylib::primpart(q)
```

```
polylib::primpart({xpr}, <{inds}>)
```

Description

`polylib::primpart(f)` returns the primitive part of the polynomial `f`.

If the input is a `polynomial`, the greatest common divisor of its coefficients is removed. The function `gcd` must be able to calculate this gcd.

If the first argument is an expression, it is converted into a polynomial in the indeterminates specified by the second argument, or in all of its indeterminates if no second argument is given. `polylib::primpart` returns `FAIL` if the expression cannot be converted into a polynomial.

For a rational number, its sign is returned.

Examples

Example 1

In the following example, a bivariate polynomial is given. Its coefficients are the integers 3, 6, and 9; the primitive part is obtained by dividing the polynomial by their gcd.

```
polylib::primpart(poly(6*x^3*y + 3*x*y + 9*y, [x, y]));
```

$$\mathrm{poly}(2\,x^3\,y + x\,y + 3\,y, [x, y])$$

However, consider the same polynomial viewed as a univariate polynomial in `x`. Its coefficients are polynomials in `y` in this case, and their gcd `3*y` is divided off.

```
polylib::primpart(poly(6*x^3*y + 3*x*y + 9*y, [x]));
```

$$\mathrm{poly}(2\,x^3 + x + 3, [x])$$

Example 2

`polylib::primpart` divides the coefficients by their gcd, but does not normalize the result. This must be done explicitly:

```
polylib::primpart(4*x*y + 6*x^3 + 6*x*y^2 + 9*x^3*y, [x])
```

$$\frac{x^3\,(9\,y+6)}{3\,y+2} + \frac{x\,(6\,y^2+4\,y)}{3\,y+2}$$

```
normal(polylib::primpart(4*x*y + 6*x^3 + 6*x*y^2 + 9*x^3*y, [x]))
```

$$3\,x^3 + 2\,y\,x$$

Parameters

`f`

Polynomial

`q`

Rational number

`xpr`

Expression

`inds`

List of identifiers

Return Values

`polylib::primpart` returns an object of the same type as the input, or `FAIL`.

Overloaded By

f

Algorithms

The primitive part of a polynomial f is a polynomial g whose coefficients are relatively prime such that $f = rg$ for some element r of the coefficient ring.

See Also

MuPAD Functions

`content` | `factor` | `gcd` | `icontent` | `irreducible`

polylib::randpoly

Create a random polynomial

Syntax

```
polylib::randpoly()
```

```
polylib::randpoly(<list>, <ring>, <Degree = n>, <Terms = k>, <Coeffs = f>, <Monic>)
```

Description

`polylib::randpoly()` returns a univariate random polynomial with integer coefficients; the global identifier `x` is used as the indeterminate.

`polylib::randpoly(list)` returns a random polynomial in all indeterminates given in `list`.

`polylib::randpoly(list, ring)` returns a random polynomial in the indeterminates given in `list` over the coefficient ring `ring`.

See `poly` for a detailed description of possible indeterminates and coefficient rings.

The polynomial is created by randomly choosing as many exponents as specified through the option `Terms` and then choosing random coefficients. It may of course happen that for some coefficient 0 is chosen, therefore the actual number of terms in the result can be smaller than the value of the option `Terms`.

If the option `Coeffs=f` is given, the random coefficients are generated by calling `f()`. Otherwise, if `ring` is `Expr`, the coefficients will be random integers in the range - 999, …, 999. If `ring` is a user-defined domain, it must have a method `"random"` to create the coefficients if no function is given.

If the option `Monic` is given, the resulting polynomial has exactly the specified degree and the leading coefficient is 1.

If the requested number of terms exceeds the maximal possible number of terms for the specified degree and number of variables, a warning is emitted and a dense polynomial is created.

Environment Interactions

Unless a generator is specified through the option `Coeffs`, `polylib::randpoly` uses `random` to create the exponents and coefficients. Therefore it is sensitive to the environment variable `SEED`.

Examples

Example 1

We generate a univariate random polynomial in the indeterminate `z`, and use the default values for the other options. Therefore the polynomial has integer coefficients, is of degree 5, and has 6 terms.

```
polylib::randpoly([z])
```

$$\text{poly}(-535\,z^5+916\,z^4+663\,z^3-764\,z^2-741\,z-65,\,[z])$$

Example 2

We create a bivariate random polynomial over the finite field with 7 elements. This works because `Dom::IntegerMod` has a `"random"` slot that generates random elements:

```
polylib::randpoly([x,y],Dom::IntegerMod(7),Degree=3,Terms=4);
```

$$\text{poly}(2\,x^3\,y^3+x^3\,y+4\,x\,y,\,[x,\,y],\,\text{Dom::IntegerMod}(7))$$

Parameters

`list`

List of indeterminates

`ring`

Coefficient ring

Options

`Degree`

Option, specified as `Degree = k`

The maximum degree the result can have in each variable. `k` must be a nonnegative integer. Default is 6.

`Terms`

Option, specified as `Terms = t`

Makes `polylib::randpoly` generate the sum of `t` random terms. `t` must be a positive integer or `infinity`. If `t` equals `infinity`, `polylib::randpoly` returns a dense polynomial. Default is 5.

`Coeffs`

Option, specified as `Coeffs = f`

Create the coefficients of the result by calling `f()`.

`Monic`

The created polynomial is monic, i.e., the leading coefficient is 1.

Return Values

Polynomial in the given indeterminates over the given ring. If no list of indeterminates is given, `[x]` is used. If no ring is given, `Expr` is used.

See Also

MuPAD Functions

`poly` | `random`

polylib::realroots

Isolate all real roots of a real univariate polynomial

Syntax

```
polylib::realroots(p)
```

```
polylib::realroots(p, eps)
```

Description

`polylib::realroots(p)` returns intervals isolating the real roots of the real univariate polynomial `p`.

`polylib::realroots(p, eps)` returns refined intervals approximating the real roots of `p` to the relative precision given by `eps`.

All coefficients of `p` must be real and numerical, i.e., either integers, rationals or floating-point numbers. Numerical symbolic objects such as `sqrt(2)`, `exp(10*PI)` etc. are accepted, if they can be converted to real floating-point numbers via `float`. The same holds for the precision goal `eps`.

The isolating intervals are ordered such that their centers are increasing, i.e., $a_i + b_i < a_{i+1} + b_{i+1}$.

The number `nops(realroots(p))` of intervals is the number of real roots of `p`. Multiple roots are counted only once. Cf. “Example 3” on page 25-38.

Isolating intervals may be quite large. The optional argument `eps` may be used to refine the intervals such that they approximate the real roots to a relative precision `eps`. With this argument the returned intervals satisfy $b_i - a_i \le \frac{\text{eps}\,|a_i+b_i|}{2}$, i.e., each center $\frac{(a_i+b_i)}{2}$ approximates a root with a relative precision `eps/2`.

Note: Some care should be taken when trying to obtain highly accurate approximations of the roots via small values of `eps`. Internally, bisectioning with exact rational

arithmetic is used to locate the roots to the precision `eps`. This process may take much more time than determining the isolating intervals without using the second argument `eps` in `polylib::realroots`. It may be faster to use moderate values of `eps` to obtain first approximations of the roots via `polylib::realroots`. These approximations may then be improved by a fast numerical solver such as `numeric::fsolve` with an appropriately high value of `DIGITS`. Cf. "Example 6" on page 25-40. However, note that `polylib::realroots` will always succeed in locating the roots to the desired precision eventually. Numerical solvers may fail or return a root not belonging to the interval which was used for the initial approximation.

Note: Unexpected results may be obtained when the polynomial contains irrational coefficients. Internally, any such coefficient c is converted to a floating-point number. This float is then replaced by an approximating rational number r satisfying $|r-c| \le \frac{1}{10^{\mathrm{DIGITS}}}\,|c|$. Finally, `polylib::realroots` returns rigorous bounds for the real roots of the rationalized polynomial. Despite the fact that all coefficients are approximated correctly to `DIGITS` decimal places this may change the roots drastically. In particular, multiple roots or clusters of poorly separated simple roots are very sensitive to small perturbations in the coefficients of the polynomial. See "Example 4" on page 25-39 and "Example 5" on page 25-39.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, if there are non-integer or non-rational coefficients in the polynomial. Any such coefficient is replaced by a rational number approximating the coefficient to `DIGITS` significant decimal places.

Examples

Example 1

We use a polynomial expression as input to `polylib::realroots`:

```
p := (x - 1/3)*(x - 1)*(x - 4/3)*(x - 2)*(x - 17):
```

```
polylib::realroots(p)
```

$$[[0, 1], [1, 1], [1, 2], [2, 2], [16, 32]]$$

The roots 1 and 2 are found exactly: the corresponding intervals have length 0. The other isolating intervals are quite large. We refine the intervals such that they approximate the roots to 12 decimal places. Note that this is independent of the current value of `DIGITS`, because no floating-point arithmetic is used:

```
polylib::realroots(p, 10^(-12))
```

$$\left[\left[\frac{1466015503701}{4398046511104}, \frac{733007751851}{2199023255552}\right], [1, 1], \left[\frac{1466015503701}{1099511627776}, \frac{733007751851}{549755813888}\right], [2, 2], [17, 17]\right]$$

We convert these exact bounds for the real roots to floating point approximations. Note that with the default value of `DIGITS=10` we ignore 2 of the 12 correct digits the rational bounds could potentially give:

```
map(%, map, float)
```

$$[[0.3333333333, 0.3333333333], [1.0, 1.0], [1.333333333, 1.333333333], [2.0, 2.0], [17.0, 17.0]]$$

```
delete p:
```

Example 2

Orthogonal polynomials of degree n have n simple real roots. We consider the Legendre polynomial of degree 5, available in the library `orthpoly` for orthogonal polynomials:

```
polylib::realroots(orthpoly::legendre(5, x), 10^(-DIGITS)):

map(%, float@op, 1)
```

$$[-0.906179846, -0.5384693101, 0.0, 0.5384693101, 0.9061798459]$$

Example 3

We consider a polynomial with a multiple root:

```
p := poly((x - 1/3)^3*(x - 1), [x])
```

$$\mathrm{poly}\left(x^4 - 2\,x^3 + \frac{4\,x^2}{3} - \frac{10\,x}{27} + \frac{1}{27}, [x]\right)$$

Note that only one isolating interval [0, 1] is returned for the triple root $\frac{1}{3}$:

```
polylib::realroots(p)
```

$$[[0, 1], [1, 1]]$$

```
delete p:
```

Example 4

We consider a polynomial with non-rational roots:

```
p := (x - 3)^2*(x - PI)^2:
```

Converting the result of `polylib::realroots` to floating-point numbers one sees that the exact roots `3, 3, PI, PI` are approximated only to 3 decimal places:

```
map(polylib::realroots(p, 10^(-10)), map, float)
```

$$[[2.998807805, 2.998807805], [3.001213582, 3.001213582], [3.140323518, 3.140323519], [3.142840401, 3.142840401]]$$

This is caused by the internal rationalization of the coefficients of `p`.

The intervals returned by `polylib::realroots(p, 10^(-10))` correctly locate the 4 exact roots of this rationalized polynomial to a precision of 10 digits. However, because all 4 roots are close, the small perturbations of the coefficients introduced by rationalization have a drastic effect on the location of the roots. In particular, rationalization splits the two original double roots into 4 simple roots.

```
delete p:
```

Example 5

We consider a further example involving non-exact coefficients. First we approximate the roots of a polynomial with exact coefficients:

```
p1 := (x - 1/3)^3*(x - 4/3):
```

```
map(polylib::realroots(p1, 10^(-10)), map, float)
```

```
[[0.3333333333, 0.3333333333], [1.333333333, 1.333333333]]
```

Now we introduce roundoff errors by replacing one entry by a floating-point approximation:

```
p2 := (x - 1.0/3)^3*(x - 4/3):
```

```
map(polylib::realroots(p2, 10^(-10)),map,float)
```

```
[[0.3332481323, 0.3332481323], [1.333333333, 1.333333333]]
```

In this example rationalization caused the triple root `1/3` to split into one real root and two complex conjugate roots.

```
delete p1, p2:
```

Example 6

We want to approximate roots to a precision of 1000 digits:

```
p := x^5 - 129/20*x^4 + 69/5*x^3 - 14*x^2 + 12*x - 8:
```

We recommend not to obtain the result directly by `polylib::realroots(p,10^(-1000))`, because the internal bisectioning process for refining crude isolating intervals converges only linearly. Instead, we compute first approximations of the roots to a precision of 10 digits:

```
approx := map(polylib::realroots(p, 10^(-10)), float@op, 1)
```

```
[1.489177599, 1.752191733, 3.255184556]
```

These values are used as starting points for a numerical root finder. The internal Newton search in `numeric::fsolve` converges quadratically and yields the high precision results much faster than `polylib::realroots`:

```
DIGITS := 1000:
```

```
roots := map(approx, x0 -> numeric::fsolve([p = 0], [x = x0]))
```

```
[[x = 1.489177598846870281338916114673844643894...],
      [x = 1.752191733304413195335101727880090131407...],
      [x = 3.255184555797733438479691333705558491124...]]
[[x =
1.489177598846870281338916114673844643894669835092536119855751693956799904640967549560
[x =
1.752191733304413195335101727880090131407549255515946934886764975651528596777530139410
[x =
3.255184555797733438479691333705558491124209926859892564938063100419466261488128732680
```

```
delete approx, DIGITS, roots, x0:
```

Parameters

p

A univariate polynomial: either an expression or a polyomial of domain type `DOM_POLY`.

eps

A (small) positive real number determining the size of the returned intervals.

Return Values

List of lists $[[a_1, b_1], [a_2, b_2], \ldots]$ with rational numbers $a_i \le b_i$ is returned. Lists with $a_i = b_i$ represent exact rational roots. Lists with $a_i < b_i$ represent open intervals containing exactly one real root. If the polynomial has no real roots, then the empty list `[ ]` is returned.

See Also

MuPAD Functions

`numeric::fsolve` | `numeric::polyroots` | `numeric::realroot` | `numeric::realroots`

polylib::representByElemSym

Represent symmetric by elementary symmetric polynomials

Syntax

```
polylib::representByElemSym(f, l)
```

Description

`polylib::representByElemSym(f, [x1,...,xn])` returns a polynomial `g` in the identifiers x_1 through x_n such that replacing each `xi` by the `i`-th elementary symmetric polynomial gives `f`.

The list `l` must have as many operands as `f` has indeterminates.

The result is `FAIL` if the input is not symmetric.

Examples

Example 1

The symmetric polynomial $x^2 + y^2$ can be written as $(x + y)^2$ - 2 $(x\,y)$:

```
polylib::representByElemSym(poly(x^2+y^2), [u,v]);
```

$$\mathrm{poly}(u^2 - 2\,v, [u, v])$$

Example 2

`polylib::representByElemSym` works over domains also:

```
f:=poly(x^2+y^2, Dom::IntegerMod(7)):
polylib::representByElemSym(f, [u,v])
```

$$\mathrm{poly}(u^2 + 5\,v,\ [u,\ v],\ \mathrm{Dom::IntegerMod}(7))$$

Parameters

`f`

Symmetric polynomial

`l`

List of indeterminates

Return Values

Result is a polynomial having the same coefficient ring as `f`.

Algorithms

It is a well-known theorem that every symmetric polynomial can be written in this way.

See Also

MuPAD Functions
`polylib::elemSym`

polylib::resultant

Resultant of two polynomials

Syntax

```
polylib::resultant(f, g, <x>)
```

```
polylib::resultant(fexpr, gexpr, <inds>, <x>)
```

Description

`polylib::resultant(f, g)` returns the resultant of `f` and `g` with respect to their first variable.

`polylib::resultant(f, g, x)` returns the resultant of `f` and `g` with respect to the variable `x`.

`polylib::resultant(fexpr, gexpr, inds, x)` returns the resultant of `fexpr` and `gexpr` with respect to the variable `x`; `fexpr` and `gexpr` are viewed as polynomials in the indeterminates `inds`.

Both input polynomials must have exactly the same second and third operand, i.e. their variables and coefficient rings must be identical.

If the arguments are expressions then these are converted into polynomials using `poly`. `polylib::resultant` returns `FAIL` if the expressions cannot be converted.

If the argument `inds` is missing, the input expressions are converted into polynomials in all indeterminates occurring in at least one of them. They are *not* independently converted, hence the conversion cannot result in two polynomials with different variables causing an error. See “Example 1” on page 25-45.

If the coefficient ring is a domain, it must have a `"_divide"` method.

If the coefficient ring is `Expr`, `polylib::resultant` returns an expression if called with two univariate polynomials. See “Example 2” on page 25-45.

For polynomials over `IntMod(n)`, the computation may stop with an error if `n` is not prime.

Examples

Example 1

If the input consists of expressions, the sets of indeterminates occurring in the expressions need not coincide:

```
polylib::resultant(a*x + c, c*x + d, x);
```

$$a\,d - c^2$$

Example 2

If the coefficient ring of two univariate input polynomials is `Expr`, the result is an expression:

```
polylib::resultant(poly(x^2 -1), poly(x + 1));
```

$$0$$

Parameters

`f, g`

Polynomials

`fexpr, gexpr`

Expressions

`x`

Indeterminate

`inds`

List of indeterminates

Return Values

If the input consists of polynomials in at least two variables, `polylib::resultant` returns a polynomial in one variable less than the input.

Overloaded By

p, q

Algorithms

The resultant of two polynomials is defined to be the determinant of their Sylvester matrix. A call to `polylib::resultant` is more efficient than consecutive calls to `linalg::sylvester` and `det`.

See Also

MuPAD Functions

`det` | `linalg::sylvester` | `polylib::discrim`

polylib::sortMonomials

Sorting monomials with respect to a term ordering

Syntax

```
polylib::sortMonomials(f)

polylib::sortMonomials(f, vars)

polylib::sortMonomials(f, ord)

polylib::sortMonomials(f, vars, ord)
```

Description

`polylib::sortMonomials(f, ord)` returns a list of all monomials constituting the polynomial `f`, sorted in descending order with respect to `ord`.

A monomial ordering may be: one of the identifiers `LexOrder`, `DegreeOrder`, `DegInvLexOrder`; or an object of type `Dom::MonomOrdering` or convertible to that type; or any object returning a number when called as `ord(m1,m2)` for two degree vectors `m1` and `m2`. A degree vector is a list of integers, as returned by `degreevec`.

If no order is given, the lexikographical order is used.

If no list of variables is given, all indeterminates of `f` are used.

Given two degree vectors, `m1` is considered to be greater than `m2` if and only if `ord(m1,m2)` is positive.

Examples

Example 1

The monomials of the polynomial below are compared using a monomial ordering from `Dom::MonomOrdering` .

```
polylib::sortMonomials(poly(x^2+x*y^3+2, [x,y]), DegRevLex(2))
```

$$[\mathrm{poly}(x\,y^3, [x, y]), \mathrm{poly}(x^2, [x, y]), \mathrm{poly}(2, [x, y])]$$

Parameters

`f`

Polynomial or polynomial expression

`vars`

Nonempty list of identifiers

`ord`

Monomial ordering

Return Values

List of polynomials or expressions of the same type as `f`.

Overloaded By

f

See Also

MuPAD Domains
`Dom::MonomOrdering`

MuPAD Functions
`lmonomial` | `nthmonomial`

polylib::splitfield

Splitting field of a polynomial

Syntax

```
polylib::splitfield(p)
```

Description

Given a polynomial p over a field K in one indeterminate X, `polylib::splitfield(p)` returns a simple field extension F of K and some elements a_1, ..., a_n of F, such that $\prod_{i=1}^{n}(X-\alpha_i)$ is an associate of p, and such that F is the smallest extension of K containing all of the a_i.

If the input is a polynomial expression, as in "Example 1" on page 25-49, it is treated as a polynomial over the rationals.

The polynomial `p` need not be irreducible.

The name for the primitive element of the field extension is generated using `genident` and is therefore different in every call of `polylib::splitfield`, even if the same polynomial is passed.

MuPAD must be able to factor polynomials over the coefficient field of `p`.

The coefficient field must be perfect. Otherwise, it may happen that `polylib::splitfield` does not terminate.

Examples

Example 1

We adjoin $\sqrt{-1}$ to the rationals:

```
polylib::splitfield(x^2+1)
```

$$[\text{Dom::AlgebraicExtension}(\text{Dom::Rational}, X1^2 + 1 = 0, X1), [X1, 1, -X1, 1]]$$

Example 2

A call to `polylib::splitfield` becomes more interesting for polynomials for of degree at least 3:

```
polylib::splitfield(x^3-2)
```

$$\left[\text{Dom::AlgebraicExtension}(\text{Dom::Rational}, X3^6 + 108 = 0, X3), \left[\frac{X3}{2} - \frac{X3^4}{36}, 1, \frac{X3^4}{18}, 1, -\frac{X3^4}{36} - \frac{X3}{2}, 1\right]\right]$$

Example 3

In this example, we work over the field of univariate rational functions (the quotient field of the univariate polynomials) over the rationals:

```
R:=Dom::DistributedPolynomial([x], Dom::Rational):
F:=Dom::Fraction(R):
f:=poly(y^3-x,[y],F):
polylib::splitfield(f)
```

$$\left[\text{Dom::AlgebraicExtension}(\text{Dom::Fraction}(\text{Dom::DistributedPolynomial}([x], \text{Dom::Rational}, \text{LexOrder})), X5^6 + 27x^2 = 0, X5), \left[\frac{X5}{2} - \frac{X5^4}{18x}, 1, \frac{X5^4}{9x}, 1, -\frac{X5}{2} - \frac{X5^4}{18x}, 1\right]\right]$$

Parameters

p

Univariate polynomial over a field or univariate polynomial expression

Return Values

`polylib::splitfield` returns a list of two operands: the first one is the splitting field of the polynomial, i.e. a `Dom::AlgebraicExtension` of the coefficient ring; the second one is a list of all roots of the polynomial in the splitting field, each root followed by its multiplicity.

See Also

MuPAD Functions

`evalp` | `factor`

polylib::sqrfree

Square-free factorization of polynomials

Syntax

```
polylib::sqrfree(f, <recollect>)
```

Description

`polylib::sqrfree(f)` returns the square-free factorization of `f`, that is, a factorization of f in the form $f = u\, p_1^{e_1} \ldots p_r^{e_r}$ with primitive and pairwise different square-free divisors p_i.

`polylib::sqrfree(f)` returns the square-free factorization of the polynomial f, that is, a factorization of f in the form $f = u\, f_1^{e_1} \ldots f_r^{e_r}$ with primitive and pairwise different square-free divisors f_i (i.e., $gcd(f_i, f_j) = 1$ for $i \neq j$).

u is a unit of the coefficient ring of f, and e_i are positive integers.

The result of `polylib::sqrfree` is an object of the domain type `Factored`. Let `g:= polylib::sqrfree(f)` be such an object. It is represented internally as the list `[u, f1, e1, ..., fr, er]` of odd length $2r + 1$.

You may extract the unit u and the terms $f_i^{e_i}$ by the ordinary index operator `[ ]`, i.e., `g[1] = u, g[2] = f1^e1, g[2] = f2^e2, ...`.

The calls `Factored::factors(g)` and `Factored::exponents(g)` return a list of the factors f_i and the exponents e_i ($1 \leq i \leq r$), respectively. The call `convert(g, DOM_LIST)` gives the internal representation of a factored object, i.e., the list `[u, f1, e1, ..., fr, er]`.

Note that the result of `polylib::sqrfree` is printed as an expression and behaves like that. As an example, the result of `polylib::sqrfree(x^2+2*x+1)` is the object printed as `(x+1)^2` which is of type `"_power"`.

Please read the help page of `Factored` for details.

The call `polylib::sqrfree(f, FALSE)` returns a square-free factorization of `f`, where the exponents e_i need not be pairwise different.

`polylib::sqrfree` can handle univariate and multivariate polynomials over `Expr`, residue class rings `IntMod(p)` with prime modulus `p`, domains representing a unique factorization domain of characteristic zero, and finite fields.

If the argument of `polylib::sqrfree` is an expression, its numerator and denominator are converted into polynomials in all occurring indeterminates.

These polynomials are regarded as polynomials over some extension of the rational numbers (i.e., over `Expr`, see `poly`). The choice of that extension follows the same rules as in the case of the function `factor`.

Factors of the denominator of an expression are indicated by negative multiplicities.

Examples

Example 1

The factors in a squarefree factorization are pairwise relatively prime, but they need not be irreducible:

```
polylib::sqrfree(
  2 - 2*x - 6*x^4 + 6*x^5 + 6*x^8 - 6*x^9 -2*x^12 + 2*x^13
)
```

$$2\,(x^3+x^2+x+1)^3\,(x-1)^4$$

Example 2

Even if a factorization into irreducibles has been found, irreducible factors with the same multiplicity are collected again:

```
polylib::sqrfree(x^6 + x^4*y^6 + x^2*y^2*9)
```

$$(x\,(x^2+3\,y))^2$$

You can avoid this by giving a second argument:

```
polylib::sqrfree(x^6 + x^4*y*6 + x^2*y^2*9, FALSE)
```

$$x^2\,(x^2+3\,y)^2$$

Example 3

`polylib::sqrfree` works also for polynomials:

```
polylib::sqrfree(poly(2 + 5*x + 4*x^2 + x^3))
```

$$\mathrm{poly}(x+2,\,[x])\,\mathrm{poly}(x+1,\,[x])^2$$

Parameters

`f`

A polynomial or an arithmetical expression

`recollect`

TRUE or FALSE

Return Values

Factored object, i.e., an object of the domain type `Factored`.

See Also

MuPAD Functions

`content` | `factor` | `Factored` | `irreducible` | `polylib::primpart`

polylib::subresultant

Subresultants of two polynomials

Syntax

```
polylib::subresultant(f, g, <x>, <i>)

polylib::subresultant(fexpr, gexpr, <x>, <i>)
```

Description

`polylib::subresultant(f, g)` returns the table of subresultants of polynomials `f` and `g` with respect to their first variable.

`polylib::subresultant(f, g, i)` returns the `i`th subresultant of polynomials `f` and `g` with respect to their first variable.

`polylib::subresultant(f, g, x)` returns the table of subresultants of polynomials `f` and `g` with respect to the variable `x`.

`polylib::subresultant(f, g, x, i)` returns the `i`th subresultant of polynomials `f` and `g` with respect to the variable `x`.

`polylib::subresultant(fexpr, gexpr, x)` returns the table of subresultants of polynomial expressions `fexpr` and `gexpr` with respect to the variable `x`.

`polylib::subresultant(fexpr, gexpr, x, i)` returns the `i`th subresultant of polynomial expressions `fexpr` and `gexpr` with respect to the variable `x`.

`polylib::subresultant` returns a particular subresultant or a table of all subresultants.

The variables and coefficient rings of both input polynomials must be identical.

The 0th subresultant is the resultant of two polynomials. See “Example 1” on page 25-56

If you do not specify the variable when computing the subresultants of two polynomials, `polylib::subresultant` returns subresultants of the polynomials with respect to their first variable. See “Example 2” on page 25-57.

If you call `polylib::subresultant` for polynomial expressions, you must specify the variable with respect to which you want to compute subresultants. MuPAD uses the `poly` function to convert polynomial expressions to polynomials with the specified variable. The system also converts computed subresultants back to polynomial expressions.

If `poly` cannot convert expressions to polynomials, `polylib::subresultant` returns `FAIL`.

If the degree of the polynomial `f` is less than the degree of the polynomial `g`, the `polylib::subresultant` function interchanges `f` and `g`.

If the coefficient ring is a domain, it must have a `_divide` method.

Examples

Example 1

If you do not specify which subresultant to return, `polylib::subresultant` returns the table of all subresultants:

```
f := poly(3*x^4 + 3*x^3 + 4):
g := poly(x^4 + x^3 + x^2 + x + 1):
polylib::subresultant(f, g)
```

0	poly(205, [x])
1	poly(9 x − 39, [x])
2	poly(9 x + 9 x^2 − 3, [x])
3	poly(3 x + 3 x^2 − 1, [x])
4	poly(3 x^3 + 3 x^4 + 4, [x])

You can specify the number of the subresultant that you want to compute. For example, compute the 0th subresultant of the polynomials `f` and `g`:

```
polylib::subresultant(f, g, 0)
```

$$\text{poly}(205, [x])$$

The 0th subresultant is also the resultant of the polynomials:

```
polylib::resultant(f, g)
```

$$205$$

Example 2

`polylib::subresultant` handles multivariate polynomials and polynomial expressions. When you compute subresultants of multivariate polynomials or polynomial expressions, you can specify the variable with respect to which you want to compute subresultants:

```
f := poly(3*x^4*y + 4*z^2):
g := poly(x^4 + x^3*y^3*z^3):
polylib::subresultant(f, g, z)
```

0	$\text{poly}(27\,x^{18}\,y^9 + 64\,x^8, [x, y])$
1	$\text{poly}(16\,x^4 - 12\,x^7\,y^4\,z, [x, y])$
2	$\text{poly}(3\,x^4\,y + 4\,z^2, [x, y])$
3	$\text{poly}(x^4 + x^3\,y^3\,z^3, [x, y])$

For multivariate polynomials, specifying the variable is not necessary. If you do not specify the variable when computing the subresultants of two polynomials, `polylib::subresultant` returns subresultants of the polynomials with respect to their first variable:

```
f := poly(3*x^4*y + 4*z^2):
g := poly(x^4 + x^3*y^3*z^3):
```

```
polylib::subresultant(f, g)
```

0	$\mathrm{poly}(192\,y^{13}\,z^{18}+256\,z^{8},[y,z])$
1	$\mathrm{poly}(48\,y^{7}\,z^{10}+48\,x\,y^{4}\,z^{7},[y,z])$
2	$\mathrm{poly}(12\,y^{7}\,z^{8}+12\,x\,y^{4}\,z^{5},[y,z])$
3	$\mathrm{poly}(-4\,z^{2}+3\,x^{3}\,y^{4}\,z^{3},[y,z])$
4	$\mathrm{poly}(3\,x^{4}\,y+4\,z^{2},[y,z])$

If you call `polylib::subresultant` for polynomial expressions, you must specify the variable with respect to which you want to compute subresultants:

```
f := 3*x^4*y + 4*z^2:
g := x^4 + x^3*y^3*z^3:
polylib::subresultant(f, g)
```

```
Error: A variable is missing. [polylib::subresultant]
```

```
polylib::subresultant(f, g, x)
```

0	$192\,y^{13}\,z^{18}+256\,z^{8}$
1	$48\,y^{7}\,z^{10}+48\,x\,y^{4}\,z^{7}$
2	$12\,y^{7}\,z^{8}+12\,x\,y^{4}\,z^{5}$
3	$-4\,z^{2}+3\,x^{3}\,y^{4}\,z^{3}$
4	$3\,x^{4}\,y+4\,z^{2}$

Parameters

`f, g`

Polynomials over `Expr` (the ring of arbitrary MuPAD expressions)

`fexpr, gexpr`

Polynomial expressions

`x`

An indeterminate

`i`

A nonnegative integer

Return Values

Subresultant of two polynomials (or polynomial expressions) or a table of subresultants.

Overloaded By

p, q

See Also

MuPAD Functions

`det` | `linalg::sylvester` | `polylib::discrim` | `polylib::resultant`

polylib::support

Support of a polynomial

Syntax

```
polylib::support(p)
```

Description

`polylib::support(p)` returns the support of `p`, that is, the list of indices with non zero coefficient in `p`.

Examples

Example 1

The support of a multivariate polynomial is the list of the degree vectors of its terms:

```
polylib::support(poly(x*y*z + x + 1, [x, y, z]))
```

```
[[1, 1, 1], [1, 0, 0], [0, 0, 0]]
```

For a univariate polynomial, the support is the list of the degrees of its terms. In the following polynomial `x` appears with degrees `3`, `1`, and `0`:

```
polylib::support(Dom::UnivariatePolynomial(x)(x^3*y*z + x + 1))
```

```
[3, 1, 0]
```

Parameters

p

A polynomial.

Return Values

List of elements of the support.

Pref – User Preferences

Pref::abbreviateOutput
Pref::alias
Pref::autoExpansionLimit
Pref::autoPlot
Pref::callBack
Pref::callOnExit
Pref::floatFormat
Pref::fourierParameters
Pref::heavisideAtOrigin
Pref::ignoreNoDebug
Pref::keepOrder
Pref::kernel
Pref::maxMem
Pref::maxTime
Pref::output
Pref::outputDigits
Pref::postInput
Pref::postOutput
Pref::report
Pref::trailingZeroes
Pref::typeCheck
Pref::userOptions
Pref::verboseRead
Pref::warnDeadProcEnv

Pref::abbreviateOutput

Controls the use of abbreviations in outputs

Syntax

```
Pref::abbreviateOutput(TRUE)

Pref::abbreviateOutput(FALSE)

Pref::abbreviateOutput(NIL)

Pref::abbreviateOutput()
```

Description

When displaying results, MuPAD by default finds common subexpressions and replaces them with abbreviations. See “Example 1” on page 26-2.

If you want to see the results without abbreviations, use the `Pref::abbreviateOutput(FALSE)` command. See “Example 2” on page 26-3.

The `Pref::abbreviateOutput()` command shows whether abbreviations are enabled or disabled. See “Example 3” on page 26-4.

To restore the default setting, use the `Pref::abbreviateOutput(NIL)` command. See “Example 4” on page 26-5.

The output of the `Pref::abbreviateOutput` command itself displays the previous setting. You can save this previous setting and switch to a new setting in a single call of `Pref::abbreviateOutput`. See “Example 5” on page 26-5.

Examples

Example 1

By default, you can see the abbreviations in long outputs:

```
solve(a*x^3 + b*x + c, x, MaxDegree = 3, IgnoreSpecialCases)
```

$$\left\{ \sigma_1 - \frac{b}{3\,a\,\sigma_1},\ \frac{b}{6\,a\,\sigma_1} - \frac{\sigma_1}{2} - \frac{\sqrt{3}\left(\sigma_1 + \frac{b}{3\,a\,\sigma_1}\right) i}{2},\ \frac{b}{6\,a\,\sigma_1} - \frac{\sigma_1}{2} + \frac{\sqrt{3}\left(\sigma_1 + \frac{b}{3\,a\,\sigma_1}\right) i}{2} \right\}$$

where

$$\sigma_1 = \left(\sqrt{\frac{b^3}{27\,a^3} + \frac{c^2}{4\,a^2}} - \frac{c}{2\,a} \right)^{1/3}$$

Example 2

Setting `Pref::abbreviateOutput(FALSE)`, you can disable the abbreviations in outputs:

```
Pref::abbreviateOutput(FALSE):
solve(a*x^3 + b*x + c, x, MaxDegree = 3, IgnoreSpecialCases)
```

$$\left\{\left(\sqrt{\frac{b^3}{27\,a^3}+\frac{c^2}{4\,a^2}}-\frac{c}{2\,a}\right)^{1/3}-\frac{b}{3\,a\left(\sqrt{\frac{b^3}{27\,a^3}+\frac{c^2}{4\,a^2}}-\frac{c}{2\,a}\right)^{1/3}},\right.$$

$$\frac{b}{6\,a\left(\sqrt{\frac{b^3}{27\,a^3}+\frac{c^2}{4\,a^2}}-\frac{c}{2\,a}\right)^{1/3}}-\frac{\left(\sqrt{\frac{b^3}{27\,a^3}+\frac{c^2}{4\,a^2}}-\frac{c}{2\,a}\right)^{1/3}}{2}$$

$$-\frac{\sqrt{3}\left(\left(\sqrt{\frac{b^3}{27\,a^3}+\frac{c^2}{4\,a^2}}-\frac{c}{2\,a}\right)^{1/3}+\frac{b}{3\,a\left(\sqrt{\frac{b^3}{27\,a^3}+\frac{c^2}{4\,a^2}}-\frac{c}{2\,a}\right)^{1/3}}\right)i}{2},$$

$$\frac{b}{6\,a\left(\sqrt{\frac{b^3}{27\,a^3}+\frac{c^2}{4\,a^2}}-\frac{c}{2\,a}\right)^{1/3}}-\frac{\left(\sqrt{\frac{b^3}{27\,a^3}+\frac{c^2}{4\,a^2}}-\frac{c}{2\,a}\right)^{1/3}}{2}$$

$$\left.+\frac{\sqrt{3}\left(\left(\sqrt{\frac{b^3}{27\,a^3}+\frac{c^2}{4\,a^2}}-\frac{c}{2\,a}\right)^{1/3}+\frac{b}{3\,a\left(\sqrt{\frac{b^3}{27\,a^3}+\frac{c^2}{4\,a^2}}-\frac{c}{2\,a}\right)^{1/3}}\right)i}{2}\right\}$$

Example 3

You can check the current setting:

```
Pref::abbreviateOutput()
```

```
FALSE
```

Example 4

You can restore the default setting:

```
Pref::abbreviateOutput(NIL):
Pref::abbreviateOutput()
```

$$\text{TRUE}$$

Example 5

You can save the current setting and switch it to a new one in one function call:

```
old := Pref::abbreviateOutput(FALSE):
solve(x^3 + x + 1 = 0, x, MaxDegree = 3)
```

$$\left\{ \left(\frac{\sqrt{31}\sqrt{108}}{108}-\frac{1}{2}\right)^{1/3}-\frac{1}{3\left(\frac{\sqrt{31}\sqrt{108}}{108}-\frac{1}{2}\right)^{1/3}},\ \frac{1}{6\left(\frac{\sqrt{31}\sqrt{108}}{108}-\frac{1}{2}\right)^{1/3}}-\frac{\left(\frac{\sqrt{31}\sqrt{108}}{108}-\frac{1}{2}\right)^{1/3}}{2}-\frac{\sqrt{3}\left(\frac{1}{3\left(\frac{\sqrt{31}\sqrt{108}}{108}-\frac{1}{2}\right)^{1/3}}+\left(\frac{\sqrt{31}\sqrt{108}}{108}-\frac{1}{2}\right)^{1/3}\right)i}{2},\ \frac{1}{6\left(\frac{\sqrt{31}\sqrt{108}}{108}-\frac{1}{2}\right)^{1/3}}-\frac{\left(\frac{\sqrt{31}\sqrt{108}}{108}-\frac{1}{2}\right)^{1/3}}{2}+\frac{\sqrt{3}\left(\frac{1}{3\left(\frac{\sqrt{31}\sqrt{108}}{108}-\frac{1}{2}\right)^{1/3}}+\left(\frac{\sqrt{31}\sqrt{108}}{108}-\frac{1}{2}\right)^{1/3}\right)i}{2} \right\}$$

You can restore the saved setting:

```
Pref::abbreviateOutput(old):
solve(x^3 + x + 1 = 0, x, MaxDegree = 3)
```

$$\left\{\sigma_1 - \frac{1}{3\,\sigma_1},\ \frac{1}{6\,\sigma_1} - \frac{\sigma_1}{2} - \frac{\sqrt{3}\left(\frac{1}{3\,\sigma_1} + \sigma_1\right) i}{2},\ \frac{1}{6\,\sigma_1} - \frac{\sigma_1}{2} + \frac{\sqrt{3}\left(\frac{1}{3\,\sigma_1} + \sigma_1\right) i}{2}\right\}$$

where

$$\sigma_1 = \left(\frac{\sqrt{31}\,\sqrt{108}}{108} - \frac{1}{2}\right)^{1/3}$$

Return Values

Previously set value TRUE or FALSE

See Also

MuPAD Functions

`output::asciiAbbreviate` | `output::subexpr`

Pref::alias

Controls the output of aliased expressions

Syntax

```
Pref::alias(TRUE)
```

```
Pref::alias(FALSE)
```

```
Pref::alias(NIL)
```

```
Pref::alias()
```

Description

An `alias` is an abbrevation for a MuPAD expression. If `Pref::alias` is enabled, the `alias` abbrevations will be used for output.

`Pref::alias()` returns the current value.

`Pref::alias(TRUE)` switches the usage of `alias` abbrevations in outputs on. This is the default setting.

`Pref::alias(FALSE)` switches the usage of aliases in outputs off.

`Pref::alias(NIL)` restores the default value which is `TRUE`.

`Pref::alias` has no effect on `print` and `fprint`.

Environment Interactions

`Pref::alias` changes the output of aliased expressions.

Examples

Example 1

If an aliased expression occurs in output, it is replaced by the alias abbrevation:

```
alias(X = a + b):
X, a + b
```

$$X, X$$

This only works if the syntactical structure of expression matches the aliased expression:

```
2*X
```

$$2\,a+2\,b$$

`prog::exprtree` shows that `2*X` does not contain `a + b` any more:

```
prog::exprtree(X): prog::exprtree(2*X):
```

```
_plus
|
+-- a
|
`-- b
```

```
_plus
|
+-- _mult
|   |
|   +-- a
|   |
|   `-- 2
|
`-- _mult
    |
    +-- b
    |
    `-- 2
```

The same holds for `X+c`:

```
X + c; prog::exprtree(X + c):
```

$$a+b+c$$

```
_plus
|
+-- a
|
+-- b
|
`-- c
```

With `Pref::alias(FALSE)` the back translation of aliases in the output is disabled:

```
Pref::alias(FALSE):
X
```

$$a + b$$

`Pref::alias` has no effect on `print` and `fprint` outputs:

```
Pref::alias(TRUE):
print(X):
```

$$a + b$$

Return Values

Previously set value

See Also

MuPAD Functions

`alias` | `expr2text` | `fprint` | `print`

Pref::autoExpansionLimit

Set a limit for automatic expansions

Syntax

```
Pref::autoExpansionLimit(n)
```

```
Pref::autoExpansionLimit(NIL)
```

```
Pref::autoExpansionLimit()
```

Description

`Pref::autoExpansionLimit(n)` sets a limit for the size of the arguments up to which the functions `bernoulli`, `Ei`, `euler`, `fact`, `fact2`, `gamma`, `harmonic`, `igamma`, `psi`, and `zeta` produce explicit results. Cf. “Example 1” on page 26-10.

It also sets a limit for the exponent up to which real and imaginary parts of powers are computed explicitly. Cf. “Example 2” on page 26-12.

Use `expand` for larger arguments if explicit results are desired. Cf. “Example 1” on page 26-10.

The call `Pref::autoExpansionLimit()` returns the current value of the limit without changing it.

The call `Pref::autoExpansionLimit(NIL)` resets the limit to its default value 1000.

Examples

Example 1

The functions `bernoulli`, `euler`, `gamma`, `zeta` etc. automatically produce explicit results if the arguments are not too large:

```
bernoulli(22),
```

```
euler(24),
gamma(26),
zeta(28)
```

$$\frac{854513}{138},\ 1551453416355708690 5,\ 1551121004333098598400000 0,\ \frac{6785560294\,\pi^{28}}{5646536601700762736718 75}$$

These functions return symbolic answers when the argument is larger than the limit set by `Pref::autoExpansionLimit`:

```
Pref::autoExpansionLimit()
```

$$1000$$

```
bernoulli(1002),
euler(2002),
gamma(3001),
zeta(4001)
```

$$\mathrm{bernoulli}(1002),\ \mathrm{euler}(2002),\ \Gamma(3001),\ \zeta(4001)$$

We reduce this limit:

```
Pref::autoExpansionLimit(20):

bernoulli(22),
euler(24),
gamma(26),
zeta(28)
```

$$\mathrm{bernoulli}(22),\ \mathrm{euler}(24),\ \Gamma(26),\ \zeta(28)$$

We can use `expand` to obtain explicit results:

```
expand(bernoulli(22)),
expand(euler(24)),
expand(gamma(26)),
expand(zeta(28))
```

$$\frac{854513}{138},\ 1551453416355708690 5,\ 1551121004333098598400000 0,\ \frac{6785560294\,\pi^{28}}{5646536601700762736718 75}$$

We restore the default value:

```
Pref::autoExpansionLimit(NIL):
```

Example 2

If binomial expansion is needed, the closed formula for the real part of an expression can become quite large:

```
Re((a+sqrt(2)*I)^6) assuming a in R_
```

$$\sqrt{2}\,(\sqrt{2}\,\sigma_2 - a\,\sigma_1) - a\,(\sqrt{2}\,\sigma_1 + a\,\sigma_2)$$

where

$$\sigma_1 = a\,(2\sqrt{2}\,a^2 + \sqrt{2}\,(a^2-2)) - \sqrt{2}\,(4a - a\,(a^2-2))$$

$$\sigma_2 = \sqrt{2}\,(2\sqrt{2}\,a^2 + \sqrt{2}\,(a^2-2)) + a\,(4a - a\,(a^2-2))$$

Thus, for exponents beyond `Pref::autoExpansionLimit()`, no expansion is carried out:

```
Re((a+sqrt(2)*I)^123456) assuming a in R_
```

$$\Re((a+\sqrt{2}\,i)^{123456})$$

Parameters

n

The limit: a positive numerical real value

Return Values

Previously defined limit.

See Also

MuPAD Functions

`bernoulli` | `Ei` | `euler` | `fact` | `fact2` | `gamma` | `harmonic` | `igamma` | `Im` | `psi` | `Re` | `zeta`

Pref::autoPlot

Automatically plot graphical objects

Syntax

```
Pref::autoPlot(TRUE)
```

```
Pref::autoPlot(FALSE)
```

```
Pref::autoPlot(NIL)
```

```
Pref::autoPlot()
```

Description

`Pref::autoPlot(TRUE)` makes graphical objects be plotted instead of typeset.

By default, graphical objects such as `plot::Function3d` are output just like any other MuPAD object, i.e., as a rendered representation of the input. After setting `Pref::autoPlot(TRUE)`, graphical objects and sequences of such objects are automatically rendered instead, as if the user had written `plot(...)`.

This setting only works when typesetting is enabled.

Return Values

Previously set value

See Also

MuPAD Functions
`plot`

Pref::callBack

Informations during evaluation

Syntax

```
Pref::callBack(<func>)
```

Description

The function `func` defined by `Pref::callBack(func)` will be called permanently, when the MuPAD kernel works. Therewith informations can be displayed to inform the user.

A call of `Pref::callBack` without arguments returns the current value. The argument `NIL` resets the default value, which is `NIL`.

Examples

Example 1

The following combination of `Pref::postInput` (initialization) and time count with `Pref::callBack` shows the seconds during evaluating.

```
Pref::postInput(proc() begin START:= time(); TIME:= START end_proc):
Pref::callBack(proc()
                begin
                  if time() - TIME > 1000 then // 1 sec
                    TIME:= TIME+1000;
                    print(floor((time() - START)/1000))
                  end_if
                end_proc):
NOW:= time():
while time() - NOW <= 10000 do 1 end_while:
```

```
     1
```

```
2

3

4

5

6

7

8

9
```

Parameters

`func`

Function to display informations

Return Values

Previously defined function

See Also

MuPAD Functions

`Pref::postInput` | `Pref::postOutput` | `Pref::report`

Pref::callOnExit

Defines an exit handler

Syntax

```
Pref::callOnExit(f)
```

```
Pref::callOnExit(list)
```

```
Pref::callOnExit(NIL)
```

```
Pref::callOnExit()
```

Description

`Pref::callOnExit(f)` defines a function `f` which is called on exit or `reset` of MuPAD.

`Pref::callOnExit(list)` defines a list of functions which are executed in the order of their occurrence in `list` on exit of MuPAD.

`Pref::callOnExit(NIL)` sets the default value, which is `NIL`.

`Pref::callOnExit()` returns the current value.

Parameters

`f`

A function

`list`

A list of functions

Return Values

`Pref::callOnExit` returns the previously defined function, list of functions, or `NIL`.

Algorithms

`Pref::callOnExit` can be used to send communication modules a disconnect message or to remove temporary user-defined files when leaving MuPAD.

See Also

MuPAD Functions
`Pref::postOutput | reset`

Pref::floatFormat

Representation of floating-point numbers

Syntax

```
Pref::floatFormat(mode)

Pref::floatFormat(NIL)

Pref::floatFormat()
```

Description

`Pref::floatFormat` controls the output format of floating-point numbers.

The representation mode can be one of the characters `"e"`, `"f"`, `"g"`, `"h"`, or `"x"`. These are the standard `C`-command `printf` switches. Their meaning is:

- "e":

  exponential representation (floating-point representation, “scientific format”).
- "f":

  decimal representation without exponents.
- "g":

  a mix between `"e"` and `"f"`. Numbers x satisfying $\frac{1}{10^{\mathrm{DIGITS}}} \le |x| \le 10^{\mathrm{DIGITS}}$ are displayed without exponents. All other numbers are displayed in floating-point representation.
- "h" or "x":

  hexadecimal representation, except for `expr2text` and typesetting, which fall back to `"g"`.

The default value is `"g"`.

`Pref::floatFormat()` returns the current mode without changing it. The call `Pref::floatFormat(NIL)` resets to the default value `"g"`.

Examples

Example 1

The exponential representation of a floating-point number consists of its sign, its mantissa and its exponent:

```
Pref::floatFormat("e"):
12345.67890, -0.00012345
```

$$12345.6789,\ -0.00012345$$

Without exponents, the size of a number is indicated by trailing or leading zeroes:

```
Pref::floatFormat("f"):
7.0*10^21, 7.0/10^21
```

$$7.0\ 10^{21},\ 7.0\ 10^{-21}$$

The mixed representation:

```
Pref::floatFormat("g"):
1e-10, 9.99e-11
```

$$0.0000000001,\ 9.99\ 10^{-11}$$

```
2.0^36, 2.0^37
```

$$68719476744.0,\ 1.374389535\ 10^{11}$$

Hexadecimal display is ignored in typeset output and `expr2text`:

```
Pref::floatFormat("h"):
12345.67890, 0.00012345;
```

```
expr2text(12345.67890, 0.00012345)
```

```
12345.6789, 0.00012345
```

```
"12345.6789, 0.00012345"
```

Hexadecimal display is used in the ASCII `print` output:

```
PRETTYPRINT := FALSE:
print(Plain, 12345.67890, 0.00012345);
PRETTYPRINT := TRUE:
```

```
3.039adcc63f141208@3, 8.1725b672ee34260@-4
```

The representation is reset to the default mode:

```
Pref::floatFormat(NIL):
```

Parameters

`mode`

One of the character strings `"e"`, `"f"`, `"g"`, `"h"`, or `"x"`

Return Values

Previously defined representation mode

See Also

MuPAD Functions

`DIGITS` | `Pref::outputDigits` | `Pref::trailingZeroes` | `print`

Pref::fourierParameters

Specify parameters for Fourier and inverse Fourier transforms

Syntax

```
Pref::fourierParameters(c, s)

Pref::fourierParameters([c, s])

Pref::fourierParameters(NIL)

Pref::fourierParameters()
```

Description

`Pref::fourierParameters(c, s)`, or the equivalent call `Pref::fourierParameters([c, s])`, specifies parameters used by the `fourier` and `ifourier` functions when computing Fourier and inverse Fourier transforms. See "Example 1" on page 26-23.

The Fourier transform of the expression `f = f(t)` with respect to the variable `t` at the point `w` is defined as follows:

$$F(w) = c \int_{-\infty}^{\infty} f(t)\, e^{i\,s\,w\,t}\, dt$$

.

The inverse Fourier transform of the expression `F = F(w)` with respect to the variable `w` at the point `t` is defined as follows:

$$f(t) = \frac{|s|}{2\,\pi\,c} \int_{-\infty}^{\infty} F(w)\, e^{-i\,s\,w\,t}\, dw$$

.

`c` and `s` are the parameters of the Fourier transform controlled by `Pref::fourierParameters`.

By default, `c = 1` and `s = -1`. Other common choices for the parameter `c` are $\frac{1}{2\pi}$ or $\frac{1}{\sqrt{2\pi}}$. Other common choices for the parameter `s` are 1, -2π, or 2π.

`Pref::fourierParameters()` returns the current values of the Fourier parameters without changing them.

`Pref::fourierParameters(NIL)` restores the default settings `c = 1`, `s = -1`.

`Pref::fourierParameters` also controls the parameters used by the `fourier::addpattern` and `ifourier::addpattern` functions. See "Example 2" on page 26-24.

Environment Interactions

Changing Fourier parameters using `Pref::fourierParameters` can affect results returned by `fourier` and `ifourier` in the current MuPAD session.

Examples

Example 1

Compute the Fourier transform of this expression using the default values `c = 1`, `s = -1` of the Fourier parameters:

```
assume(Re(a) > 0):
fourier(t*exp(-a*t^2), t, w)
```

$$-\frac{\sqrt{\pi}\, w\, e^{-\frac{w^2}{4a}}\, i}{2\, a^{3/2}}$$

Use `Pref::fourierParameters` to change the values of the Fourier parameters to `c = 1`, `s = 1`. Then compute the Fourier transform of the same expression again:

```
Pref::fourierParameters(1, 1):
```

```
fourier(t*exp(-a*t^2), t, w)
```

$$\frac{\sqrt{\pi}\, w\, e^{-\frac{w^2}{4a}}\, i}{2\, a^{3/2}}$$

Change the values of the Fourier parameters to $\frac{1}{2\pi}$ and 1. Compute the Fourier transform using these values:

```
Pref::fourierParameters(1/(2*PI), 1):
fourier(t*exp(-a*t^2), t, w)
```

$$\frac{w\, e^{-\frac{w^2}{4a}}\, i}{4\sqrt{\pi}\, a^{3/2}}$$

For further computations, restore the default values of the Fourier transform parameters:

```
Pref::fourierParameters(NIL):
```

Example 2

Use the default values of the Fourier transform parameters:

```
Pref::fourierParameters()
```

$$[1, -1]$$

Add this new Fourier transform pattern for the function `foo`:

```
fourier::addpattern(foo(t), t, w, bar(w)):
fourier(foo(t), t, w)
```

$$\mathrm{bar}(w)$$

The Fourier pair `(foo, bar)` is assumed to be valid for the current values of the Fourier parameters. When changing these parameters, you change the definition of the Fourier transform. Therefore, after changing Fourier parameters, the transform of `foo(t)` is not `bar(w)` anymore. The `fourier` function computes the result which is valid for the new parameters:

```
Pref::fourierParameters(c, s):
fourier(foo(t), t, w)
```

$$c\,\mathrm{bar}(-s\,w)$$

Now restore the Fourier transform parameters to their default values 1 and -1:

```
Pref::fourierParameters(NIL):
```

Parameters

c

Arithmetical expression

s

Arithmetical expression

Return Values

List containing the previously set values of `c` and `s`

See Also

MuPAD Functions

`fourier` | `fourier::addpattern` | `ifourier` | `ifourier::addpattern`

Pref::heavisideAtOrigin

Set value of Heaviside function at origin

Syntax

```
Pref::heavisideAtOrigin(val)

Pref::heavisideAtOrigin(NIL)

Pref::heavisideAtOrigin()
```

Description

`Pref::heavisideAtOrigin(val)` sets the value of the `heaviside` function at the origin and returns the old value. See “Example 1” on page 26-26. The default value of `heaviside` at the origin is 1/2. Other common choices for `heaviside(0)` are 0 or 1. This preference can affect the output of functions that call `heaviside`.

`Pref::heavisideAtOrigin()` returns the current value of `heaviside(0)`.

`Pref::heavisideAtOrigin(NIL)` restores the default setting `heaviside(0) = 1/2`.

Environment Interactions

Changing the value of `heaviside(0)` using `Pref::heavisideAtOrigin` can affect results returned by functions that call `heaviside`, such as `ztrans(heaviside(x),x,z)`.

Examples

Example 1

`Pref::heavisideAtOrigin` controls the value of the `heaviside` function at the origin. The default value is 1/2. Other common choices are 0 or 1.

Return the value of `heaviside(0)` and `ztrans(heaviside(x),x,z)`.

```
heaviside(0);
ztrans(heaviside(x),x,z)
```

$$\frac{1}{2}$$

$$\frac{1}{z-1}+\frac{1}{2}$$

Set `heaviside(0)` to `1` using `Pref::heavisideAtOrigin`. Store the old value returned to restore it later.

```
oldval := Pref::heavisideAtOrigin(1)
```

$$\frac{1}{2}$$

Check the new value of `heaviside(0)`. Find the Z-transform of `heaviside(x)` for this value.

```
heaviside(0);
ztrans(heaviside(x),x,z)
```

$$1$$

$$\frac{1}{z-1}+1$$

The output of `ztrans` is affected by the new value of `heaviside(0)`.

Restore the old value of `heavisideAtOrigin` using `oldval`.

```
Pref::heavisideAtOrigin(oldval):
```

Alternatively, restore the default value of `heavisideAtOrigin` by specifying the input as `NIL`.

```
Pref::heavisideAtOrigin(NIL):
```

Parameters

`val`

Arithmetical expression

Return Values

The previously set value of `val`.

See Also

MuPAD Functions
`heaviside` | `Pref::fourierParameters`

Pref::ignoreNoDebug

Controls debugging of procedures

Syntax

```
Pref::ignoreNoDebug(TRUE)
```

```
Pref::ignoreNoDebug(FALSE)
```

```
Pref::ignoreNoDebug(NIL)
```

```
Pref::ignoreNoDebug()
```

Description

`Pref::ignoreNoDebug(TRUE)` allows debugging of `procedures` even if they have the option `noDebug` set.

`Pref::ignoreNoDebug()` returns the current value.

`Pref::ignoreNoDebug(NIL)` resets the default value, which is `FALSE`.

`Pref::ignoreNoDebug(FALSE)` resets the default value, which is `FALSE`.

Return Values

Previously set value

See Also

MuPAD Domains
`DOM_PROC`

MuPAD Functions
`debug`

Pref::keepOrder

Order of terms in sum outputs

Syntax

```
Pref::keepOrder(<Always>)
Pref::keepOrder(<DomainsOnly>)
Pref::keepOrder(<System>)
Pref::keepOrder(NIL)
Pref::keepOrder()
```

Description

`Pref::keepOrder` influences the output order of terms in sums.

Usually, the output system uses its own ordering of the terms in a `sum` to optimize the appearance of the output. This order may be different from the internal ordering of the sum. The output system prefers to re-order the terms such that the first term is positive.

Sometimes it is desirable to see the terms of a sum in the internal order. This can be achieved with `Pref::keepOrder(Always)`.

By default, the term order of polynomials and domain elements is left unchanged.

`Pref::keepOrder(NIL)` restores the default state, which is `DomainsOnly`.

`Pref::keepOrder()` returns the currently set value.

Examples

Example 1

Here we create a domain element `e`, an expression `f`, and a polynomial `p` containing sums. With the default setting `DomainsOnly`, only the output of the expression `f` is not in the internal order:

```
d := newDomain("d"):  d::print := x -> extop(x):
e := new(d, b - a):  f := b - a:  p := poly(1 - x):
e, f, p
```

$$-a+b,\ b-a,\ \mathrm{poly}(-x+1,\ [x])$$

With the setting `Always`, `e`, `f`, and `p` are all printed in the internal order:

```
Pref::keepOrder(Always):
e, f, p
```

$$-a+b,\ -a+b,\ \mathrm{poly}(-x+1,\ [x])$$

With the setting `System`, the output order differs from the internal ordering for `e`, `f`, and `p`:

```
Pref::keepOrder(System):
e, f, p
```

$$b-a,\ b-a,\ \mathrm{poly}(1-x,\ [x])$$

`Pref::keepOrder(NIL)` restores the default state; `Pref::keepOrder()` returns the current setting:

```
Pref::keepOrder(NIL):  Pref::keepOrder()
```

DomainsOnly

Options

`Always`

The output always corresponds to the internal order.

`DomainsOnly`

In polynomials and domain elements, the ordering of terms corresponds to the internal order. Other sums may be re-ordered by the output system.

This is the default setting of `Pref::keepOrder`.

`System`

The output order of terms in sums is determined by the output system and does not necessarily correspond to the internal order.

Return Values

Previously defined value: `Always`, `DomainsOnly`, or `System`.

See Also

MuPAD Domains
`Dom::MultivariatePolynomial` | `Dom::Polynomial` | `Dom::UnivariatePolynomial` | `DOM_POLY`

MuPAD Functions
`print`

Pref::kernel

Version number of the presently used kernel

Syntax

```
Pref::kernel()
```

```
Pref::kernel(<BitsInLong>)
```

```
Pref::kernel(<BuildNr>)
```

Description

The version numbers of the kernel and the library may differ. `Pref::kernel` refers to the kernel, whereas the call `version()` returns the version number of the installed MuPAD library.

Examples

Example 1

Here the version numbers of kernel and library differ:

```
Pref::kernel() = version()
```

$$[6, 2, 0] = [6, 2, 0]$$

Example 2

A 32-bit architecture:

```
Pref::kernel(BitsInLong)
```

$$32$$

Example 3

At the time of this writing, kernels build number was 42703:

```
Pref::kernel(BuildNr)
```

```
42703
```

Options

`BitsInLong`

`Pref::kernel(BitsInLong)` returns the number of bits of a long integer number. On a 64-bit architecture it returns 64, otherwise 32.

`BuildNr`

The kernel has an additional build number which enables the developers to identify the exact sources for this kernel.

Return Values

Version number: a list of three nonnegative integers or a number.

See Also

MuPAD Functions

`buildnumber` | `version`

Pref::maxMem

Set a memory limit for the session

Syntax

```
Pref::maxMem(kbytes)
```

```
Pref::maxMem(NIL)
```

```
Pref::maxMem()
```

Description

`Pref::maxMem(kbytes)` with `kbytes` greater than 0 sets a limit for the physically allocated memory of the current MuPAD session. A computation exceeding this memory limit raises an error.

The physically allocated memory is the second of the values returned by `bytes()`.

Note: The memory limit is "soft" because the memory is checked only occasionally. Usually, more memory is actually used before the excess is detected. Cf. "Example 1" on page 26-35.

The call `Pref::maxMem()` returns the current value of the memory limit without changing it.

The call `Pref::maxMem(NIL)` switches off the memory watch dog.

Examples

Example 1

No computation should increase the memory usage of the current MuPAD session to more than a total of 10 megabytes:

```
Pref::maxMem(10 * unit::MByte):
```

The following loop creates larger and larger matrices until the memory limit is exceeded. Note that the current physical memory allocation returned by `bytes()[2]` is measured in bytes:

```
for n from 100 to 150 step 5 do
   A := linalg::vandermonde([x.j $ j=1..n]);
   print(n, ceil(bytes()[2]/1024)*unit::kByte);
end_for:
```

```
     100, 7311 kByte

     105, 9311 kByte

     110, 9648 kByte

     115, 10113 kByte

 Error: Out of
memory [watchdog-memory];   Evaluating: linalg::vandermonde


Error: Out of memory. [watchdog-memory]
  Evaluating: linalg::vandermonde
```

Note that the memory limit was exceeded when computing the 115 ×115 Vandermonde matrix. However, because the memory consumption is measured only occasionally, this matrix was generated successfully without an error. Only in the next step, the memory watchdog recognizes excessive memory usage and aborts the computation of the 120×120 Vandermonde matrix.

```
Pref::maxMem(NIL):
delete A:
```

Parameters

`kbytes`

The memory limit in kBytes: a nonnegative integer or an expression using `unit::Byte`, `unit::kByte`, `unit::MByte`, or `unit::GByte`.

Return Values

Previously defined memory limit: 0 or an expression involving `unit::MByte`.

See Also

MuPAD Functions
`bytes` | `MAXDEPTH` | `Pref::maxTime`

Pref::maxTime

Time limit for computations

Syntax

```
Pref::maxTime(seconds)
```

```
Pref::maxTime(NIL)
```

```
Pref::maxTime()
```

Description

`Pref::maxTime(seconds)` with `seconds` greater than 0 sets a time limit for all following MuPAD instructions. Each computation not finished within the given time raises an error.

The call `Pref::maxTime()` returns the current value of the time limit without changing it.

The call `Pref::maxTime(NIL)` switches off the timer watch dog.

Examples

Example 1

No computation should take more than 10 seconds:

```
Pref::maxTime(10 * unit::sec):
```

Note that `time` returns the CPU time in milliseconds. The following `while` loop is designed to run longer than 10 seconds:

```
TIME:= time():
while time() - TIME < 20000 do null() end_while
```

```
Error: Execution time is exceeded. [watchdog-time]

Pref::maxTime(NIL): delete TIME:
```

Parameters

seconds

The time limit in seconds: a nonnegative integer or an expression involving time units.

Return Values

Previously defined time limit: 0 or an expression involving `unit::sec`.

See Also

MuPAD Functions

`Pref::maxMem` | `rtime` | `time` | `traperror`

Pref::output

Modify the screen output of objects

Syntax

```
Pref::output(f)
```

```
Pref::output(NIL)
```

Description

`Pref::output` allows to modify the screen output of objects returned by the MuPAD kernel.

When the MuPAD kernel returns a result `x`, say, of a computation, the function `f` is called before the result is printed to the screen. Instead of `x`, the return value `f(x)` is used as screen output of the computation.

Make sure that a user-defined output function `f` processes *arbitrary MuPAD objects*.

The call `Pref::output(NIL)` resets the output function to the identity map: the screen output coincides with the object returned by the computation. `NIL` is the default value of the output function.

Examples

Example 1

All numbers of type `Type::Numeric` shall be displayed as floating point numbers. Since the kernel may return sequences of objects, the output function may be called with an unknown numbers of parameters. It uses `map` to apply its functionality to all of its arguments. Whenever a numerical object of type `Type::Numeric` is encountered, it is replaced by a floating-point approximation:

```
f := proc(x)
begin
  if args(0) > 1 then
```

```
      return(map(args(), f))
  end_if;
  if testtype(x, Type::Numeric) then
      return(float(x))
  else return(x)
  end_if;
end_proc:
Pref::output(f):

4/9; sin(3); 4/9, sin(3), 1/2 + 17*I
```

$$0.4444444444$$

$$\sin(3)$$

$$0.4444444444,\ \sin(3),\ 0.5 + 17.0\,i$$

We restore the standard mode:

```
Pref::output(NIL): delete f:
```

Example 2

The procedure `generate::TeX` is applied to the result of a computation. The corresponding TeX code (a string) is displayed:

```
Pref::output(generate::TeX):
sqrt(x^2 - 1/x)
```

```
"\sqrt{x^2 - \frac{1}{x}}"
```

We restore the standard mode:

```
Pref::output(NIL):
```

Parameters

`f`

The “output function”: a procedure

Return Values

Previously defined “output function”, or `NIL`.

See Also

MuPAD Functions

`Pref::keepOrder` | `Pref::postInput` | `Pref::postOutput`

Pref::outputDigits

Set the number of digits in floating-point outputs

Syntax

```
Pref::outputDigits(n)

Pref::outputDigits(<UseDigits>)

Pref::outputDigits(<InternalPrecision>)

Pref::outputDigits()
```

Description

`Pref::outputDigits(n)` sets the number of digits in outputs of floating-point numbers to an integer `n`. This command does not set the precision for calculations. See “Example 1” on page 26-44.

`Pref::outputDigits(InternalPrecision)` sets the number of digits in floating-point outputs to settings MuPAD used when creating these floating-point numbers. If you use `Pref::outputDigits(InternalPrecision)`, the lengths of floating-point numbers in the same output region can differ because the numbers were created with different precision. See “Example 2” on page 26-44.

`Pref::outputDigits(UseDigits)` restores the setting to the number of digits previously set by `DIGITS`. The default value for `DIGITS` is 10. Suppose, you use internal precision for displaying numbers or explicitly specify a number of digits in outputs. If you want to switch back to the number of digits specified by `DIGITS`, use `Pref::outputDigits(UseDigits)`. See “Example 3” on page 26-45.

`Pref::outputDigits()` returns the current setting for the number of digits in outputs of floating-point numbers. This command does not change the setting.

Examples

Example 1

Display the floating-point approximation of $\frac{1}{3}$ using 40 digits without changing the precision for calculations. The default number of digits that MuPAD uses for calculations with floating-point numbers is 10. MuPAD can increase the precision of calculations by several digits (guard digits). Round-off errors cause the last digits in the following result to be incorrect:

```
Pref::outputDigits(40):
float(1/3)
```

```
0.33333333333333333304421275400386548426 47
```

Note: Always explicitly set the precision you need. Do not rely on having guard digits.

To set the precision for calculations with floating-point numbers, use `DIGITS`. Display the floating-point approximation of $\frac{1}{3}$ using 40 and 15 digits:

```
DIGITS := 40:
float(1/3);
Pref::outputDigits(15):
float(1/3)
```

```
0.3333333333333333333333333333333333333 3
```

```
0.333333333333333
```

Example 2

Approximate the ratio $\frac{1}{6}$ with different precisions:

```
DIGITS := 15:
```

```
x := float(1/6):
DIGITS := 3:
y := float(1/6):
DIGITS := 40:
z := float(1/6):
```

Display the floating-point approximation of $\frac{1}{6}$ using the number of digits that MuPAD uses internally. The lengths of floating-point numbers in the same output region can differ because the numbers were created with different precision:

```
Pref::outputDigits(InternalPrecision):
x;
y;
z
```

$$0.166666666666667$$

$$0.167$$

$$0.1666666666666666666666666666666666666667$$

Example 3

To switch back to the number of digits set previously by `DIGITS`, use `Pref::outputDigits(UseDigits)`:

```
Pref::outputDigits(UseDigits):
x;
y;
z
```

$$0.1666666666666666666666677695741500707036$$

$$0.1666666666678793262690305709838867187 5$$

$$0.1666666666666666666666666666666666666667$$

Parameters

n

An integer

Return Values

Previously set value

See Also

MuPAD Functions
`DIGITS` | `float` | `Pref::floatFormat`

Pref::postInput

Actions after input

Syntax

```
Pref::postInput(f)
```

```
Pref::postInput(NIL)
```

```
Pref::postInput()
```

Description

`Pref::postInput` allows to set user actions directly after input.

After entering any MuPAD command `x`, say, and sending this command to the kernel, `f(x)` is called before the kernel starts to process the input. This happens for any input until the post-input is switched off via the call `Pref::postInput(NIL)`.

The function `f` implicitly uses the `option hold`, i.e., `f` sees the input command as entered and parsed without any evaluation.

`f` cannot change the input command that is sent to the kernel for evaluation. However, `f` can store the input in some global variable for later processing, or some other actions can be performed.

`Pref::postInput()` returns the current value of the post-input function or `NIL`, respectively.

`Pref::postInput`, possibly in conjunction with `Pref::postOutput`, is useful for initializing variables to compute status information such as the execution time for the command that is to be executed. See “Example 2” on page 26-49.

Examples

Example 1

The post-input function sees the input as entered, i.e., before evaluation by the kernel:

```
Pref::postInput(proc() begin
  print(Unquoted, "input" = args())
end):
```

```
1 + 2
```

```
input = 1 + 2
```

$$3$$

```
1 + 2, x = sin(0.1)
```

```
input = (1 + 2, x = sin(0.1))
```

$$3, x = 0.09983341665$$

```
x := 1234; y := 5678
```

```
input = (x := 1234)
```

$$1234$$

```
input = (y := 5678)
```

$$5678$$

Post-input is switched off. This command calls the post-input function for the very last time:

```
Pref::postInput(NIL):
```

```
input = Pref::postInput(NIL)
```

```
delete x, y:
```

Example 2

For any command, the run time is to be computed and displayed. The function declared in `Pref::postInput` sets a global timer value `TIME` after each input. After the output of the result, the function declared in `Pref::postOutput` compares the current time and the starting time `TIME`.

```
Pref::postInput(() -> (TIME:= time())):
Pref::postOutput(() -> "Time: ".expr2text((time() - TIME)*msec)):

int(cos(x)*exp(sin(x)), x)
```

$$e^{\sin(x)}$$

```
Time: 40 msec
```

$$e^{\sin(x)}$$

```
Time: 40 msec

Pref::postInput(NIL): Pref::postOutput(NIL):
delete TIME:
```

Example 3

As another example of using `Pref::postInput` for storing information to influence the output, we combine it with `Pref::output` to include (a rendered version of) the input and the result:

```
Pref::postInput(() -> (LASTINPUT := args())):
Pref::output(() -> val(LASTINPUT) = args()):
```

This makes MuPAD write “input = result” to the screen, while leaving the history (accessible by %) intact:

```
int(x, x);
sum((-1)^i/(2*i+1), i=0..infinity);
sin(%)
```

$$\int x \, dx = \frac{x^2}{2}$$

$$\sum_{i=0}^{\infty} \frac{(-1)^i}{2\,i+1} = \frac{\pi}{4}$$

$$\sin(\%) = \frac{\sqrt{2}}{2}$$

Parameters

`f`

The function to be executed after input: a procedure. The default value of this function is `NIL` (no post-input).

Return Values

Previously set post-input function.

See Also

MuPAD Functions
`Pref::postOutput`

Pref::postOutput

Actions after output

Syntax

```
Pref::postOutput(f)
```

```
Pref::postOutput(NIL)
```

```
Pref::postOutput()
```

Description

After the result `x`, say, of a MuPAD command is printed on the screen, `f(x)` is called and executed before the next prompt for user input appears. This happens for any output until the post-output is switched off via `Pref::postOutput(NIL)`.

After the usual output of the result `x` of a MuPAD command, the return value of `f(x)` is printed on the screen with `PRETTYPRINT` = `FALSE`. However, `f(x)` does not return any value to the MuPAD session. It cannot be accessed via `last`.

`Pref::postOutput()` returns the current value of the post-output function or `NIL`, respectively.

`Pref::postOutput`, possibly in conjunction with `Pref::postInput`, can be used to produce status information after each output. One may think of timer informations, memory usage, result types etc.

Examples

Example 1

Here, `Pref::postOutput` is used to enumerate the output line and display the type of the result. It uses the global variable `LineNumber` which must be initialized before any output is produced. The definition of the post-output operation as well as the

initialization of the global variable can be done in the file "`userinit.mu`" which is read automatically during start-up.

```
Pref::postOutput(
  proc()
  begin
    LineNumber:= LineNumber + 1;
    "Out[" . expr2text(LineNumber). "]: ".
    "type = ".expr2text(op(map([args()], domtype)));
  end_proc):
LineNumber:= 0:

int(x^5*exp(-x), x)
```

$$-e^{-x}\,(x^5+5\,x^4+20\,x^3+60\,x^2+120\,x+120)$$

```
Out[1]: type = DOM_EXPR

int(x^5*exp(-x), x = 0..infinity),
numeric::int(x^5*exp(-x), x = 0..infinity)
```

120, 120.0

```
Out[2]: type = DOM_INT, DOM_FLOAT
```

The following `print` command returns the void object `null()` to the MuPAD session. The output of `null()` is suppressed:

```
print("print returns the void object")
```

"print returns the void object"

The following command is terminated by a semicolon to suppress the output. Consequently, no post-output is created, either.

```
x := sin(2):
```

Post-output is switched off:

```
Pref::postOutput(NIL): delete LineNumber, x:
```

Example 2

For any command, the run time is to be computed and displayed. The function declared in `Pref::postInput` sets a global timer value `TIME` after each input. After the output of the result, the function declared in `Pref::postInput` compares the current time and the starting time `TIME`. The current `TEXTWIDTH` is used to prepend some suitable whitespace via `stringlib::format` to flush right the timer information:

```
Pref::postInput(() -> (TIME:= time())):
Pref::postOutput(
  proc() begin
   stringlib::format("Time: ".expr2text(time() - TIME)." msec",
                     TEXTWIDTH-1, Right)
  end_proc):

int(x^10*exp(-x), x)
```

$$-e^{-x}\left(x^{10}+10\,x^{9}+90\,x^{8}+720\,x^{7}+5040\,x^{6}+30240\,x^{5}+151200\,x^{4}+604800\,x^{3}+1814400\,x^{2}+3628800\,x+3628800\right)$$

```
                                                Time: 84.005 msec
```

```
Pref::postInput(NIL): Pref::postOutput(NIL):
delete T, TIME:
```

Example 3

The following post-output lists all identifiers with properties in the result of the last MuPAD command. It extracts the indeterminates via `indets` and uses `property::hasprop` to query whether they have properties:

```
Pref::postOutput(
  proc()
  begin
    select(indets({args()}), property::hasprop);
    "identifiers with properties: " . expr2text(op(%))
  end_proc):

assume(0 < a < b): a + b + c
```

$$a + b + c$$

```
identifiers with properties: a, b
```

```
Pref::postOutput(NIL): delete a, b:
```

Parameters

f

The function to be executed after output: a procedure. The default value of this function is `NIL` (no post-output).

Return Values

Previously set post-output function.

See Also

MuPAD Functions

`Pref::postInput`

Pref::report

Informations during evaluation

Syntax

```
Pref::report(level)
```

Description

`Pref::report` controls the frequency of report messages of the MuPAD kernel during evaluation.

A kernel function displays frequently the three informations *memory used*, *memory reserved* and *evaluation time in seconds*.

The level `0` disables printing information. If `level` is `1`, about every hour a message will be printed. With `9` as argument the most reports will be printed. The frequency depends on the machine's speed.

A call of `Pref::report` without arguments returns the current value. The argument `NIL` resets the default value `0`.

Examples

Example 1

Frequently information:

```
Pref::report(9):
limit((1+1/n)^n,n=infinity)
```

```
[used=1612k,
reserved=1738k, seconds=1] [used=2716k, reserved=2856k, seconds=2]
                                exp(1)
```

Reset to no information:

```
Pref::report(0):
```

Parameters

`level`

An integer between `0` and `9`, or `NIL`

Return Values

Last defined level

See Also

MuPAD Functions
`Pref::callBack`

Pref::trailingZeroes

Trailing zeroes when printing floating-point numbers

Syntax

```
Pref::trailingZeroes(value)
```

```
Pref::trailingZeroes()
```

Description

`Pref::trailingZeroes` determines, whether trailing zeroes will be appended, when floating-point numbers are printed.

If enabled (with argument `TRUE`), after the significant numbers of a floating-point number (behind the point) zeroes will be appended until the number of digits reaches the value of `DIGITS`.

A call of `Pref::trailingZeroes` without arguments will return the current value. The argument `NIL` will reset the default value, which is `FALSE`.

Examples

Example 1

By default, trailing zeroes will not be displayed:

```
DIGITS:= 10:
1.4
```

```
      1.4
```

Display of trailing zeroes will be enabled:

```
Pref::trailingZeroes(TRUE):
```

```
1.4
```

```
1.400000000
```

The default mode is restored:

```
Pref::trailingZeroes(NIL):
```

Parameters

`value`

TRUE, FALSE or NIL

Return Values

Last defined value.

See Also

MuPAD Functions
DIGITS | Pref::floatFormat | print

Pref::typeCheck

Type checking of formal parameters

Syntax

```
Pref::typeCheck(Always | Interactive | None)
```

```
Pref::typeCheck(NIL)
```

```
Pref::typeCheck()
```

Description

`Pref::typeCheck` determines the kind of type checking of procedure parameters.

The definition of a MuPAD procedure may contain formal parameters. There is a syntax to attach a type specification to these parameters. If and when type checking is enabled, the types of actual parameters are checked against the type specifications and an error is raised if a parameter does not meet the specification.

Type specifications are used as the second parameter of `testtype`. The most important ones are "Domain Types" and objects of the domain Type. With Type, user defined types can be easily added to the system to extend the type checking mechanism.

The arguments of `Pref::typeCheck` can be:

- `None`

  No parameters are checked.
- `Interactive`

  Parameters entered interactively are checked. This is the default.
- `Always`

  All formal parameters are checked.

The default value `Interactive` means: When the user is calling a procedure `f`, its parameters will be checked, but none of the procedures called by the user called procedure `f` performs type checking.

A call of `Pref::typeCheck` without arguments returns the current value. The argument `NIL` resets the default value, which is `Interactive`.

Examples

Example 1

We define a procedure `f` expecting an identifier and an integer:

```
f:= proc(a : DOM_IDENT, b : DOM_INT)
    begin
      evalassign(a, b, 1)
    end_proc:
f(a, 2)
```

```
    2
```

Now `a` has the value `2`, but an identifier is expected:

```
f(a, 2)
```

```
Error: The object '2' is incorrect. The type of argument number 1 must be 'DOM_IDENT'.
  Evaluating: f
```

```
delete a:
```

Options

`Always`

Parameter types are checked for every call.

`Interactive`

Parameter types are checked for interactive calls, not for “inner” calls during the computation.

None

No parameter type checks are performed by the MuPAD kernel. Explicit checks inside procedures still use the usual `testargs` mechanism.

Return Values

Previously defined value

Algorithms

The syntax to test parameters directly (without a test in the procedure body) is the formal parameter followed by a colon and then the type object: `proc(a : DOM_IDENT, b : Type::Integer)`. This means: `a` must be of the type `DOM_IDENT` and `b` must be of the type `Type::Integer`.

Note that you cannot use automatic type checking for arguments that are used for overloading inside the procedure.

The objects of the Type library are usually more general than the MuPAD kernel types.

See Also

MuPAD Domains
`DOM_PROC`

MuPAD Functions
`args | domtype | hastype | proc | testargs | testtype | type`

Pref::userOptions

Additional options when starting MuPAD

Syntax

```
Pref::userOptions()
```

Description

`Pref::userOptions()` returns additional options, given by the user when calling MuPAD.

When starting the MuPAD kernel with the flag `"-U"` the user can define options that can be used in the MuPAD session.

Examples

Example 1

If you start MuPAD with the command `mupad -U "Hello World"`, `Pref::userOptions` returns the string `"Hello World"`:

```
Pref::userOptions()
```

```
"Hello World"
```

Example 2

To define several user options one can use a separator between the strings. MuPAD is called with `mupad -U "myhome,/home/user/myhome,2"`:

```
Pref::userOptions()
```

```
"myhome,/home/user/myhome,2"
```

The following call splits the string into the three parts (to demonstrate, the string is written explicitely):

```
s := "myhome,/home/user/myhome,2":
opts := []:
ind := 0:
while (ind := stringlib::contains(s, ",", Index)) <> FALSE do
  opts := opts . [s[1 .. ind-1]];
  s := s[ind+1 .. length(s)]:
end_while:
opts := opts . [s]
```

```
["myhome", "/home/user/myhome", "2"]
```

```
delete s, opts, ind:
```

Return Values

User defined options as strings

Pref::verboseRead

Shows reading of files

Syntax

```
Pref::verboseRead(value)
```

```
Pref::verboseRead()
```

Description

With `Pref::verboseRead` the reading of library packages and files can be shown.

The arguments of `Pref::verboseRead` represent:

- `0:`

  no messages when reading files (default).
- `1:`

  message if a library packages will be read.
- `2:`

  messages if a package or any library function will be read.
- `NIL:`

  restore the default value `0`.

A call of `Pref::verboseRead` without arguments returns the current value.

Examples

Example 1

Show the reading of library packages:

```
reset():
Pref::verboseRead(1):
sin(x)

loading package 'Type' [mupad/share/lib/lib.tar#lib/]
                            0.8414709848
```

Show reading of all library files:

```
reset():
Pref::verboseRead(2):
sin(1.0)

reading file mupad/share/lib/lib.tar#lib/SPECFUNC/sin.mu
reading file mupad/share/lib/lib.tar#lib/SPECFUNC/sinh.mu reading
file mupad/share/lib/lib.tar#lib/STDLIB/infinity.mu loading package
'Type' [mupad/share/lib/lib.tar#lib/] reading file mupad/share/lib/lib.tar#lib/TYPE/Ar
                            0.8414709848
```

The default mode is restored:

```
Pref::verboseRead(NIL):
```

Parameters

`value`

0, 1, 2, or `NIL`

Return Values

Last defined value

See Also

MuPAD Functions

`fread` | `prog::trace` | `read`

Pref::warnDeadProcEnv

Warnings about wrong usage of lexical scope

Syntax

```
Pref::warnDeadProcEnv(TRUE)

Pref::warnDeadProcEnv(FALSE)

Pref::warnDeadProcEnv(NIL)

Pref::warnDeadProcEnv()
```

Description

`Pref::warnDeadProcEnv()` returns the current setting.

`Pref::warnDeadProcEnv(TRUE)` switches on warnings about unreachable procedure environments.

`Pref::warnDeadProcEnv(FALSE)` switches warning messages off.

`Pref::warnDeadProcEnv(NIL)` will reset the default value, which is `FALSE`.

If a procedure is executed a *procedure environment* is created for this procedure. It contains the current values of formal parameters and local variables. On exit of the procedure this procedure environment is normally not needed any more and destroyed.

If a procedure returns a local procedure as its result, this local *procedure escapes its scope*. Usually this is no problem. Only if the escaping procedure contains references to formal parameters or local variables of the outer procedure these *variables escape their scope*. These variables can not be dereferenced since they reference values of a procedure environment of the outer procedure which does not exist any more.

Use option *escape* in the outer procedure in order to keep its procedure environment untouched.

Environment Interactions

Allows or suppresses warning messages.

Examples

Example 1

Here we write procedure `p` which returns a local procedure. The returned procedure adds the value of its argument `y` to the value of the argument `x` of the first procedure. The following naive implementation produces a strange output and, when the resulting procedure is called, a warning message and an error:

```
Pref::warnDeadProcEnv(FALSE):
p := proc(x) begin y -> x + y end:
f := p(1);  f(2)
```

$$y \to \mathrm{DOM\_VAR}(1, 2) + y$$

```
 Warning:
Uninitialized variable 'unknown' used.   Evaluating: f

 Error: Illegal operand. [_plus]   Evaluating:
f
```

If `Pref::warnDeadProcEnv` is set to `TRUE` MuPAD will print a warning message when the local procedure escapes its scope:

```
Pref::warnDeadProcEnv(TRUE):
p := proc(x) begin y -> x + y end:
f := p(1)

 Warning:
Found dead closure of procedure 'p'.
```

$$y \to \mathrm{DOM\_VAR}(1, 2) + y$$

Use option *escape* in the outer procedure to prevent this warning. The returned procedure `f` will then work as expected:

```
p := proc(x) option escape; begin y -> x + y end:
f := p(1);  f(2)
```

$$y \to x + y$$

$$3$$

Return Values

Previously set value

See Also

MuPAD Functions

`proc`

prog – Programmer's Toolbox

prog::check
prog::exprlist
prog::exprtree
prog::find
prog::getname
prog::getOptions
prog::init
prog::isGlobal
prog::ntime
prog::profile
prog::remember
prog::sort
prog::tcov
prog::test
prog::testexit
prog::testinit
prog::trace
prog::traced
prog::untrace
prog::wait

prog::check

Checking objects

Syntax

```
prog::check(object, <infolevel>, options)
```

Description

The call `prog::check(object)` checks the MuPAD object `object`. `object` may be a procedure, a function environment, or a domain. One may also give a list of such objects.

If `All` is given as first parameter, all defined procedures, function environments and domains are checked (see `anames`).

`infolevel` determines the amount of information given while checking. The following values are useful:

- 1

  summarizing number of warnings per checked object, if at least one warning occurs (default)
- 2

  as 1, but a short message is printed even if no warning was produced
- 3

  summary of warnings per checked object
- 5

  displays each checked object, followed by individual warnings, followed by a summary and the number of warnings, if any.
- 10 ... 15

  additional outputs (for debugging/information)

`options` can be any of the described options.

With option `All`, all are checked. Without options, the set `{Domain, Global, Interface, Level, Local, Protect, Save}` is used.

Note: The arguments of `hold` expressions are not checked.

Examples

Example 1

The following function contains a number of mistakes, some of which were actually legal in previous versions of MuPAD.

Lines `1` and `2` contains declarations of local variables. In line `4` an undeclared (global) variable `g` is used. Line `7` applies `level` to a local variable (the call simply returns the value of `X` in MuPAD 2.0). Line `10` contains an assignment to a formal parameter. This parameter will be overwritten and its old value lost:

```
f:= proc(X, Y)                  //  1  Local
      local a, b;               //  2  Local
    begin                       //  3
      g:= proc(X)               //  4  Global
            option hold;        //  5
          begin                 //  6
            a:= level(X, 2);    //  7  Level
            a:= a + X           //  8
          end_proc;             //  9
      Y:= g(Y);                 // 10  Assign, Global
    end_proc:
prog::check(f, 3)
```

```
Critical usage of 'level' on local variable ' [f]

Function 'level' applied to variables: {X} in [f, proc in 'f']

Global idents: {g} in [f]

Unused local variables: {b} in [f]

Warnings: 3 [f]
```

Only search for global variables, but give more messages:

```
prog::check(f, 5, Global)
Checking f (DOM_PROC)
  Global variable 'g' in [f]
  Global variable 'g' in [f]
  Global idents: {g} in [f]
  Warnings: 1 [f]
```

Now check everything:

```
prog::check(f, 5, All)
Checking f (DOM_PROC)
  Global variable 'g' in [f]
  Critical usage of 'level' on local variable ' [f]
  Function 'level' applied to variables: {X} in [f, proc in 'f']
  Procedure environment of [f] used by [f, proc in 'f']
  Assignment to formal parameter 'Y' in [f].
  Global variable 'g' in [f]
  Global idents: {g} in [f]
  Unused local variables: {b} in [f]
  Unused formal parameters: {X} in [f]
  Assignments to formal parameters: {Y} in [f]
  Warnings: 8 [f]
```

Global variables declared with the option “save” are allowed:

```
f:= proc(X)                    //  1  Local
      save g;                  //  2  Save
    begin                      //  3
      g:= X
    end_proc:
prog::check(f, 2, Global, Save)
```

```
Warnings: 0 [f]
```

Parameters

`object`

Procedure, function environment or domain to check, the identifier `All`, or a list of objects

`infolevel`

Positive integer that determines the completeness of messages

Options

`All`

Enables all known options

`Global`

Report unknown global identifiers

`Local`

Report unused local variables

These are variables that were declared by `local`, but never used in the procedure.

`Localf`

Report unused local variables *and* unused formal parameters

The same as `Local`, but the same check is additionally performed for formal parameters of a procedure. Those are the argument names as given in the definition of the procedure.

`Assign`

Report assignments to formal parameters of procedures

Because a formal parameter will be overwritten, those assignments *could* indicate a programming error (however, not imperative).

`Level`

The application of `level` to local variables is reported. Starting with MuPAD 2.0, local variables are simply replaced by their values on evaluation and calling `level` on them does not have any effect.

`Domain`

Report undefined entries of domains (uses the slot `"undefinedEntries"`)

`Interface`

Information about undefined entries of a domain interface are printed, i.e., entries in the domain interface, that are not defined as entries of the domain.

`Environment`

Information about assignments to environment variables of MuPAD are printed. These assignments could change the global behavior of MuPAD if the change is not undone (preferably using `save`, to catch error conditions).

`Protect`

Information about assignments to protected variables of MuPAD are printed.

`Save`

A message about a global identifier is suppressed, when the checked object is a procedure and the identifier is saved with option “save”.

`Special`

Information about some special cases are printed. Currently, the only implemented special case is assignments to `HISTORY`.

`Escape`

`prog::check` prints warnings about procedures which may require the option `escape`.

Return Values

`prog::check` returns the void object `null()`. Output messages are printed on the screen.

See Also

MuPAD Functions

`debug` | `prog::getname` | `prog::init` | `prog::isGlobal` | `prog::trace`

prog::exprlist

Convert an expression into a nested list

Syntax

```
prog::exprlist(ex)
```

Description

`prog::exprlist` returns a list that contains all operands of the expression `ex`. Each operand of type `DOM_EXPR` is converted into a list, too.

The return value of `prog::exprlist` can be used directly as argument for `adt::Tree` resp. `output::tree`.

Examples

Example 1

The example shows the nested list for the expression `a + b*2 - d*(a + c)`:

```
prog::exprlist(a + b*2 - d*(a + c))
```

$$[\_plus, a, [\_mult, b, 2], [\_mult, d, [\_plus, a, c], -1]]$$

The return value can be used to create and display a tree:

```
expose(adt::Tree(prog::exprlist(a + b*2 - d*(a + c))))
```

```
_plus
|
+-- a
|
+-- _mult
```

```
|    |
|    +-- b
|    |
|    `-- 2
|
`-- _mult
     |
     +-- d
     |
     +-- _plus
     |    |
     |    +-- a
     |    |
     |    `-- c
     |
     `-- -1
```

Parameters

ex

Expression to convert

Return Values

List

See Also

MuPAD Functions

`adt::Tree` | `output::tree` | `prog::exprtree`

prog::exprtree

Visualize an expression as tree

Syntax

```
prog::exprtree(ex, <Quiet>)
```

Description

`prog::exprtree(ex)` visualizes any MuPAD expression `ex` as tree.

Every expression in MuPAD is internally a tree. The operations are the nodes, and the operands are the leafs.

Examples

Example 1

The example shows the structure of the expression `a + b*2 - d*(a + c)`:

```
prog::exprtree(a + b*2 - d*(a + c))
```

```
_plus
|
+-- a
|
+-- _mult
|   |
|   +-- b
|   |
|   `-- 2
|
`-- _mult
    |
    +-- d
```

```
|
+-- _plus
|    |
|    +-- a
|    |
|    `-- c
|
`-- -1
```

Tree1

`Tree1` is the return value of type `adt::Tree`. This object can be exposed or taken for other operations.

The option `Quiet` suppresses the output, only the tree is returned:

```
prog::exprtree(a + b*2 - d*(a + c), Quiet)
```

Tree2

Parameters

`ex`

Expression to visualize

Options

`Quiet`

With this option no output will be printed on screen. The return value of type `adt::Tree` represents the tree structure of `ex`.

Return Values

Object of type `adt::Tree`

See Also

MuPAD Functions
`adt::Tree`

prog::find

Find operands of objects

Syntax

```
prog::find(obj, piece, <Depth = d>, <Type>)
```

Description

`prog::find(obj, piece)` returns the position of the object `piece` in the MuPAD object `obj` as list. The list represents a "path" to the given object. With this list and the functions `op` and `subsop`, the object can directly be accessed.

A path to an object `piece` is a list that contains integers `i1, ..., in`.

The meaning is that `piece` is the `in`-th operand of the `(in - 1)`-st operand etc. of the `i1`-st operand of the given object `obj`.

Stated differently, `op(ex, [i1, ..., in]) = opr`.

If the searched object is containing several times, a sequence of lists is returned.

An empty list `[ ]` as path determines the object `obj` itself.

Examples

Example 1

The identifier `a` is the first operand of the expression:

```
prog::find(a + b + c, a)
```

```
[1]
```

The number `1` occurs several times:

```
prog::find(f(1, 1, 1), 1)
```

$$[1], [2], [3]$$

Example 2

The identifier `a` is the first operand of the second operand of the first operand of the expression:

```
prog::find(b*(a - 1) + b*(x - 1), a)
```

$$[1, 2, 1]$$

The result of `prog::find` can be used to access the element with `op` or replace it with `subsop`:

```
op(b*(a - 1) + b*(x - 1), [1, 2, 1]);
subsop(b*(a - 1) + b*(x - 1), [1, 2, 1] = A)
```

$$a$$

$$b\,(A-1)+b\,(x-1)$$

Example 3

How many calls of `return` does `sin` contain?

```
nops([prog::find(sin, return)])
```

$$55$$

`sin` contains many `return` calls! However, `sin` is a function environment and the slots are examined, too. To examine only the main procedure, take the first operand of the function environment:

```
nops(prog::find(op(sin, 1), return))
```

$$23$$

Example 4

`prog::find` can also find all objects of a given type:

```
nops(prog::find(sin, DOM_PROC, Type))
```

```
     64
```

To find only the top level procedures, option `Depth` can be used:

```
nops(prog::find(sin, DOM_PROC, Type, Depth = 1))
```

```
     14
```

Example 5

`prog::find` works with tables and other containes, too:

```
T := table(1 = sin(x), 2 = cos(x), 3 = tan(x),
           4 = tan(y), 5 = sin(y), 6 = cos(y)):
prog::find(T, sin)
```

```
     [1, 2, 0], [5, 2, 0]
```

```
prog::find(T, "cos", Type)
```

```
     [2, 2], [6, 2]
```

Example 6

In this example we show how to manipulate an existing function by substitution. We use `subsop` for the substitution and `prog::find` to get the path for the substitution. Here we replace the `^`-function by the function `mypower` which additionally counts the number of its calls:

```
f := x -> `+`(x^j $ j = 1 .. random(10)()):
mypower := (b, e) -> (count := count + 1;  b^e):
```

```
map([prog::find(f, `^`)],
    X -> (f := subsop(f, X = mypower))):
```

After calling the function `f` ten times in a loop, we see the resulting number of calls of `^` in `count`:

```
count := 0:
for i from 1 to 10 do f(i); end:
count
```

```
     48
```

Parameters

`obj`

Any MuPAD object

`piece`

Any MuPAD object

Options

`Depth`

Option, specified as `Depth = d`

This option allows examining operands of the given object, that are domains, procedures and function environments, only with recursion depth `d`.

Option `Depth` can be used to find procedures, but not locally defined procedures inside the procedures that were found in the first step.

`Type`

When the option `Type` is given, `prog::find` does not search positions `p` in obj such that `op(obj, p) = piece`, but rather those with `testtype(op(obj, p), piece) = TRUE`. Cf. “Example 4” on page 27-15.

Return Values

List of numbers that determine the position of the given object inside of the given expression, or a sequence of lists, if the expression contains the object several times

Algorithms

`prog::find` can be used to examine and manipulate complex MuPAD objects with `subsop`.

See Also

MuPAD Functions
`has` | `op` | `prog::exprtree` | `subsop`

prog::getname

Name of an object

Syntax

```
prog::getname(object)
```

Description

`prog::getname(object)` returns the name of the MuPAD object `object`.

The return value is a string, irrespective of the type of the input.

Names can be extracted from procedures, identifiers, function environments, domains and their methods (and strings, of course). If no name can be extracted from an object, the string `"(noname)"` is returned.

For all other MuPAD objects the result of `expr2text(object)` is returned as name.

Examples

Example 1

My own name:

```
prog::getname(prog::getname)
```

```
"prog::getname"
```

The name of a Domain:

```
prog::getname(Dom::ExpressionField())
```

```
"Dom::ExpressionField()"
```

The “name” of an arbitrary MuPAD object:

```
prog::getname(1)
```

```
"1"
```

```
prog::getname(a + 2*b)
```

```
"a + 2*b"
```

Parameters

`object`

Any MuPAD object

Return Values

Name as string

See Also

MuPAD Functions

`expr2text` | `info` | `op` | `print` | `text2expr`

prog::getOptions

Get and verify options within a procedure

Syntax

```
prog::getOptions(n, arguments, allOptions, <ErrorOnUnexpected>, <optionTypes>)
```

Description

`prog::getOptions` called within a procedure collects and verifies all options from the list of arguments of the calling procedure.

When you write your own procedure, `prog::getOptions` lets you embed the collection and verification of all options of the procedure. When a user calls your procedure, `prog::getOptions` scans all the arguments and provides a data structure that contains all option values. See “Example 1” on page 27-21.

The `prog::getOptions` function returns a list that contains a table of all valid options along with their values and a list of unexpected arguments. For expected options, `prog::getOptions` returns the following values:

If an option can have only `TRUE` or `FALSE` values, a user of your procedure can provide the option name, instead of providing an option-value pair. If a user provides an option without specifying its value, `prog::getOptions` returns the Boolean value `TRUE` for that option. See “Example 2” on page 27-22.

`prog::getOptions` returns a list with two components: a table of expected options and a list of unexpected arguments. You can access the components of that list separately by using the index operator (see `_index`). See “Example 3” on page 27-22.

By default, the `prog::getOptions` function does not error when it finds an unexpected argument. To switch to issuing errors on unexpected arguments, use the parameter ErrorOnUnexpected. See “Example 4” on page 27-23.

The procedure that includes `prog::getOptions` must accept the arguments in the following order: first nonoptional parameters, and then all options. The `prog::getOptions` function lets you specify the number `n` of the argument from which the function starts verifying options. The `prog::getOptions` function assumes that

the first `n - 1` arguments are parameters and, therefore, does not verify them. See “Example 5” on page 27-23.

The parameter optionTypes lets you specify acceptable types for the option values. You can specify acceptable types for some or all of the expected options provided in the allOptions table. See “Example 6” on page 27-24.

The first three parameters of `prog::getOptions` (n, arguments, and allOptions) are required. ErrorOnUnexpected and optionTypes are optional. You must provide the parameters of `prog::getOptions` using the order shown in the Calls section of this page. Therefore, if you want to use the fifth parameter optionTypes, you also must explicitly use the fourth parameter ErrorOnUnexpected.

The second parameter of the `prog::getOptions` function, arguments, is a list of all arguments of your procedure. `prog::getOptions` scans the arguments provided in that list. Although `prog::getOptions` accepts any list as a second argument, the calling procedure always must provide a list of all its arguments (both parameters and options) to avoid potential errors. The syntax `[ args()]` provides a list of all arguments of the calling procedure. See “Example 7” on page 27-25.

Examples

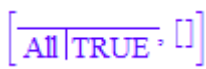
Example 1

To embed the option collection and verification step into your procedure, call `prog::getOptions` function within that procedure. To test the behaviour of `prog::getOptions`, create a function `f` that calls `prog::getOptions` to scan and verify the arguments of `f`. For example, create the function `f` that accepts only one option `All`:

```
f := () -> prog::getOptions(1, [args()], table(All = TRUE)):
```

`prog::getOptions` returns a list. The first entry of the list is a table containing all expected options and their values. The second entry is a list of unexpected arguments. For example the function call `f()` does not contain any unexpected arguments:

```
f();
```



The function call `f(Unexpected)` contains the unexpected argument `Unexpected`:

```
f(Unexpected);
```

```
[ All | TRUE , [Unexpected] ]
```

Example 2

When users call the procedure that includes `prog::getOptions`, they can specify new values for any of the valid options of that procedure. In this case, `prog::getOptions` returns the new values. If a user uses the option without specifying its value, `prog::getOptions` returns the Boolean value `TRUE` for that option. If a user does not explicitly use an expected option, `prog::getOptions` returns the default value of that option provided in the allOptions table. For example, in the function call `f(Proc = op, Warning)`, the `prog::getOptions` function returns the following values:

- If a user provides a value for the option, `prog::getOptions` returns that value.
- If a user does not use the option in a procedure call, `prog::getOptions` returns the default value for that option.

```
f := () -> prog::getOptions(1, [args()],
                               table(All = FALSE,
                                     Proc = id,
                                     Warning = FALSE))[1]:
f(Proc = op, Warning)
```

All	FALSE
Proc	op
Warning	TRUE

Example 3

To access the table of expected options and the list of unexpected arguments separately, use the index operator (see `_index`):

```
f := () -> prog::getOptions(1, [args()],
                               table(All = TRUE))[1]:
g := () -> prog::getOptions(1, [args()],
                               table(All = TRUE))[2]:
```

```
ExpectedOptions = f(Unexpected);
UnexpectedOptions = g(Unexpected)
```

$$\text{ExpectedOptions} = \begin{array}{|c|}\hline \text{All} \mid \text{TRUE} \\ \hline \end{array}$$

$$\text{UnexpectedOptions} = [\text{Unexpected}]$$

Example 4

When a user supplies unexpected arguments to your procedure, `prog::getOptions` can collect these arguments and return them as a list. Also, `prog::getOptions` can stop and issue an error when it finds the first unexpected argument. To issue an error instead of listing unexpected arguments, use `TRUE` as the fourth parameter of `prog::getOptions`:

```
f := () -> prog::getOptions(1, [args()], table(All = TRUE), TRUE):
f(Unexpected)
```

```
Error: The argument number 1 is invalid.
  Evaluating: f
```

Example 5

The `prog::getOptions` function does not distinguish parameters from options automatically. If some arguments of a procedure are parameters, exclude them from option verification. Otherwise, the `prog::getOptions` function lists those parameters as unexpected arguments. For example, `prog::getOptions` lists 1 and 2 as unexpected arguments of the function call `f(1, 2, All)`:

```
f := () -> prog::getOptions(1, [args()], table(All = TRUE))[2]:
UnexpectedOptions = f(1, 2, All);
```

$$\text{UnexpectedOptions} = [1, 2]$$

If you set the `prog::getOptions` function to error on unexpected arguments, it will error on the parameters too:

```
f := () -> prog::getOptions(1, [args()], table(All = TRUE), TRUE):
```

```
f(1, 2, All);
```

```
Error: The argument number 1 is invalid.
  Evaluating: f
```

To exclude first `n` parameters of a function from option verification, provide the number `n + 1` as a first argument of `prog::getOptions`. In a function call, specify all `n` parameters before you specify options. For example, to avoid checking the first two arguments in the function call `f(1, 2, All)`, use the following syntax:

```
f := () -> prog::getOptions(3, [args()], table(All = TRUE))[2]:
UnexpectedOptions = f(1, 2, All);
```

$$\text{UnexpectedOptions} = []$$

When you use `prog::getOptions`, you must provide all nonoptional parameters first, and then provide the options. The following syntax does not work because `prog::getOptions` assumes that the first two arguments in the function call `f(1, All, 2)` are parameters, and the number 2 is an option:

```
f := () -> prog::getOptions(3, [args()], table(All = TRUE), TRUE):
f(1, All, 2);
```

```
Error: The argument number 3 is invalid.
  Evaluating: f
```

Example 6

To specify acceptable types of the option values, use a table that contains acceptable options as indices and their expected types as entries. For example, specify that the `All` option must be of the type `DOM_BOOL`, and the `Proc` option must be of the type `Type::Function`:

```
f:= () -> prog::getOptions(1, [args()],
                           table(All = TRUE, Proc = id),
                           TRUE,
                           table(All = DOM_BOOL,
                                 Proc = Type::Function)
                           )[1]:
```

Now, options can have only values of the correct types:

```
f(All = FALSE, Proc = id)
```

All	FALSE
Proc	id

If you try to use a value of the wrong type, the function issues an error:

```
f(All = FALSE, Proc = 0)
```

```
Error: The type of argument number 2 is incorrect.
  Evaluating: f
```

Also, you can define and use a procedure for performing the type check. For arguments of the valid type, the procedure must return `TRUE` or an expression that can be evaluated to `TRUE` by the `bool` function:

```
f := () -> prog::getOptions(
                    2, [args()],
                    table(File = ""), TRUE,
                    table(File = (X -> contains({DOM_STRING,
                                                 DOM_INT},
                                                 type(X))))
                              )[1]:
f(FALSE, File = 0), f(TRUE, File = "test.log")
```

File	0	File	"test.log"

Example 7

Using arguments to separate options from parameters is not recommended because it can lead to errors. Always use the first parameter of `prog::getOptions` to specify how many parameters you have. Although prog::getOptions accepts any list as a second argument, the best practice is to use only `args()`. The following example demostrates that using arguments to separate options from parameters can result in the wrong error message. Although this error message correctly indicates that one of the options has a value of the wrong type, the index of the argument is wrong:

```
f := () -> prog::getOptions(1, [args(2..args(0))],
                               table(Option1 = TRUE),
```

```
                              TRUE,
                              table(Option1 = DOM_BOOL)):
f(x, Option1 = 1, Option2)


Error: The type of argument number 1 is incorrect.
  Evaluating: f
```

To get the correct error message for this example, use the first parameter of `prog::getOptions` to exclude `x` from option verification:

```
f := () -> prog::getOptions(2, [args()], table(Option1 = TRUE),
                              TRUE, table(Option1 = DOM_BOOL)):
f(x, Option1 = 1, Option2)


Error: The type of argument number 2 is incorrect.
  Evaluating: f
```

Parameters

`n`

A positive integer that specifies the number of the first option in the list of arguments. When calling the procedure, a user must provide all nonoptional paramaters before the options.

`arguments`

A list of all arguments of the procedure. Use `args()` to access all arguments.

`allOptions`

A `table` that contains all acceptable options as indices and their default values as entries: `table(Option = defaultValue)`

`ErrorOnUnexpected`

A Boolean constant `TRUE` or `FALSE`. If the value is `TRUE`, `prog::getOptions` issues an error when it finds an unexpected argument. If the value is `FALSE`, `prog::getOptions` collects all unexpected arguments and returns them as a list. By default, ErrorOnUnexpected = `FALSE`.

`optionTypes`

A `table` that contains acceptable options as indices and their expected types as entries: `table(Option = optionType)`. Here `optionType` must be a valid second argument of the `testtype` function or a procedure that returns `TRUE` (or an expression that can be evaluated to `TRUE` by the `bool` function) for arguments of the valid type. If you want to specify optionTypes, you also must explicitly specify ErrorOnUnexpected.

Return Values

List that contains a table and a list. The table contains all valid options of the calling procedure and their values. For expected options that are not specified in the procedure, the values are their default values. The list contains all unexpected arguments in the procedure.

See Also

MuPAD Functions

`testtype`

prog::init

Loading objects

Syntax

```
prog::init(object)
```

Description

`prog::init(object)` initializes the MuPAD object `object`.

Almost all MuPAD objects (domains, procedures etc.) are loaded into memory at their first use. This mechanism saves a lot of memory and time while starting MuPAD. Most of the MuPAD objects are not needed in a given session and would only fill up the system.

This strategy is transparent with respect to the usage of MuPAD objects. On slower computers, you may notice a delay on the first use of a function or domain.

Using `Pref::verboseRead`, you can make MuPAD print information on files loaded automatically.

Examples

Example 1

Initializing all objects from any additional MuPAD library increases the memory requirements. For example, you can initialize the object from the linalg library:

```
bytes()
```

```
                522304, 815604, 2147483647
```

```
prog::init(linalg):
```

Check the memory usage again:

```
bytes()
```

```
            15990660, 16507016, 2147483647
```

Example 2

Using `Pref::verboseRead`, we obtain information on what is loaded by the system:

```
reset():
Pref::verboseRead(2):
prog::init(prog::trace)
```

```
loading package
'prog' [lib/] reading file lib/PROG/checkini.mu reading file lib/PROG/trace.mu
```

Parameters

`object`

MuPAD object to initialize or option `All`

Options

`All`

Initializing all MuPAD objects

With this option (instead of some MuPAD object), all MuPAD objects will be initialized.

Return Values

`prog::init` returns the void object `null()`.

See Also

MuPAD Functions

`Pref::verboseRead` | `prog::check`

prog::isGlobal

Information about reserved identifiers

Syntax

```
prog::isGlobal(ident)
```

Description

`prog::isGlobal(ident)` checks whether the identifier `ident` is "used by the system". Here, "used by the system" means that `ident` is an environment variable (e.g., `PRETTYPRINT`), a system-wide constant (e.g., `PI` or `undefined`), an option (for some function call, e.g., `All`), or a system function (such as `sin`).

The most of those identifiers are protected (see `protect`).

Examples

Example 1

Assume you would like to use some identifiers as options for a new function you wrote. In this example, we will check the elements of the list `[All, Beta, Circle, D, eval, First]` for suitability. (Note that `eval` would not be a good choice, even if it was not a system function, because options should start with a capital letter.)

We define a test function which is mapped to the list and returns `FAIL`, if the tested object is not an identifier, `TRUE`, if the identifier is used by the system and `FALSE` otherwise:

```
LIST:= [All, Beta, Circle, D, eval, First]:
map(LIST, X -> if domtype(X) <> DOM_IDENT then
                 X = FAIL
               else
                 X = prog::isGlobal(X)
               end_if)
```

$$[\mathrm{All} = \mathrm{TRUE},\ \mathrm{Beta} = \mathrm{FALSE},\ \mathrm{Circle} = \mathrm{FALSE},\ D = \mathrm{FAIL},\ \mathrm{eval} = \mathrm{FAIL},\ \mathrm{First} = \mathrm{TRUE}]$$

The identifiers `All` and `First` can be used as options because they have already been protected by the system (actually, they are already used as options, which makes them a good choice), the identifiers `Beta` and `Circle` are free and one must only take care that they have no value if they will be used as options—they should be protected first. `D` and `eval` have values and cannot be used as options.

Parameters

`ident`

Identifier to check

Return Values

`prog::isGlobal` return `TRUE`, if the given identifier is used by the system, otherwise `FALSE`.

See Also

MuPAD Functions

`anames` | `domtype` | `prog::check` | `type`

prog::ntime

Hardware independent time unit

Syntax

```
prog::ntime()
```

Description

`prog::ntime()` returns a time unit that represents roughly the speed of the current machine *for typical library programs*.

`prog::ntime` can be used to perform timing tests of typical MuPAD library programs on different machines.

`prog::ntime` uses a mix of different operations to calculate the time factor.

One call to `prog::ntime` takes about 1.5 seconds.

A real timing value must be divided by the value of `prog::ntime`, to get a machine independent timing value.

Examples

Example 1

On this machine, a timing must be divided by the value of `prog::ntime`, then the timing is comparable with the timing of the same code on another machine, also divided by the value of `prog::ntime` on the other machine:

```
prog::ntime()
```

```
0.7052155095
```

Return Values

Floating point number

See Also

MuPAD Functions
`prog::testinit` | `time`

prog::profile

Display timing data of nested function calls

Syntax

```
prog::profile(stmt)
```

Description

`prog::profile(stmt)` evaluates the MuPAD statement `stmt` and displays timing data of all nested function calls, additionally a graph with the calling structure.

`prog::profile` measures and displays the time usage of *library functions*. Kernel functions are not measured. For every function called during the evaluation of `stmt`, `prog::profile` prints the time spent in this function and the number of calls.

`prog::profile` can be helpful in finding time critical functions and unnecessary nested function calls.

`stmt` could be reading a whole test file, too.

A trick to observe also kernel functions is to call `prog::trace` with the kernel function as argument. `prog::trace` takes a library wrapper procedure around the kernel function, that has the same name and can be found in the output of `prog::profile`, when the kernel function is used during the evaluation of `stmt`. The time use of the wrapper function is nearly zero.

The first part of the output is a table with the timing values for each procedure, the second part is a graph, that presents information about the dependences between all measured functions, when `stmt` is evaluated.

The table contains several columns that are described below.

Each row shows all data of one function (called “the function”), that was measured by `prog::profile`. There is one special entry: The first entry is called `procedure entry point`. It shows the sum of all functions and reperesents the evaluation of `stmt`.

- `"percent usage of all"`

  the time spent in the function with respect to the whole time used for the evaluation of `stmt` (in percent)

- `"time self per single call"`

  the value `"time self"` divided by the sum of all calls of the function (in milli seconds)

- `"time self"`

  the whole time spent *in the body* of the function, i.e., the sum of all calls, without the time, used by all other measured functions called by the function (in milli seconds)

- `"time children per single call"`

  the value `"time children"` divided by the sum of all calls of the function (in milli seconds)

- `"time children"`

  the sum of *all time* (`self` and `children`) spent in all functions that are called by the function directly

- `"calls/normal exit"`

  number of all calls of the function that leave the function without errors

- `"calls/remember exit"`

  number of all calls of the function that return a remembered value by the kernel remember mechanism (and does not call the function body)

- `"calls/errors"`

  number of all calls of the function that leave the function with an error

- `["[index] function name"]`

  the index of the function (assigned by `prog::profile`) and the name of the function

An index is assigned to each function, in descending order of time usage, to identify and find the function in the call graph that is described now.

The second part of the output of `prog::profile` is a dependence graph. It shows each function, their parents (functions that call the function directly), and their children

(functions that are called from the function directly), together with timing information and the number of calls.

Each part of the graph that is separated by horizontal lines of minus chars, belongs to one function. It contains several columns:

- `"index"`

  the index, assigned unique to the function
- `"%time"`

  the percentage of the function on the whole run time
- `"self"`

  the sum of all times used by the function (in milli seconds)
- `"children"`

  the sum of all times used by the children of the function (in milli seconds)
- `"called"`

  the number of all calls of the function
- `["[index] name"]`

  the index and the name of the function

  There are two kinds of entries: the function that belongs to the part has its index in the first column of the part, and in this column, only their name is printed.

  All other functions (parents and children of the function) are only printed in this column with their index and name together, with small indentation for highlighting the function that belongs to the part.

  The parents are located above the function itself, all children are written below the line with the function, the part belongs to.

  For a more detailed explanation of the lines in a graph part see the first example.

Examples

Example 1

We define three functions `f`, `g` and `h`. `prog::profile` displays the time spent in each function and the number of calls to it:

```
f := proc()
       local t;
     begin
       t := time();
       while time() < 10 + t do nothing end_while
     end_proc:
g := proc()
       local t;
     begin
       f();
       t := time();
       while time() < 10 + t do nothing end_while;
       f()
     end_proc:
h := proc() begin g(), f(), g() end_proc:

prog::profile(h()):
```

```
  percent usage of all     |    time self per
single call     |      |    time self     |      |      |    time
children per single call     |      |      |      |    time children
    |      |      |      |      |   calls/normal exit     |      |
     |      |      |   |  calls/remember exit     |      |      |
     |      |   |  |  calls/errors     |      |      |      |
 |   |  |  |  [index] funct. name ------------------------------------------------
 100.0   70.0   70.0     .      .   1  .  .  [0]  proc. entry pt.
-----------------------------------------------------------------
  71.4   10.0   50.0     .      .   5  .  .  [1]  f   28.6   10.0
  20.0   20.0   40.0  2  .  .  [2]  g     .      .      .    70.0
  70.0  1  .  .  [3]  h ------------------------------------------------------------
  index     %time      self  children     called  [index] name ---------------------
  [0]     100.0        70         0          1  proc. entry point
                        0        70          1      [3]  h ---------------------------
                       40         0          4      [2]  g
                10         0          1      [3]  h   [1]      71.4
       50          0          5  f ---------------------------------------------------
```

```
                      20        40          2      [3]  h   [2]
    28.6        20        40          2  g
40          0          4      [1]  f ------------------------------------------------
  [3]       0.0          0        70          1  h
       10          0          1      [1]  f
20        40          2      [2]  g ------------------------------------------------
  Time sum: 70 ms
```

(The output is shortened slightly, because the page is too small.)

The lines of the table above are described following:

```
  percent usage of all     |    time self per
single call     |      |    time self     |      |      |    time
children per single call     |      |      |      |    time children
    |      |      |      |      |   calls/normal exit     |      |
     |      |      |   |  calls/remember exit     |      |      |
     |      |   |  |  calls/errors     |      |      |      |
 |   |  |  |  [index] funct. name ------------------------------------------------
 100.0   70.0   70.0     .      .   1  .  .  [0]  proc. entry pt.
-----------------------------------------------------------------
```

The whole function call takes `100` percent of the time (certainly), `70.0` milli seconds and is called once (the evaluation of `stmt`), without an error.

```
  71.4   10.0   50.0     .      .   5  .  .
 [1]  f
```

Function `f` takes `71.4` percent of all evaluation time in its body. It uses `10.0` milli seconds per call on the average (in this case exactly), their children (if existing) uses no time measurable (because it has no children), and it is called `5` times and returns without errors. `f` has the index `1`.

```
  28.6   10.0   20.0   20.0   40.0  2  .  .
 [2]  g
```

Function `g` takes `28.6` percent of the whole time that are `20.0` milli seconds, and `10.0` milli seconds on the average per call. Their children uses `20.0` milli seconds pre call on the average and `40.0` total, and `g` is called twice and returns without errors. `g` gets the index `2`.

```
    .      .      .    70.0   70.0  1  .  .
 [3]  h
```

Function h uses nearly no evaluation time, their children uses 70.0 milli seconds on the average and 70.0 total, and h is called once and finished without errors. h gets the index 3.

The parts of the graph above are described following:

```
index     %time      self  children     called
 [index] name ------------------------------------------------------------------
  [0]     100.0        70         0          1  proc entry point
                       0        70          1      [3]  h ----------------------------
```

The whole function call takes 100 percent of the evaluation time (by definition), that are 70.0 milli seconds, and it is called once.

It calls once the function h with index [3] (as argument of prog::profile), and h uses 70.0 milli seconds of the time that are spent in the children of h, not in the body.

```
index     %time      self  children     called
 [index] name ------------------------------------------------------------------
                      40         0          4      [2]  g
                10         0          1      [3]  h   [1]      71.4
       50          0          5  f ---------------------------------------------------
```

Function f spends 71.4 percent of the whole evaluation time. It uses 50.0 milli seconds, their children uses no time measurable, and it is called 5 times.

f has two parents and no children.

f is called by its parent g4 times and by h once.

f spends 40 milli seconds by itself (in its body), when it is called from g (the first line in the part of f), and f spends 10 milli seconds in its body, when it is called from h (the second line).

```
index     %time      self  children     called
 [index] name ------------------------------------------------------------------
                      20        40          2      [3]  h   [2]
    28.6        20        40          2  g
40          0          4      [1]  f ---------------------------------------------------
```

Function g takes 28.6 percent of the whole time, 20 milli seconds in its body, and their children take 40 milli seconds. g is called twice.

g is called from h twice, and spends 20 ms in its body and 40 ms in its children.

g calls the function f four times, and f spend 40 ms in its body, when it is called from g.

```
index     %time      self  children     called
 [index] name ------------------------------------------------------------------
  [3]       0.0         0        70          1  h
       10          0          1      [1]  f
20         40          2      [2]  g ------------------------------------------------
```

Function h takes nearly no evaluation time, their children spends 70.0 milli seconds, and h is called once.

h calls the functions f and g directly, f once and g twice.

f uses 10 ms in its body, when it is called from h, and g uses 20 ms in its body and 40 ms in its children.

Parameters

stmt

A MuPAD statement

Return Values

Result of stmt

Algorithms

The timings displayed by prog::profile are generated by the kernel.

The evaluation of stmt inside prog::profile takes partly substantially longer than evaluating stmt directly. This extra time does not influence the validity of the timings, i.e., if prog::profile reports f taking three times as long as g, then this is also the case when evaluating stmt directly.

See Also

MuPAD Functions

prog::trace | time

prog::remember

Extended remember mechanism for procedures

Syntax

```
prog::remember(f, <depends>, <PreventRecursion, <predef>>)
```

Description

`prog::remember(f)` returns a modified copy of the procedure `f` that stores previously computed results and additional information in a remember table. When you call `f` with arguments that you already used in previous calls, `f` finds the results in its remember table and returns them immediately.

If you assign `f` to an identifier or a domain slot, you also must assign the copy returned by `prog::remember` to the same identifier or slot, for example, `f := prog::remember(f)`.

`f :=prog::remember(f)` remembers results without context information, such as properties or the value of `DIGITS`. The first time you call `f` with any new combination of input parameters, the remember table of `f` stores `` `input` -> `f(input)` ``. After that, when you call `f` with the same input parameters, it takes the result `f(input)` from the remember table instead of recomputing it. See “Example 1” on page 27-42.

`f := prog::remember(f, depends)` remembers results and additional context information. The dependency function `depends` lets you specify the context information to store along with computed results in the remember table and verify in each function call. See “Example 2” on page 27-43.

Typically, it is useful to store and verify properties of the input and the values of `DIGITS` and `ORDER`. To access properties of the input, use `property::depends`. This dependency function verifies all three values:

```
() -> [property::depends(args()), DIGITS, ORDER]
```

Another common problem is that an overloading function does not register when its overloading slot changes in some other function or domain. This dependency function that uses `slotAssignCounter` lets you avoid this problem:

```
() -> [property::depends, slotAssignCounter("foo")]
```

To combine all three tasks, use this dependency function:

```
() -> [property::depends(args()),
   DIGITS, ORDER, slotAssignCounter("foo")]
```

The first time you call `f` with any new combination of input parameters, the remember table of `f` stores `` `[input, depends(input)]` -> `f(input)` ``. After that, when you call `f` with the same input parameters, it checks whether `depends(input)` returns the same value as before. If it does, then `f` takes the result `f(input)` from the remember table. Otherwise, it computes `f(input)` and adds the new values `` `[input, depends(input)]` -> `f(input)` `` to the remember table. The only exception to this rule is results computed with different values of `MAXEFFORT`. If in previous calls `f(input)` was computed with lower `MAXEFFORT`, then the new call with higher `MAXEFFORT` is evaluated and remembered results are replaced with the new ones.

If the dependency function is constant or returns the value that does not depend on the input, then the remember mechanism disregards context information.

You can call the modified procedure with the `Remember` option as the first argument, and one of these special options as the second argument:

- `Clear` clears the remember table of the procedure.
- `ClearPrevent` clears the remember table that prevents infinite recursions inside the procedure. For details about preventing infinite recursions, see the description of the `PreventRecursion` option.
- `Print` returns the remember table of the procedure.

For example, the call `f(Remember, Clear)` clears the remember table of `f`. Also see “Example 3” on page 27-45.

Examples

Example 1

Create this function:

```
f := X -> if X > 1 then f(X - 1)*f(X - 2) - f(X - 2) else 1 end_if:
```

Calling this function is time-consuming because the function calls itself recursively and evaluates every call:

```
f(20), time(f(20))
```

```
0, 3260.204
```

Using the remember mechanism eliminates these reevaluations. To enable the remember mechanism, use `prog::remember`:

```
f := prog::remember(f):

f(200), time(f(200))
```

```
0, 0.0
```

Example 2

Create the procedure `pos` that checks if its parameter is positive:

```
pos := proc(x)
       begin
         is(x > 0)
       end_proc:
```

Enable the remember mechanism for `pos`:

```
pos := prog::remember(pos):
```

`pos` returns `UNKNOWN` for variable `a`:

```
pos(a)
```

```
UNKNOWN
```

Now use `assume` to specify that variable `a` is positive:

```
assume(a > 0):
```

When you call `pos` for variable `a`, it finds the value of `pos(a)` in the remember table. In this case, the remember table does not store the context information, and therefore

does not check for the new assumptions on variable `a`. It returns the remembered result, which is incorrect because of the new assumption:

```
pos(a)
```

```
UNKNOWN
```

Calling `pos` for `a^3` returns the correct result because `pos(a^3)` is not in the remember table yet:

```
pos(a^3)
```

```
TRUE
```

Assume that `a` is negative:

```
assume(a < 0):
```

Now both calls return incorrect values because the results are taken from the remember tables:

```
pos(a), pos(a^3)
```

```
UNKNOWN, TRUE
```

To make the remember mechanism aware of the changes in assumptions, use `prog::remember` with the second argument `property::depends` as the dependency function:

```
unassume(a):
pos := proc(x)
       begin
         is(x > 0)
       end_proc:
pos := prog::remember(pos, property::depends):
pos(a)
```

```
UNKNOWN
```

Now `pos` reacts properly to the new assumption:

```
assume(a > 0):
```

```
pos(a)
```

$$\text{TRUE}$$

`pos` also returns the correct result after you clear the assumption:

```
unassume(a):
pos(a)
```

$$\text{UNKNOWN}$$

Example 3

Create the procedure `pos` and enable the remember mechanism for it:

```
pos := proc(x)
       begin
         is(x > 0)
       end_proc:
pos := prog::remember(pos, getprop):
```

Call `pos` for these parameters:

```
pos(a):
assume(b > a, _and):
pos(b):
```

After you call the procedure at least once, it creates the remember table. To see the remember table of a procedure, use the special option `Print`. The value 10^6 in the second column is the value of `MAXEFFORT` used during computations.

```
pos(Remember, Print)
```

[pos(a), [$\mathbb{C}$, 0]]	[UNKNOWN, 1000000.0]
[pos(b), [(a, ∞), 0]]	[UNKNOWN, 1000000.0]

To clear the remember table of a procedure and thus force the function to reevaluate all results, use the special option `Clear`:

```
pos(Remember, Clear):
pos(b)
```

UNKNOWN

Example 4

Create the procedure `deps` that collects all operands of the properties of a given expression, including the identifiers of assumed properties:

```
deps := proc(x)
        begin
          if domtype(x) <> DOM_IDENT then
            op(map(indets(x), deps))
          else
            x, deps(getprop(x))
          end_if
        end_proc:
```

Set the following assumption. Note that now `deps` contains potentially infinite recursions because the property of `x` refers to `y`, and the property of `y` refers back to `x`:

```
assume(x > y):

deps(x)
```

```
Error: Recursive definition [See ?MAXDEPTH]
```

To prevent infinite recursions, use `prog::remember` with the `PreventRecursion` option:

```
deps := prog::remember(deps, PreventRecursion):
deps(x)
```

$$x, y, x$$

To simplify the return value of `deps`, rewrite the function so that it returns a set of all identifiers:

```
deps := proc(x)
        begin
          if domtype(x) <> DOM_IDENT then
            _union(op(map(indets(x), deps)))
          else
            {x} union deps(getprop(x))
          end_if
```

```
        end_proc:
deps := prog::remember(deps, PreventRecursion):

deps(x)
```

$$\{x, y\} \cup x$$

Now `deps` expects the return value to be a set. By default, when recursion is detected, the procedure returns the value of its input (which is not a set in this example). When preventing recursion in a procedure where the type of the input differs from the type of the return value, specify the value `predef` that the procedure returns when recursion is detected:

```
deps := proc(x)
        begin
          if domtype(x) <> DOM_IDENT then
            _union(op(map(indets(x), deps)))
          else
            {x} union deps(getprop(x))
          end_if
        end_proc:
deps := prog::remember(deps, PreventRecursion, () -> {args()}):
```

Here `predef` returns a set with the input as an operand:

```
deps(x)
```

$$\{x, y\}$$

Parameters

`f`

A procedure or function environment

`depends`

A procedure or expression

`predef`

A procedure or expression

Options

`PreventRecursion`

With this option, the procedure returned by `prog::remember` uses remembered information to prevent infinite recursion inside the procedure.

`f := prog::remember(f, PreventRecursion, predef )` stores the input parameters only during the function call. This approach lets you avoid reevaluating the same function call when the function calls itself recursively. Instead, it returns the input (by default) or the result of the call `predef(input)` (if you specify `predef`). If returning the input is not an appropriate result for the function call (for example, if the return value of `f` and the input are of different types), then you must specify the value `predef`. See "Example 4" on page 27-46.

At the end of the function call, all remembered values are discarded. If you call the function with the same input parameters again, the function call is evaluated with the same costs as before.

You can prevent recursion inside the function call and simultaneously use the remember mechanism outside the function call by using this syntax: `f := prog::remember(f, depends, PreventRecursion, predef )`. If you want to use the remember mechanism with the context information, specify the dependency function `depends` as usual. If you want to use the remember mechanism without the context information and prevent recursions inside a procedure, specify `depends` as a constant (or any function whose return value does not depend on the input). Note that if you omit the `depends` function and just use the syntax `f := prog::remember(f, PreventRecursion, predef )`, then the remember mechanism does not work outside the function call. In this case, you only prevent recursions.

Return Values

Modified procedure or function environment

See Also

MuPAD Functions

`proc` | `property::depends` | `slotAssignCounter`

More About

- "Remember Mechanism"

prog::sort

Sort objects by an index function

Syntax

```
prog::sort(list, func, <Reverse>, <p1, p2, …>)
```

Description

`prog::sort(list, func)` applies the function `func` to any object of the list `list` and returns a list with the given objects sorted by the order of the indices calculated by `func`.

`func` is applied only once to any object in `list`.

If optional arguments are present, then the indices are computed from the objects `x` of `list` by `f(x, p1, p2, ...)`.

An alternative call to `prog::sort` is the call `sort(list, (X, Y) -> func(X) <= func(Y))`.

Examples

Example 1

Sort a list of expressions by their length:

```
prog::sort([2*x, x - 4, sin(x), x + y + z], length)
```

$$[\sin(x),\ 2\,x,\ x+y+z,\ x-4]$$

Sort a list of lists by the number of operands, with descending order:

```
prog::sort([[1,2,3],[4,2],[0 $ 10],[]], nops, Reverse)
```

$$[[0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0],\ [1,\ 2,\ 3],\ [4,\ 2],\ []]$$

Parameters

`list`

A list of MuPAD objects

`func`

A function that must return a numerical value, when applied to any object of the list

`Reverse`

An option

`p1, p2, …`

Any MuPAD objects accepted by `func` as additional parameters

Options

`Reverse`

`prog::sort` compares the calculated indices in reverse order.

Return Values

List with the same objects as the given list

See Also

MuPAD Functions

`sort` | `stringlib::order` | `sysorder`

prog::tcov

Report on test coverage (passed program lines)

Syntax

```
prog::tcov(Reset)

prog::tcov(<stmt>, Write = fname)

prog::tcov(Append = fname)

prog::tcov(<stmt>, Info, <Summary>, <Lines>, <Hidden>, <Unused>, <All>)

prog::tcov(<stmt>, Annotate, <Path = pname>, <Comment = comment>)

prog::tcov(<stmt>, Export = fname, <Annotate>, <Path = pname>, <Comment = comment>, <Gr
```

Description

`prog::tcov` inspects the data on the statements executed in library code. MuPAD collects these data if you start the MuPAD engine with the option -t. To set different options for starting the MuPAD engine, use the **Arguments** field in the Engine dialog.

You can use `prog::tcov` in two different modes:

- With a given first argument stmt, `prog::tcov` resets all tcov information, evaluates the statement, and shows all debug node passes for this statement during the evaluation. `prog::tcov`(stmt) clears all information about the debug node passes of current session.
- Without the first argument stmt, `prog::tcov` shows the debug node pass information collected by the MuPAD engine during the whole session.

You can display the logged debug node passes on the screen or export the data to an HTML file. You also can save the data about the debug node passes to a file, which enables you to read or recover a whole session state later.

`prog::tcov` can produce annotated source files containing the information collected by `prog::tcov` and the MuPAD source code.

Environment Interactions

`prog::tcov` can produce screen outputs.

If you use the `Annotate` option, `prog::tcov` creates new files. For some operating systems creating new files might require special permissions.

Examples

Example 1

To use `prog::tcov`, start the kernel in tcov mode with option -t.

The outputs in the following examples are cropped in this documentation.

The following example shows a short procedure created and called inside `prog::tcov`. The line numbers correspond to the ones in the Debugger:

```
f:=                  // 1
proc(a, b)           // 2
begin                // 3
  if a > b then      // 4
    return(a)        // 5
  else               // 6
    return(f(b, a))  // 7
  end_if             // 8
end_proc:            // 9
```

`prog::tcov` returns all the lines passed during the evaluation of `f`:

```
prog::tcov(f(2, 1), Info, Lines)

   File: /tmp/debug0.5932  Use index: 50%
     Nodes: 2/4 (0 hidden)   Passes: 2          Line       4,0:
  1 pass          Line       5,0:     1 pass

prog::tcov(f(1, 2), Info, Lines)

   File: /tmp/debug0.5932  Use index: 75%       Nodes:
3/4 (0 hidden)   Passes: 4          Line       4,0:     2 passes
        Line       5,0:     1 pass          Line       7,0:     1
pass
```

Example 2

The following example shows the logging of passes during a session. Before running this example, define the function `f` from “Example 1” on page 27-53.

If you start the kernel with the option -v, the expose command shows the debug nodes with pass information.

`prog::tcov(Reset)` resets all tcov information:

```
prog::tcov(Reset):
prog::tcov(Info)

SUMMARY   Files
   : 1 in 1 libraries   Nodes     : 0/4 (0 hidden)   Use index : 0%
  Passes    : 0 (~  0.00 passes per all nodes)
```

If you call the function `f` twice, the number of passes doubles:

```
f(1, 2): f(1, 4): prog::tcov(Info)

   File: /tmp/debug0.5932  Use index: 75%
     Nodes: 3/4 (0 hidden)   Passes: 8          Line       4,0:
  4 passes          Line       5,0:     2 passes          Line
   7,0:     2 passes
```

To see the passes, expose the function `f`:

```
expose(f)


proc(a, b)     name f;   begin     // /tmp/debug0.5932:4,0 [4 passes];
    if b < a then       // /tmp/debug0.5932:5,0 [2 passes];
   return(a)     else       // /tmp/debug0.5932:7,0 [2 passes];
    return(f(b, a))     end_if;     // /tmp/debug0.5932:9,0 [0 passes]
  end_proc
```

You can write the tcov data to a data file:

```
prog::tcov(Write = "tcov_example.dat"):
```

To delete the information about the previous passes, use the `Reset` option:

```
prog::tcov(Reset):
```

```
prog::tcov(Info)
```

```
SUMMARY   Files
   : 1 in 1 libraries   Nodes     : 0/4 (0 hidden)   Use index : 0%
  Passes    : 0 (~  0.00 passes per all nodes)
```

To retrieve the former state, use the Append option:

```
prog::tcov(Append = "tcov_example.dat"):
prog::tcov(Info, Summary)
```

```
SUMMARY
  Files     : 1 in 1 libraries   Nodes     : 3/4 (0 hidden)   Use
index : 75%   Passes    : 8 (~  2.00 passes per all nodes)
```

Also, you can use the Append option to add the passes:

```
prog::tcov(Append = "tcov_example.dat"):
prog::tcov(Info, Summary)
```

```
SUMMARY
  Files     : 1 in 1 libraries   Nodes     : 3/4 (0 hidden)   Use
index : 75%   Passes    : 16 (~  4.00 passes per all nodes)
```

Example 3

The following example presents incomplete pieces of code. Note that you cannot execute this example without additional code lines.

Suppose, you have a source file with the following function:

```
1:  f := proc(a, b)
2:       begin
3:         if a > b then
4:           return(a)
5:         elif a = b then return(0)
6:         else
7:           f(b, a)
8:         end_if
9:       end_proc:
```

Before executing this source file, `read` the commands. After reading commands, all the objects defined in the source file are available in the notebook. Calling the function `f` several times and creating the annotated source file, you get:

```
ff(2, 1): // passing lines 3 and 4
f(1, 1): // passing lines 3 and 5 twice
         // because line 5 has two debug nodes
f(1, 2): // passing lines 3, 5, and 7 and
         // recursively 3 and 4, leave via line 9
         // because the statement in line 7 has no return
prog::tcov(Annotate)
```

The annotated source file uses the same path as the source file and looks like this:

```
// Generated by prog::tcov session

 1:  f := proc(a, b)
 2:       begin
 3:4         if a > b then
 4:2           return(a)
 5:2         elif a = b then return(
 5:1                                0)
 6:          else
 7:1           f(b, a)
 8:          end_if
 9:1       end_proc:
```

Note that the line 5 contains two debug nodes and appears in two lines. The line splits where the second debug node starts.

For better readability of the annotated source files, use the HTML export.

Parameters

`stmt`

Any MuPAD statement or MuPAD expression

`fname`

A file name given as a string

`pname`

A directory name given as a string

`comment`

Any string

Options

`Reset`

Reset the number of passes at each debug node to 0.

`Write`

Option, specified as `Write = fname`

This option allows you to write the information about all debug node passes of the current session to the file fname. You can use this file for external analysis (see the Algorithms section) or to recover or merge the information collected by `prog::tcov` (see the `Append` option).

`Append`

Option, specified as `Append = fname`

Append all information about debug node passes from the file fname to the current session.

This option allows you to merge the data generated during several sessions.

`Info`

Display the information about debug node passes.

`Summary`

Display only a short summary.

`Lines`

Display each pass through a debug node.

`Unused`

Display all code lines with debug nodes, including unpassed ones.

Hidden

This option allows you to display the hidden debug nodes. A hidden debug node is a node in a procedure with the noDebug option.

All

Display unpassed and hidden debug nodes.

Export

Option, specified as `Export = fname`

This option allows you to display the debug node passes information in summary for all read source files and for all individual source files.

The information is ordered according to the names of the directories containing the source files. Directory names can be folded.

You can see the list of all the files of a library below each library name.

Each file name presents a link that points to the annotated source file.

You can select graphical indices. Each point leads to the related line in the annotated source file.

Annotate

This option allows you to rewrite each executed MuPAD source file `filename.mu` as `filename.tcov` with an annotation at the beginning of each line. The annotation contains the line number of a debug node and the number of passes of this line, followed by the source code.

In text mode the line containing several debug nodes splits so that each line contains one debug node (see “Example 3” on page 27-55).

The new files have the extension '.tcov' instead of '.mu'. See also option `Path`.

If this option is used together with `Export`, `prog::tcov` creates the annotated source files as `HTML` files with the extension `.tcov.html`. The line colors depend on the passes.

Path

Option, specified as `Path = pname`

This option allows you to specify a path pname to the annotated source files and the exported status file. If you do not specify the path, `prog::tcov` creates the files in the same directory where the source files are.

`Comment`

Option, specified as `Comment = comment`

This option allows you to write the string comment on the first line of each annotated source file (see the option `Annotate`) or in the header of an exported HTML file (see the option `Export`).

`Graphical`

Show a graphical index for each source file in an HTML export file.

Return Values

`prog::tcov` returns the void object `null()` of type `DOM_NULL`.

Algorithms

To be able to use the `prog::tcov` function, start the MuPAD engine with option -t. Use the *Arguments* field in the Engine dialog to set this option.

If you start the kernel using both options `-v` and `-t`, the function expose shows information about the debug node and passes (see “Example 2” on page 27-54).

The functionality of `prog::tcov` depends on the internal debugger. For details, see the help page for the `debug` command.

Some special considerations:

- If the MuPAD library is read from a tar archive (file `lib.tar`), `prog::tcov` excludes from inspection all the files from this archive. The output of `prog::tcov` also includes the call of `prog::tcov` itself and some other MuPAD utility function passes.
- `prog::tcov` counts only the lines containing a “debug node”.

When called with the option `Write` or the option `Append`, `prog::tcov` creates a data file using the following format: `"filename":fileindex:`. For each read MuPAD source

file, "filename" is the name of the source file and fileindex is a numerical index. For temporary files, the index is negative:

- `-1:-1:`

The colon separates the first and the second parts:

- `fileindex:line:column:hidden:passes:unused:`

For each debug node, fileindex corresponds to the first part, line and column determine the start of the debug node in the source file, hidden is 1 for hidden nodes, otherwise 0, passes is the number of passes, unused is an empty and currently unused string.

See Also

MuPAD Functions

`debug` | `expose` | `prog::check` | `prog::profile` | `prog::trace`

prog::test

Automatic comparing of calculation results

Syntax

```
prog::test(stmt, res | TrapError = errnr, <Timeout = sec>, <message>, options)
```

```
prog::test(stmt)
```

Description

`prog::test` works in two different modes: interactive and inside of test files.

In interactive mode a single call of `prog::test` can be used to compare two MuPAD statements.

The call `prog::test(stmt, res)` evaluates both arguments `stmt` and `res`. When the evaluation leads to exactly the same MuPAD object and no `Enhancement` was requested, nothing is printed and `prog::test` returns the void object `null()`.

If the results are different, the test fails and a message is printed.

The additional arguments are described in the following part for using `prog::test` in test files.

Another mode is using `prog::test` inside of test files. A test file must start with `prog::testinit` . This function initializes the test file. Then you can write several tests using `prog::test`. The last statement in a test file must be `prog::testexit()`. You also can specify the name of the tested procedure by using `print(Unquoted, "testname")` after `prog::testinit`. This name does not affect the tested procedure itself. It only appears in the test reports generated by your test script.

The tests can be arbitrary MuPAD statements and `prog::test` statements. However, most of the functionality should be executed as argument of `prog::test`. Only initialization of variables should be performed outside of `prog::test` statements in a test file, because `prog::test` traps every error (with the function `traperror`) and prints a specific error message.

Note: If an error occurs outside of `prog::test`, reading of the test file is interrupted.

If no error occurs (as should be the default case), the results are compared and a message is printed, if they are different.

Timing information can be collected and compared that consider only the evaluation time of the first argument `stmt` of `prog::test` (see `prog::testinit`).

If a test fails, for example, the two first arguments of `prog::test` lead to different MuPAD objects, or if an enhancement request was given, `prog::test` prints a message. This message lists the following pieces of information:

1. The first line starts with the `Error in test` string and contains the name and a sequence number of the individual test.
2. The next three lines contain the input, the expected result, and the result actually observed.
3. For each of the options `Priority`, `Enhancement`, `Message`, `Developers`, and `BugId`, if the option has been set, a corresponding line will be printed. Note that `Message` can be set by simply providing a message string.

This information is followed by an empty blank line.

If only one argument is given, the argument is evaluated and compared with `TRUE`, i.e., `prog::test(ex)` is equivalent to `prog::test(ex, TRUE)`.

When a test is initialized with `prog::testinit` and ended by `prog::testexit`, a short message is printed with the following format:

```
Info: 20 test, 1 error, runtime factor 1.7 (expected 2.0)
```

The message contains the number of all tests performed (20), the number of errors (1), and two time factors: The first time factor is the based on the actual time of the test and the second time factor is the expected value given by `prog::testinit`.

Examples

Example 1

`prog::test` can be called interactively:

```
prog::test(1 + 1, 2):
prog::test(is(2 > 1)):
prog::test(sin(PI), 0, "check sin"):
```

These tests checked all right. In the next tests wrong results are tested against, to demonstrate the messages given by `prog::test`:

```
prog::test(1 + 2, 2):

Error in test 4

Input: 1 + 2

Expected:  2

Got:       3

Near line: 1


prog::test(is(x > 1)):

Error in test 5

Input: is(1 < x)

Expected:  TRUE

Got:       UNKNOWN

Near line: 1


prog::test(sin(PI), PI, "check sin"):

Error in test 6

Input: sin(PI)

Expected:  PI

Got:       0

Message: check sin

Near line: 1
```

Example 2

A test file must contain calls to `prog::testinit` and `prog::testexit`. In the following file, we test a function defined in the same file, which is rather uncommon, obviously.

```
// test file "test.tst"
test:= (a, b) -> a^2 + 2*b^2 - a*b:
prog::testinit("test", 0.1):
print(Unquoted, "testname"):
prog::test(test(1, 4), 29, Message = "my first example"):
prog::test(test(3, -2), 24, "the second example"):
prog::test(error("test"), TrapError = 1028):
prog::testexit():
```

The first statement is only a comment. The second line contains an initialization of a test procedure called `test`. Then the test is initialized with `prog::testinit`.

After that three tests are performed: The first test is right, the second expected result is wrong, and the third test produces an error, but the expected result is this error, the error number returned by `traperror` is `1028` (user call of `error`).

The whole test takes nearly no time:

```
read("test.tst")

Info: memory limit is 256 MB

Error in test 2

Input: test(3, -2)

Expected:  24

Got:       23

Message: the second example

Near line: 4


Info: time used outside of 'prog::test' takes 100%

Info: 3 tests, 1 error, runtime factor  0.0  (expected  0.1)

Info: CPU time:  1.1 s
```

```
Info: Memory allocation 9026800 bytes [prog::testexit]
```

Example 3

Most of the options accepted by `prog::test` are more or less directly placed in the output:

```
prog::test(1+1, 1, Baseline, Message(2)="well ...",
  Priority=Low, BugId="123-456")

Baseline Error in test 7

Input: 1 + 1

Expected:  1

Got:       2

Priority: Low

Message: well ...

BugId: 123-456

Near line: 2
```

Example 4

To test that a certain call does not take longer than a specified number of seconds, use the option `Timeout`:

```
prog::test(prog::wait(5.0), null(), Timeout = 2)
```

Example 5

In most cases, the actual and the expected result are simply compared for equality. Sometimes, however, this is not desirable, especially for floating-point results:

```
prog::test(float(PI), 3.1415926535897932385)

Error in test 8

Input: float(PI)
```

```
Expected:  3.141592654

Got:       3.141592654

Near line: 1
```

The problem here is that there are many floating-point values which are not identical, yet are displayed as such (unless you increase `DIGITS` far enough to see the difference). Using the option `Method`, you can provide a function to compare the values:

```
prog::test(float(PI), 3.1415926535897932385, Method = `~=`)
```

Example 6

When implementing symbolic algorithms, there are often multiple correct and acceptable answers. In some cases, getting any of a certain set of solutions is fine. In these cases, using `Method = _in` is a reasonable way of writing tests (`_in` is the functional form of the `in` operator):

```
prog::test(int(ln(ln(a*x)^(1/2)), x),
  {
    x*ln(ln(a*x)^(1/2)) - Li(a*x)/(2*a),
    x*ln(ln(a*x))/2 - Li(a*x)/(2*a)
  },
  Method = _in,
  Timeout = 20)
```

Sometimes, however, while multiple results are acceptable, you are actually targeting for one particular output. For these cases, you can use `Enhancement` to set the golden goal:

```
prog::test((x^2+2*x+1)/(x+1),
  (x^2+2*x+1)/(x+1),
  Enhancement = x+1)

Enhancement request: 11

Input: (x^2 + 2*x + 1)/(x + 1)

Got:       (x^2 + 2*x + 1)/(x + 1)

Requested: x + 1

Near line: 3
```

If the enhancement request ever is fulfilled, the output changes:

```
prog::test(normal((x^2+2*x+1)/(x+1)),
  (x^2+2*x+1)/(x+1),
  Enhancement = x+1)

Enhancement done: 12

Input: normal((x^2 + 2*x + 1)/(x + 1))

Got:       x + 1

Requested: x + 1

Near line: 3
```

Note that a test with an enhancement request is, first and foremost, still an ordinary test and behaves as such:

```
prog::test((x^2+x+1)/(x+1),
  (x^2+2*x+1)/(x+1),
  Enhancement = x+1)

Error in test 13

Input: (x^2 + x + 1)/(x + 1)

Expected:  (x^2 + 2*x + 1)/(x + 1)

Got:       (x^2 + x + 1)/(x + 1)

Near line: 3
```

Parameters

`stmt`

A MuPAD statement to test

`res`

A MuPAD expression or statement that determines the expected result.

message

A message (a string) that is displayed if the test fails – see option `Message` below

Options

TrapError

Option, specified as `TrapError = errnr`

Expect the test to throw an error. errnr must be the integer expected from the call `traperror(stmt)` or a list of an integer and a string, as returned by `getlasterror()`.

Method

Option, specified as `Method = comp`

A method used to compare the actual and the expected result. Will be called with both expressions and must return `TRUE` or `FALSE`.

Timeout

Option, specified as `Timeout = sec`

A timeout for the evaluation of the tests. Both the actual and the expected result are evaluated with this time limit. If the computation takes too long, `prog::test` behaves as if the command had resulted in a timeout error (error number 1320).

Message

Option, specified as `Message = message` or `Message(res1) = message`

Append a message (a string) to the output of `prog::test`. If res_1 is given, the message is given if the result of evaluating stmt is res_1.

Baseline

Mark this test as failing in some sort of "baseline," to differentiate new bugs (stemming from new code developments, regression failures) from bugs already present in some specific earlier version. This affects the output of `prog::test`.

`Enhancement`

Option, specified as `Enhancement = res1`

Request some other output than the one currently tested for. Semantically, a call of the form `prog::test(inp, out, Enhancement = out2)` means "check that the call `inp` results in the same thing as the call `out`, but note that we'd actually prefer to see `out2`."

`ExpectedWarnings`

Option, specified as `ExpectedWarnings = list`

Gives a list of warnings the call should emit, as strings. Not emitting these warnings, or additional ones, is considered an error.

`High`, `Low`, `Medium`, `Priority`

Option, specified as `Priority = Low | Medium | High`

Denote the importance of this test. This will usually be a very subjective question and affects the output of `prog::test` only, to allow tools parsing the output displaying the problems of higher priority more prominently.

`Developers`

Option, specified as `Developers = devnames`

A string included in the output of `prog::test`, denoting the developers deemed responsible for the code tested. This is intended for post-processing tools.

`BugId`

Option, specified as `BugId = bugid`

Again, for the output of `prog::test`, include a reference to some bug tracking system. bugid can be any MuPAD object.

Return Values

`prog::test` returns the void object `null()`.

See Also

MuPAD Functions
`prog::testexit` | `prog::testinit` | `traperror`

prog::testexit

Closing tests

Syntax

```
prog::testexit()
```

Description

`prog::testexit` closes automatic tests from test files and prints a short statistic about the test (see `prog::test`).

`prog::testexit` closes the last opened protocol file.

Note: `prog::testexit` must be called before beginning of a new test with `prog::testinit`.

Return Values

`prog::testexit` returns the void object `null()` and closes the last opened protocol file.

See Also

MuPAD Functions

`prog::test` | `prog::testinit`

prog::testinit

Initialize tests

Syntax

```
prog::testinit()

prog::testinit(string)

prog::testinit(expected_time, <All>)

prog::testinit(string, expected_time, <All>)

prog::testinit(arch = expected_time, …, <All>)

prog::testinit(string, arch = expected_time, …, <All>)
```

Description

The function `prog::testinit` initializes automatic tests (see `prog::test`).

The second argument `expected_time` determines the time, that the test should need.

Note: This time is not the real time, but a time factor that is given by `prog::testexit` at the end of a complete test.

This time factor is computed to be independent of the real speed of the used machine.

In general (without option `All`) the base for the time factor is the sum of all times for the evaluation of the first arguments of each call of `prog::test`.

The time factor can be useful to detect differences of the run time of tests, e.g., when the system or programs where changed.

For tests which run time depends on the architecture of the computer, the expected test time factor can be given apart for each test system as equation `arch = time_factor`.

The string `arch` must be one of the results returned by the function `sysname`. `time_factor` is the time factor given by `prog::testexit` at the end of the complete test on the reference system.

Examples

Example 1

Initialize a test that should need a run time factor of `2.0`:

```
prog::testinit("test1", 2.0):
```

Initialize a test that should need a run time factor of `2.8`, where the time is measured between `prog::testinit` and `prog::testexit`:

```
prog::testinit("test2", 2.8, All):
```

Initialize a test that should need a run time factor of `12.0` on Linux and `15.5` on Windows:

```
prog::testinit("test3", "glnxa64" = 12.0, "win32" = 15.5):
```

Parameters

`string`

String: a test name

`expected_time`

Expected test time factor (see below) in seconds as floating point number

`arch`

The computer architecture name as string (see `sysname`)

Options

`All`

The base for the time factor is the whole time between the command `prog::testinit` and `prog::testexit`.

Return Values

`prog::testinit` returns the void object `null()`.

See Also

MuPAD Functions
`prog::test` | `prog::testexit`

prog::trace

Observe functions

Syntax

```
prog::trace(obj, <Recursive = FALSE>)
```

```
prog::trace({obj1, obj2, …}, <Recursive = FALSE>)
```

```
prog::trace(options)
```

Description

`prog::trace(obj)` manipulates the MuPAD object `obj` to observe entering and exiting this object.

Note: `prog::trace` has a new syntax and a new set of options. The old syntax has been removed.

`prog::trace` lets you observe functions, domains, domain methods, and function environments. Use the `prog::trace` function for debugging. See "Example 1" on page 24-84.

`prog::trace` lets you specify a set of functions, domains, methods, or function environments that you want to observe. See "Example 2" on page 27-77.

`prog::trace` lets you observe the relations between calls to the traced objects.

To trace the object `obj`, use the function call `prog::trace(obj)`. After that call, every time the function call enters or exits the object `obj`, MuPAD prints a message and returns the arguments and the return value of that call. See "Example 3" on page 27-78.

`prog::trace` lets you observe a domain or a function environment. When you call the `prog::trace` function for a domain, the function observes all methods of the domain. When you call `prog::trace` for a function environment, it observes all slots of the

function environment. To trace only particular methods (slots), provide a set of these methods (slots). See the `slot` help page for more details. See “Example 8” on page 27-82.

To prevent tracing of all slots of a function environment, set the value of the `Recursive` option to `FALSE`. See “Example 7” on page 27-81.

The function `prog::untrace(obj)` terminates tracing of an object `obj`. Here `obj` is a function, a set of functions, a domain, a domain method, or a function environment. The function `prog::traced` detects whether the system currently traces a particular object.

Examples

Example 1

Define a function `f`, and observe this function:

```
f := x -> if x > 0 then x else -f(-x) end:
prog::trace(f):
f(-2)
```

```
enter f(-2)
  enter f(2)
  computed 2
computed -2
```

$$-2$$

Change the function, and reassign the new function to `f`. Although you use the same function name (`f`), MuPAD does not trace the new function:

```
f := x -> if x > 0 then x else f(-x) end:
f(-2)
```

$$2$$

To trace the new function, call `prog::trace` again. Now, the trace mechanism observes the updated function:

```
prog::trace(f):
```

```
f(-2)
```

```
enter f(-2)
  enter f(2)
  computed 2
computed 2
```

$$2$$

For further computations, stop observation of the function:

```
prog::untrace(f)
```

Example 2

If you want to trace more than one function, use a set to specify these functions in one function call:

```
prog::trace({sin, cos, exp}):
sin(5*PI/2);
cos(5*PI);
exp(5)
```

```
enter sin((5*PI)/2)
  enter sin(PI/2)
  remembered 1
computed 1
```

$$1$$

```
enter cos(5*PI)
  enter cos(PI)
  remembered -1
computed -1
```

$$-1$$

```
enter exp(5)
computed exp(5)
```

$$e^5$$

To stop observation of all functions, use `prog::untrace` without arguments:

```
prog::untrace()
```

Example 3

Define a short function that calls itself recursively, and observe the calls:

```
fib:= proc(n)
      begin
        if n < 2 then
          n
        else
          fib(n - 1) + fib(n - 2)
        end_if
      end_proc:
prog::trace(fib):
fib(3)
```

```
enter fib(3)
  enter fib(2)
    enter fib(1)
    computed 1
    enter fib(0)
    computed 0
  computed 1
  enter fib(1)
  computed 1
computed 2
```

$$2$$

To limit the number of the nested function calls displayed by `prog::trace`, use the `Depth` option. To specify the value of `Depth`, use a separate `prog::trace` function call:

```
prog::trace(fib):
```

```
prog::trace(Depth = 2);
fib(12)
```

```
enter fib(12)
  enter fib(11)
  computed 89
  enter fib(10)
  computed 55
computed 144
```

```
     144
```

The `Depth` option is independent of the `fib` procedure. Now, if you use `prog::trace` to trace any other procedure, `prog::trace` displays the nested calls to that procedure using `Depth = 2`. Remove this global option for further computations:

```
prog::untrace(fib):
prog::trace(Depth = 0)
```

Example 4

To display memory usage, use the `Mem` option:

```
prog::trace(Mem):
prog::trace(sin):
sin(3/2*PI)
```

```
enter
remember::sin((3*PI)/2) [mem: 3267052]   enter remember::sin(PI/2)
[mem: 4033596]   remembered 1 [mem: 4033436] computed -1 [mem: 4033072]
   -1
```

The `Mem` option is independent of the traced procedure. Now, if you use `prog::trace` to trace any other procedure, `prog::trace` displays memory usage in every step of that procedure. Remove this global option for further computations::

```
prog::untrace(sin):
prog::trace(Mem = FALSE)
```

Example 5

The `NoArgs` option suppresses the output of arguments of traced objects:

```
prog::trace(linalg):
prog::trace(NoArgs);
linalg::eigenvalues(matrix([[1,  0, 0],
                            [0, -1, 2],
                            [0,  1, 1]]))

enter linalg::eigenvalues
  enter linalg::checkForFloats
  return
  enter linalg::charpoly
    enter linalg::charpolyBerkowitz
    return
  return
return
```

$$\{1, \sqrt{3}, -\sqrt{3}\}$$

The `NoArgs` option is independent of the traced procedure. Now, if you use `prog::trace` to trace any other procedure, `prog::trace` hides arguments in every step of that procedure. Remove this global option for further computations:

```
prog::untrace(linalg):
prog::trace(NoArgs = FALSE)
```

Example 6

If you use the `Parent` option, `prog::trace` shows the name of the procedure that calls the traced object:

```
prog::trace(cos):
prog::trace(Parent):
f := x -> cos(2*x):
g := (x, y) -> f(x) + f(y):
g(3/2*PI, -3/2*PI)

enter cos(3*PI) (called from f)
  enter cos(PI) (called from cos)
  remembered -1
computed -1
enter cos(-3*PI) (called from f)
  enter cos(3*PI) (called from cos)
  remembered -1
computed -1
```

```
    -2
```

```
prog::trace(f):
prog::trace(g):
g(-PI, PI)

enter g(-PI, PI)
  enter f(-PI) (called from g)
    enter cos(-2*PI) (called from f)
      enter cos(2*PI) (called from cos)
        enter cos(0) (called from cos)
        remembered 1
      computed 1
    computed 1
  computed 1
  enter f(PI) (called from g)
    enter cos(2*PI) (called from f)
    remembered 1
  computed 1
computed 2
```

```
    2
```

The `Parent` option is independent of the traced procedures. Now, if you use `prog::trace` to trace any other object, `prog::trace` shows relations between calls to the traced objects. Remove this global option for further computations:

```
prog::untrace(cos):
prog::trace(Parent = FALSE)
```

Example 7

By default, the `prog::trace` function traces all slots of a function environment. For example, trace the `besselJ` function and observe the following function call:

```
prog::trace(besselJ);
besselJ(1, 2.3)

enter besselJ(1, 2.3)
  enter besselJ::float(1, 2.3)
  computed 0.5398725326
computed 0.5398725326
```

$$0.5398725326$$

To omit tracing of all slots, set the value of the `Recursive` option to `FALSE`:

```
prog::untrace(besselJ);
prog::trace(besselJ, Recursive=FALSE);
besselJ(1, 4.5)
```

```
enter besselJ(1, 4.5)
computed -0.2310604319
```

$$-0.2310604319$$

For further computations, stop observation of the `besselJ` function:

```
prog::untrace(besselJ)
```

Example 8

You can trace domains and domain methods. For example, create the following small domain:

```
T := newDomain("T"):
T::new := proc(h, m = 0) name T; begin new(T, h*60 + m) end:
T::intern := x -> [op(x) div 60, op(x) mod 60]:
T::print := x -> expr2text(T::intern(x)[1]).":".
                 substring(expr2text(100 + T::intern(x)[2]), 2, 2):
T::_plus := () -> new(T, _plus(map(args(), op))):
T::expr := op:
T::_mult := () -> new(T, _mult(map(args(), expr))):
prog::trace(T):
T(1, 30) + T(0, 45)*T(1, 05)
```

```
enter T(1, 30)
computed 1:30
enter T(0, 45)
computed 0:45
enter T(1, 5)
computed 1:05
enter T::_mult(0:45, 1:05)
computed 48:45
enter T::_plus(1:30, 48:45)
```

```
computed 50:15
```

```
     50:15
```

MuPAD does not trace the process of displaying traced outputs. Therefore, the `T::intern` and `T::print` methods do not appear in the traced outputs.

Now, trace the arithmetic methods only. When specifying the methods to trace, use their slot names, such as `slot(T, "_plus")` or `T::_plus`:

```
prog::untrace():
prog::trace({T::_plus, T::_mult}):
T(1, 30) + T(0, 45)*T(1, 05)
```

```
enter T::_mult(0:45, 1:05)
computed 48:45
enter T::_plus(1:30, 48:45)
computed 50:15
```

```
     50:15
```

```
prog::untrace():
```

Parameters

`obj`

A MuPAD function, a domain, a method, or a function environment to observe. Specify methods by their `slot` names (strings).

`{obj`$_1$`, obj`$_2$`, …}`

A set of MuPAD functions, domains, methods, or function environments to observe.

Options

`Depth`

Option, specified as `Depth = level`

Display nested function calls only up to the recursion depth `level`. Here `level` is a positive integer. After you set this option, all new and regenerated outputs for traced objects show the nested function calls only up to the specified recursion depth. See “Example 3” on page 27-78.

`Mem`

Show the current memory usage. After you set this option, all new and regenerated outputs for traced objects show the information about the current memory usage. See “Example 4” on page 27-79.

`NoArgs`

Do not show the arguments of calls to traced objects and the returned values. Without this option, all outputs for traced objects show the arguments and returned values for each call of a traced object. See “Example 5” on page 27-79.

`Parent`

Show the name of the procedure that calls the traced object. After you set this option, all new and regenerated outputs for traced objects show the names of the procedures that call the traced objects. See “Example 6” on page 27-80.

`Recursive`

Option, specified as `Recursive = FALSE`

Do not trace all slots of a function environment or domain. By default, `Recursive = TRUE`. See “Example 7” on page 27-81.

Return Values

`prog::trace` returns the void object `null()`.

See Also

MuPAD Functions

`debug` | `prog::profile` | `prog::traced` | `prog::untrace`

prog::traced

Find traced functions

Syntax

```
prog::traced(<obj>)
```

Description

`prog::traced()` lists all traced functions.

`prog::traced(obj)` detects, whether the function `obj` is traced. If `obj` is a library or a function environment, then all methods will be checked. If no argument is given, all traced functions will be displayed.

`prog::traced` determines whether a copy exists and whether the function has been manipulated the way `prog::trace` does.

Examples

Example 1

The `sin` function is traced:

```
prog::trace(sin):
prog::traced(sin)
```

```
    TRUE
```

The `cos` function is not traced:

```
prog::traced(cos)
```

```
    FALSE
```

Parameters

obj

A MuPAD function, a function environment or a library

Return Values

`prog::traced` returns the void object `null()`.

See Also

MuPAD Functions
`prog::trace` | `prog::untrace`

prog::untrace

Terminates observation of functions

Syntax

```
prog::untrace(obj)
```

```
prog::untrace()
```

Description

`prog::untrace(obj)` terminates the observation of the MuPAD function `obj` performed by `prog::trace`.

`obj` can be a domain or a function environment, too. Then all methods of the domain or function environment will be restored.

If no argument is given, all observed objects will be restored from observation.

Examples

Example 1

The observation of a function will be terminated:

```
prog::trace(sin):
sin(2)
```

```
enter sin(2)
computed sin(2)
```

$$\sin(2)$$

```
prog::untrace(sin):
sin(2)
```

$$\sin(2)$$

Parameters

obj

The MuPAD function that is observed, or a domain or a function environment

Return Values

`prog::untrace` returns the void object `null()`.

See Also

MuPAD Functions

`debug` | `prog::profile` | `prog::trace`

prog::wait

Wait for a while

Syntax

```
prog::wait(m)
```

```
prog::wait(s)
```

Description

`prog::wait(m)` waits for `m` milli seconds.

`prog::wait(s)` waits for `s` seconds.

`prog::wait` uses the function `rtime` for time measurement.

Examples

Example 1

Wait for 3 seconds:

```
prog::wait(3000)
```

Wait for 3 seconds again:

```
prog::wait(3.0)
```

The next example shows the difference between the system time and the CPU time used by MuPAD:

```
time(prog::wait(2.5))
```

```
2040
```

In `2.5` seconds realtime the MuPAD process runs nearly two seconds.

Example 2

Use `traperror` to limit the evaluation time:

```
traperror(prog::wait(100.0), 5): lasterror()

Error: Execution time is exceeded.
```

Parameters

`m`

Milli seconds to wait as positive integer

`s`

Seconds to wait as positive floating-point number

Return Values

`prog::wait` returns the empty object `null()`.

See Also

MuPAD Functions
`rtime` | `time` | `traperror`

property – Properties and Assumptions

property::depends
property::hasprop
property::showprops

property::depends

Dependence table with all properties of an expression

Syntax

```
property::depends(ex, …)
```

Description

`property::depends(...)` returns a table which contains all information about the properties of the whole input. This table can be used to determine any change of the properties of an expression.

The returned table is mainly used inside the MuPAD library, to ensure the validity of remembered results of the remember mechanism.

A MuPAD expression can have different properties at different times, without changing its value.

Note: The kernel remember mechanism cannot determine the change of the properties and returns wrong results, when the result depends on the properties of the input.

In this case the extended remember mechanism provided by `prog::remember` should be used together with `property::depends`.

Examples

Example 1

A compare of the dependence table at different times detects changes of the properties of an expression.

The first call to the defined function `has_changed` initializes the table `DEPENDS` that keeps the dependence information:

```
DEPENDS := table():
has_changed :=
  proc(ex)
  begin
    if not contains(DEPENDS, ex)
       or property::depends(ex) <> DEPENDS[ex] then
      DEPENDS[ex] := property::depends(ex);
      TRUE
    else
      FALSE
    end_if
  end_proc:
has_changed(sin(x*PI)):
```

The properties has not changed:

```
has_changed(sin(x*PI))
```

```
    FALSE
```

Every change is detected:

```
assume(x, Type::Integer):
has_changed(sin(x*PI))
```

```
    TRUE
```

```
assume(x, Type::PosInt):
has_changed(sin(x*PI))
```

```
    TRUE
```

```
assume(x >= 0):
has_changed(sin(x*PI))
```

```
    TRUE
```

```
unassume(x):
has_changed(sin(x*PI))
```

```
    TRUE
```

```
delete DEPENDS, has_changed:
```

Example 2

The next example shows the problems with the kernel remember mechanism:

```
pos := proc(x)
         option remember;
       begin
         is(x > 0)
       end:
pos(x)
```

```
    UNKNOWN
```

The result UNKNOWN was stored for the input x and is returned, although the properties of x are changed:

```
assume(x > 0): pos(x);
assume(x < 0): pos(x)
```

```
    UNKNOWN
```

```
    UNKNOWN
```

This problem can only be solved by the extended remember mechanism together with `property::depends` (x still is less than zero):

```
pos := proc(x)
       begin
         is(x > 0)
       end:
pos := prog::remember(pos, property::depends):
pos(x)
```

```
    FALSE
```

After changing the properties of the input, the defined function recomputes the result:

```
assume(x > 0): pos(x);
```

```
unassume(x): pos(x)
```

```
TRUE
```

```
UNKNOWN
```

Parameters

ex

Any MuPAD expression

Return Values

Table that can be compared with another dependence table

See Also

MuPAD Functions

`assume` | `getprop` | `is` | `prog::remember`

property::hasprop

Does an object have properties?

Syntax

```
property::hasprop(object)
```

```
property::hasprop()
```

Description

`property::hasprop(object)` tests, whether the object has properties and returns `TRUE` if the object or any subexpression has a property, otherwise `FALSE`.

Compared with `getprop`, `property::hasprop` is a fast function and can be used to determine, whether an object has properties without using the slower functions `getprop` or `is`.

Note: In some cases, the function `is` can derive some aspects without any defined property (see “Example 2” on page 28-7)!

Examples

Example 1

Does the expression `2*(x+1)` have any properties?

```
property::hasprop(2*(x + 1))
```

```
    FALSE
```

```
assume(x > 0):
property::hasprop(2*(x + 1))
```

```
    TRUE
```

```
getprop(2*(x + 1))
```

$$(2, \infty)$$

```
delete x:
```

Example 2

`property::hasprop` returns `FALSE`, but `is` can determine an answer unequal to `UNKNOWN`:

```
property::hasprop(0 < x/(x + y) + y/(x + y))
```

```
    FALSE
```

```
is(exp(x) = 0)
```

```
    FALSE
```

Parameters

`object`

Any MuPAD object

Return Values

`TRUE` or `FALSE`

See Also

MuPAD Functions

`assume` | `getprop` | `indets` | `is` | `unassume`

property::showprops

What assumptions are made?

Syntax

```
property::showprops(object)
```

Description

`property::showprops(object)` shows all assumptions set for identifiers in `object`. If no assumptions were set, the empty list ist returned.

Examples

Example 1

```
assume(x > 0);
property::showprops(x);
```

$$[0 < x]$$

```
assumeAlso(x < 1):
property::showprops(x);
```

$$[x < 1, 0 < x]$$

```
delete x:
```

Parameters

`object`

Any MuPAD object

Return Values

List containing all assumptions.

See Also

MuPAD Functions
`assume` | `getprop` | `unassume`

solvelib – Datatypes and Utilities for the Solver

solvelib::BasicSet
C_
R_
Q_
Z_
N_
solvelib::cartesianPower
solvelib::cartesianProduct
solvelib::conditionalSort
solvelib::getElement
solvelib::isEmpty
solvelib::isFinite
solvelib::pdioe
solvelib::preImage
solvelib::splitVectorSet
solvelib::Union
solvelib::VectorImageSet

solvelib::BasicSet

Basic infinite sets

Syntax

Domain Creation

```
solvelib::BasicSet(Dom::Integer)
```

```
solvelib::BasicSet(Dom::Rational)
```

```
solvelib::BasicSet(Dom::Real)
```

```
solvelib::BasicSet(Dom::Complex)
```

Description

The domain `solvelib::BasicSet` comprises the four sets of integers, reals, rationals, and complex numbers, respectively.

The four basic sets are assigned to the identifiers `Z_`, `Q_`, `R_`, and `C_` during system initialization.

The set of positive integers, too, is assigned to the identifier `N_` during system initialization. It is not represented by a basic set but by the intersection of `Z_` and the interval `Dom::Interval([1], infinity)`.

Superdomain

`Dom::BaseDomain`

Axioms

`Ax::canonicalRep`

Categories

`Cat::Set`

Examples

Example 1

The domain of basic sets know about the basic arithmetical and set-theoretic functions:

```
J:=Dom::Interval(3/2, 21/4):
Z_ intersect J
```

$$\{2, 3, 4, 5\}$$

Methods

Mathematical Methods

`contains` — Test whether some object is a member

`contains(a, S)`

Equivalently, `is(a in S)` may be used.

Conversion Methods

`convert` — Convert a domain into a basic set

`convert(d)`

`set2prop` — Convert a set to a property

`set2prop(S)`

See Also

MuPAD Domains
`C_ | Dom::Interval | N_ | Q_ | R_ | Z_`

C_

Complex numbers

Description

`C_`, or equivalently `solvelib::BasicSet(Dom::Complex)`, represents the set of complex numbers.

The four basic sets are assigned to the identifiers `Z_`, `Q_`, `R_`, and `C_` during system initialization.

Superdomain

`Dom::BaseDomain`

Axioms

`Ax::canonicalRep`

Categories

`Cat::Set`

Methods

Mathematical Methods

`contains` — Test whether some object is a member

```
contains(a, S)
```

Equivalently, `is(a in S)` may be used.

Conversion Methods

`convert` — Convert a domain into a basic set

```
convert(d)
```

`set2prop` — Convert a set to a property

```
set2prop(S)
```

See Also

MuPAD Domains

`Dom::Interval` | `N_` | `Q_` | `R_` | `solvelib::BasicSet` | `Z_`

R_

Real numbers

Description

`R_`, or equivalently `solvelib::BasicSet(Dom::Real)`, represents the set of real numbers.

The four basic sets are assigned to the identifiers `Z_`, `Q_`, `R_`, and `C_` during system initialization.

Superdomain

`Dom::BaseDomain`

Axioms

`Ax::canonicalRep`

Categories

`Cat::Set`

Methods

Mathematical Methods

`contains` — Test whether some object is a member

```
contains(a, S)
```

Equivalently, `is(a in S)` may be used.

Conversion Methods

`convert` — Convert a domain into a basic set

```
convert(d)
```

`set2prop` — Convert a set to a property

```
set2prop(S)
```

See Also

MuPAD Domains

`C_` | `Dom::Interval` | `N_` | `Q_` | `solvelib::BasicSet` | `Z_`

Q_

Rational numbers

Description

`Q_`, or equivalently `solvelib::BasicSet(Dom::Rational)`, represents the set of rational numbers.

The four basic sets are assigned to the identifiers `Z_`, `Q_`, `R_`, and `C_` during system initialization.

Superdomain

`Dom::BaseDomain`

Axioms

`Ax::canonicalRep`

Categories

`Cat::Set`

Methods

Mathematical Methods

`contains` — Test whether some object is a member

`contains(a, S)`

Equivalently, `is(a in S)` may be used.

Conversion Methods

`convert` — Convert a domain into a basic set

```
convert(d)
```

`set2prop` — Convert a set to a property

```
set2prop(S)
```

See Also

MuPAD Domains

`C_` | `Dom::Interval` | `N_` | `R_` | `solvelib::BasicSet` | `Z_`

Z_

Integers

Description

`Z_`, or equivalently `solvelib::BasicSet(Dom::Integer)`, represents the set of integers.

The four basic sets are assigned to the identifiers `Z_`, `Q_`, `R_`, and `C_` during system initialization.

The set of positive integers, too, is assigned to the identifier `N_` during system initialization. It is not represented by a basic set but by the intersection of `Z_` and the interval `Dom::Interval([1], infinity)`.

Superdomain

`Dom::BaseDomain`

Axioms

`Ax::canonicalRep`

Categories

`Cat::Set`

Examples

Example 1

The domain of basic sets know about the basic arithmetical and set-theoretic functions:

```
J:=Dom::Interval(3/2, 21/4):
Z_ intersect J
```

$$\{2, 3, 4, 5\}$$

Methods

Mathematical Methods

`contains` — Test whether some object is a member

```
contains(a, S)
```

Equivalently, `is(a in S)` may be used.

Conversion Methods

`convert` — Convert a domain into a basic set

```
convert(d)
```

`set2prop` — Convert a set to a property

```
set2prop(S)
```

See Also

MuPAD Domains
`C_` | `Dom::Interval` | `N_` | `Q_` | `R_` | `solvelib::BasicSet`

N_

Positive integers

Description

`N_` represents the set of positive integers.

The four basic sets are assigned to the identifiers `Z_`, `Q_`, `R_`, and `C_` during system initialization.

The set of positive integers, too, is assigned to the identifier `N_` during system initialization. It is not represented by a basic set but by the intersection of `Z_` and the interval `Dom::Interval([1], infinity)`.

Superdomain

```
Dom::BaseDomain
```

Axioms

```
Ax::canonicalRep
```

Categories

```
Cat::Set
```

Methods

Mathematical Methods

`contains` — Test whether some object is a member

```
contains(a, S)
```

Equivalently, `is(a in S)` may be used.

Conversion Methods

`convert` — Convert a domain into a basic set

```
convert(d)
```

`set2prop` — Convert a set to a property

```
set2prop(S)
```

See Also

MuPAD Domains

`C_` | `Dom::Interval` | `Q_` | `R_` | `solvelib::BasicSet` | `Z_`

solvelib::cartesianPower

Cartesian power of a set

Syntax

Domain Creation

```
solvelib::cartesianPower()
```

```
solvelib::cartesianPower(S, n)
```

Description

`solvelib::cartesianPower` is the domain of all cartesian powers of subsets of the complex numbers.

`solvelib::cartesianPower(S, n)` returns the set of all n-tuples of elements of `S`.

`solvelib::cartesianPower(S, n)` returns the `n`-fold cartesian product of `S` with itself, that is, the set of all vectors of length `n` whose components are elements of `S`.

`S` must represent a subset of the complex numbers; see `solve` for an overview of the different kinds of sets in MuPAD.

The set of one-tuples of elements of `S` consists of vectors and therefore differs from the set `S` in the same way as matrices of type `matrix` with one row and one column are different from numbers.

Superdomain

`Dom::BaseDomain`

Categories

`Cat::Set`

Examples

Example 1

A cartesian power of a finite set of numbers is a finite set of vectors:

```
A:= solvelib::cartesianPower({1, 2, I}, 3)
```

$$\left\{\begin{pmatrix}1\\1\\1\end{pmatrix},\begin{pmatrix}2\\1\\1\end{pmatrix},\begin{pmatrix}1\\2\\1\end{pmatrix},\begin{pmatrix}1\\1\\2\end{pmatrix},\begin{pmatrix}2\\2\\1\end{pmatrix},\begin{pmatrix}2\\1\\2\end{pmatrix},\begin{pmatrix}1\\2\\2\end{pmatrix},\begin{pmatrix}2\\2\\2\end{pmatrix},\begin{pmatrix}i\\1\\1\end{pmatrix},\begin{pmatrix}1\\i\\1\end{pmatrix},\begin{pmatrix}1\\1\\i\end{pmatrix},\begin{pmatrix}i\\2\\1\end{pmatrix},\begin{pmatrix}2\\i\\1\end{pmatrix},\begin{pmatrix}i\\1\\2\end{pmatrix},\begin{pmatrix}1\\i\\2\end{pmatrix},\begin{pmatrix}2\\1\\i\end{pmatrix},\begin{pmatrix}1\\2\\i\end{pmatrix},\begin{pmatrix}i\\2\\2\end{pmatrix},\begin{pmatrix}2\\i\\2\end{pmatrix},\begin{pmatrix}2\\2\\i\end{pmatrix},\begin{pmatrix}i\\i\\1\end{pmatrix},\begin{pmatrix}i\\1\\i\end{pmatrix},\begin{pmatrix}1\\i\\i\end{pmatrix},\begin{pmatrix}i\\i\\2\end{pmatrix},\begin{pmatrix}i\\2\\i\end{pmatrix},\begin{pmatrix}2\\i\\i\end{pmatrix},\begin{pmatrix}i\\i\\i\end{pmatrix}\right\}$$

We can select those vectors with all components real as follows:

```
A intersect solvelib::cartesianPower(R_, 3)
```

$$\left\{\begin{pmatrix}1\\1\\1\end{pmatrix},\begin{pmatrix}2\\1\\1\end{pmatrix},\begin{pmatrix}1\\2\\1\end{pmatrix},\begin{pmatrix}1\\1\\2\end{pmatrix},\begin{pmatrix}2\\2\\1\end{pmatrix},\begin{pmatrix}2\\1\\2\end{pmatrix},\begin{pmatrix}1\\2\\2\end{pmatrix},\begin{pmatrix}2\\2\\2\end{pmatrix}\right\}$$

Example 2

Cartesian powers of the set of complex numbers may occur as the result of a call to `solve` if every n-tuple of complex numbers is a solution of the given system:

```
solve([x+y = x+y], [x, y], VectorFormat)
```

$$\mathbb{C}^2$$

Parameters

S

Set

n

Positive integer

Methods

Access Methods

`base` — Set S

```
base(A)
```

`dimension` — Exponent n

```
dimension(A)
```

Technical Methods

`print` — Print a cartesian power

```
print(A)
```

solvelib::cartesianProduct

Cartesian product of sets

Syntax

```
solvelib::cartesianProduct(S, …)
```

Description

`solvelib::cartesianProduct(S,...)` returns the cartesian product of its arguments.

The arguments may be sets of any type, consisting of complex numbers; the result is a set that consists of vectors, or a symbolic call to `solvelib::cartesianProduct`. See `solve` for an overview of the different kinds of sets in MuPAD.

Examples

Example 1

For finite sets, the result is similar to that of `combinat::cartesianProduct` but consists of vectors and not of lists:

```
S:= solvelib::cartesianProduct({1, 2}, {3, 4})
```

$$\left\{\begin{pmatrix}1\\3\end{pmatrix}, \begin{pmatrix}2\\3\end{pmatrix}, \begin{pmatrix}1\\4\end{pmatrix}, \begin{pmatrix}2\\4\end{pmatrix}\right\}$$

```
solvelib::splitVectorSet(S)
```

$$[\{1, 2\}, \{3, 4\}]$$

```
delete S:
```

Example 2

For infinite sets, results of various types are possible, e.g., `solvelib::VectorImageSet` or `solvelib::cartesianPower`:

```
solvelib::cartesianProduct(R_, R_)
```

$$\mathbb{R}^2$$

```
solvelib::cartesianProduct(PI*Z_, Z_)
```

$$\left\{ \begin{pmatrix} \pi k \\ l \end{pmatrix} \middle| \, k \in \mathbb{Z}, l \in \mathbb{Z} \right\}$$

Parameters

S

Set of complex numbers

Return Values

Set

See Also

MuPAD Functions

`combinat::cartesianProduct` | `solve` | `solvelib::cartesianPower` | `solvelib::splitVectorSet`

solvelib::conditionalSort

Possible sortings of a list depending on parameters

Syntax

```
solvelib::conditionalSort(l)
```

Description

`solvelib::conditionalSort(l)` sorts the list `l` in ascending order. Unlike for `sort`, only the usual order on the real numbers and not the internal order (see `sysorder`) is used. `solvelib::conditionalSort` does a case analysis if list elements contain indeterminates.

`solvelib::conditionalSort` invokes the inequality solver to get simple conditions in the case analysis. The ability of `solvelib::conditionalSort` to recognize sortings as impossible is thus limited by the ability of the inequality solver to recognize an inequality as unsolvable. See “Example 3” on page 29-21.

Only expressions representing real numbers can be sorted. It is an error if non-real numbers occur in the list; it is implicitly assumed that all parameters take on only such values that cause all list elements to be real.

Sorting is unstable, i.e. equal elements may be placed in any order in the resulting list; these cases may be listed separately in the case analysis.

The usual simplifications for piecewise defined objects are applied, e.g., equalities that can be derived from a condition are applied (by substitution) to the list.

Environment Interactions

`solvelib::conditionalSort` takes into account the assumptions on all occurring identifiers.

Examples

Example 1

In the simplest case, sorting a two-element list `[a,b]` just amounts to solving the inequality `a<=b` w.r.t. all occurring parameters.

```
solvelib::conditionalSort([x,x^2])
```

$$\begin{cases} [x, x^2] & \text{if } 0 < x\,(x-1) \\ [x^2, x] & \text{if } x\,(x-1) \le 0 \end{cases}$$

Example 2

If, by implicit or explicit assumptions on the parameters, no different sortings can occur, the result is just a list.

According to the implicit assumption that all list elements are real, `x` must be nonnegative.

```
solvelib::conditionalSort([sqrt(x), -3])
```

$$[-3, \sqrt{x}]$$

Example 3

Sometimes cases are not recognized as impossible.

```
assume(x>5): solvelib::conditionalSort([x,gamma(x)])
```

$$\begin{cases} [x, \Gamma(x)] & \text{if } x < \Gamma(x) \\ [\Gamma(x), x] & \text{if } \Gamma(x) \le x \end{cases}$$

Parameters

`1`

List of arithmetical expressions

Return Values

List if the sorting is the same for all possible parameter values; or an object of type `piecewise` if some case analysis is necessary.

Algorithms

The complexity of sorting a list of n elements is up to $n!$.

See Also

MuPAD Functions

`piecewise` | `sort`

solvelib::getElement

Get one element of a set

Syntax

```
solvelib::getElement(S, <Random>)
```

Description

`solvelib::getElement(S)` returns an element of `S`.

`S` can be a set of any type; see `solve` for an overview of all sets.

`solvelib::getElement` returns the value `FAIL` if:

- `S` is the empty set
- the solver cannot find any element of the set due to the solver's limitations
- the solver cannot compute the first element of a set. You can use the `Random` option to check a random element of a set instead of the first element
- the answer depends on a case analysis on some parameter

With the option `Random`, the probability to get any particular element of a set is:

- Roughly equal for the elements of a finite set
- Proportional to the multiplicities for the elements of a finite multiset of type `Dom::Multiset`
- Unspecified for the elements of an infinite set. Practically, the same result does not occur twice for infinite sets.

Examples

Example 1

If `S` is a finite set, the solver returns just one of the elements:

```
solvelib::getElement({2, 7, a})
```

$$2$$

Example 2

For image sets, the solver replaces every parameter by a constant:

```
S:=Dom::ImageSet(k*PI, k, solvelib::BasicSet(Dom::Integer))
```

$$\{\pi\, k \mid k \in \mathbb{Z}\}$$

```
solvelib::getElement(S)
```

$$0$$

Example 3

If the set is empty, the solver cannot find any element:

```
solvelib::getElement({})
```

$$\text{FAIL}$$

`solvelib::getElement` might fail to find an element of a set although that set is not empty.

```
solvelib::getElement(solve(exp(x) + cos(x) = x^2,x))
```

$$\text{FAIL}$$

Example 4

Without the option `Random`, `solvelib::getElement` always produces the same result for a set:

```
solvelib::getElement({$1..5}) $i=1..5
```

```
1, 1, 1, 1, 1
```

With the option `Random`, the returned element varies randomly from call to call:

```
solvelib::getElement({$1..5}, Random) $i=1..15
```

```
2, 1, 3, 4, 2, 4, 1, 4, 2, 5, 1, 4, 3, 1, 5
```

The distribution of the returned values is close to the uniform distribution. For multisets, the multiplicity of elements is taken into account:

```
solvelib::getElement(Dom::Multiset(1$4, 2$2), Random) $i=1..18
```

```
1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1, 1
```

Example 5

For the following set parametrized by integers, the solver fails to find an element. This is because the solver tries only the first parameter-value pair `k = 0` for which the result is undefined. After that the solver does not try any other parameter-value pairs:

```
solvelib::getElement(1/Z_)
```

```
FAIL
```

For the sets with the undefined first element, you can get the result calling the solver with the option `Random`:

```
solvelib::getElement(1/Z_, Random)
```

$$\frac{1}{902}$$

Parameters

S

Any set

Options

`Random`

Returns a random element of a set. Without this option, `solvelib::getElement(S)` always returns the same element.

Return Values

`solvelib::getElement` returns a MuPAD object representing an element of `S`, or `FAIL` if no element could be determined.

Overloaded By

S

See Also

MuPAD Functions
`solve`

solvelib::isEmpty

Predicate expressing the emptyness of a set

Syntax

```
solvelib::isEmpty(S)
```

Description

`solvelib::isEmpty(S)` returns a boolean expression that is equivalent to the statement that `S` is the empty set.

Since functions operating on boolean expressions like `assume`, `is`, or `solve` cannot handle equations involving sets, it is not possible to pass the expression `S={}` to them. `solvelib::isEmpty(S)` helps to get around this problem, as it tries to express the emptyness of `S` in an equivalent way that can be handled by the aforementioned functions. If no suitable equivalen expression is found, the unevaluated call to `solvelib::isEmpty` is returned.

`S` may be a set of any type; see `solve` for an enumeration of the various types of sets.

The `solvelib::isEmpty` function always returns Boolean expressions, even if the function cannot resolve an expression. See “Example 4” on page 29-28.

Examples

Example 1

The emptyness of a `DOM_SET` can be decided immediately:

```
solvelib::isEmpty({a, b}), solvelib::isEmpty({})
```

```
FALSE, TRUE
```

Example 2

The intersection of a finite set with any other set is empty if and only if none of the elements of the finite set is in the other set:

```
solvelib::isEmpty({a, b} intersect Z_)
```

$$a \notin \mathbb{Z} \wedge b \notin \mathbb{Z}$$

Example 3

The output of `solve` can be entered directly into `solvelib::isEmpty`:

```
solvelib::isEmpty(solve(a*x=b, x))
```

$$a = 0 \wedge b \neq 0$$

Example 4

Sometimes, no simpler equivalent expression can be found:

```
result := solvelib::isEmpty(solve(x^2 = sin(x), x))
```

$$\text{solvelib::isEmpty}(\text{solve}(x^2 - \sin(x) = 0, x))$$

The returned expression is a Boolean expression:

```
testtype(result, Type::Boolean)
```

TRUE

Parameters

s

Any set

Return Values

Boolean expression

Overloaded By

S

See Also

MuPAD Functions
`assume` | `is` | `solve`

solvelib::isFinite

Test whether a set is finite

Syntax

```
solvelib::isFinite(S)
```

Description

`solvelib::isFinite(S)` returns `TRUE`, `FALSE`, or `UNKNOWN` depending on whether `S` is finite, infinite, or the question could not be settled.

`S` may be a set of any type; see `solve` for an enumeration of the various types of sets.

Examples

Example 1

A `DOM_SET` is always finite:

```
solvelib::isFinite({2,5})
```

```
    TRUE
```

Example 2

The set of integers is infinite.

```
solvelib::isFinite(Z_)
```

```
    FALSE
```

Parameters

S

Any set

Return Values

Boolean value

Overloaded By

S

See Also

MuPAD Functions

`solve`

solvelib::pdioe

Solve polynomial Diophantine equations

Syntax

```
solvelib::pdioe(a, b, c)
```

```
solvelib::pdioe(aexpr, bexpr, cexpr, x)
```

Description

`solvelib::pdioe(a, b, c)` returns polynomials `u` and v that satisfy the equation `au + bv = c`.

`solvelib::pdioe(aexpr, bexpr, cexpr, x)` does the same after converting the arguments into univariate polynomials `a, b, c` in the variable `x`.

The coefficient ring of the polynomials `a`, `b`, and `c` must be either `Expr`, or `IntMod(p)` for some prime *p*, or a domain belonging to the category `Cat::Field`.

Examples

Example 1

If expressions are passed as arguments, a fourth argument must be provided:

```
solvelib::pdioe(x,
                13*x + 22*x^2 + 18*x^3 + 7*x^4 + x^5 + 3,
                x^2 + 1,
                x)
```

$$-\frac{x^4}{3}-\frac{7x^3}{3}-6x^2-\frac{19x}{3}-\frac{13}{3},\ \frac{1}{3}$$

Example 2

x is not a multiple of the gcd of $x + 1$ and $x^2 - 1$. Hence the equation $u(x + 1) + v(x^2 - 1) = x$ has no solution for u and v:

```
solvelib::pdioe(x + 1, x^2 - 1, x, x)
```

FAIL

Example 3

If the arguments are polynomials, the fourth argument may be omitted:

```
solvelib::pdioe(poly(a + 1, [a]),
                poly(a^2 + 1, [a]),
                poly(a - 1, [a]))
```

$\mathrm{poly}(a, [a]), \mathrm{poly}(-1, [a])$

Parameters

x

Identifier or indexed identifier

a, b, c

Univariate polynomials

aexpr, bexpr, cexpr

Polynomial expressions

Return Values

If the equation is solvable, `solvelib::pdioe` returns an expression sequence consisting of two operands of the same type as the input (expressions or polynomials). If the equation has no solution, `solvelib::pdioe` returns `FAIL`.

See Also

MuPAD Functions
`solve`

solvelib::preImage

Preimage of a set under a mapping

Syntax

```
solvelib::preImage(a, x, S)
```

Description

`solvelib::preImage(a, x, S)` returns the set of all numbers `y` such that substituting `y` for `x` in `a` gives an element of `S`.

`S` can be a set of any type (finite or infinite).

Examples

Example 1

In case of a finite set `S`, the preimage of S is just the union of all sets `solve(a=s, x)`, where `s` ranges over the elements of `S`.

```
solvelib::preImage(x^2+2, x, {11, 15});
```

$$\{-3, 3, \sqrt{13}, -\sqrt{13}\}$$

Note that computing this set may take a long time for large finite sets:

```
time(solvelib::preImage(x, x,
          Z_ intersect Dom::Interval(0, 1000000)))
```

$$14657$$

Example 2

For intervals, the preimage is usually an interval or a union of intervals.

```
solvelib::preImage(x^2+2, x, Dom::Interval(3..7));
```

$$(1, \sqrt{5}) \cup (-\sqrt{5}, -1)$$

Parameters

a

Arithmetic expression

x

Identifier

S

Set

Return Values

Set

See Also

MuPAD Functions
`solve`

solvelib::splitVectorSet

Factor a set of vectors into a cartesian product

Syntax

```
solvelib::splitVectorSet(S)
```

Description

`solvelib::splitVectorSet(S)` returns a list S_1, ..., S_n of sets of complex numbers such that `S` is the cartesian product of the S_i, or `FAIL` if such factorization could not be found.

The set `S` may be finite or infinite, of any type.

Examples

Example 1

We split a finite set of vectors into its factors:

```
solvelib::splitVectorSet({[1, 2], [1, 3], [0, 2], [0, 3]})
```

```
[{0, 1}, {2, 3}]
```

The following set cannot be written as a cartesian product:

```
solvelib::splitVectorSet({[1, 2], [0, 2], [0, 3]})
```

```
FAIL
```

Example 2

Infinite sets can also be handled:

```
S:= Dom::ImageSet([k*PI, l*PI+2], [k, l], [Z_, Z_])
```

$$\left\{ \begin{pmatrix} \pi\, k \\ \pi\, l + 2 \end{pmatrix} \;\middle|\; k \in \mathbb{Z}, l \in \mathbb{Z} \right\}$$

```
solvelib::splitVectorSet(S)
```

$$[\{\pi\, k \mid k \in \mathbb{Z}\}, \{\pi\, k + 2 \mid k \in \mathbb{Z}\}]$$

```
delete S:
```

Parameters

S

Set of vectors

Return Values

List of type `DOM_LIST`, or `FAIL`

See Also

MuPAD Functions

`solve`

solvelib::Union

Union of a system of sets

Syntax

```
solvelib::Union(set, param, paramset)
```

```
solvelib::Union(set, paramlist, vectorset)
```

Description

`solvelib::Union (set, paramlist, vectorset)` returns the set of all objects that can be obtained by replacing, in some element of `set`, the list of free parameters `paramlist` by an element of `vectorset`.

`set` may be a set of any type; it need not depend on the parameter `param`, and it may also contain other free parameters.

`paramset` may be a set of any type and may depend on some free parameters. See “Example 1” on page 29-39.

If `paramset` is empty, the result is the empty set. Overloading has no effect in this case.

`vectorset` may be a set of any type, consisting of vectors whose dimension equals the number of variables in `paramlist`.

Examples

Example 1

We compute the set of all numbers that are equal to $k + 1$ or $k + 3$ for $k = 2$, $k = 4$, or $k = l$, where l is a free parameter.

```
solvelib::Union({k+1, k+3}, k, {2,4,l});
```

$$\{3, 5, 7, l+1, l+3\}$$

Example 2

In the same way, we can let a pair of parameters range over a set of pairs:

```
solvelib::Union(Dom::ImageSet(PI*k + exp(x) + y, k, Z_),
                [x, y], {[3, 2], [1, 4]})
```

$$\{e+\pi k+4 \mid k \in \mathbb{Z}\} \cup \{e^3+\pi k+2 \mid k \in \mathbb{Z}\}$$

Parameters

`set`

Set of any type

`param`

Identifier

`paramset`

Set of any type

`paramlist`

List of identifiers

`vectorset`

Set of vectors

Return Values

`solvelib::Union` returns a set of any type; see `solve` for an overview of the different types of sets. It may also return the unevaluated call if the union could not be computed.

Overloaded By

set

See Also

MuPAD Functions
`solve`

solvelib::VectorImageSet

Domain of set of vectors that are images of sets under mappings

Syntax

Domain Creation

```
solvelib::VectorImageSet()
```

Element Creation

```
solvelib::VectorImageSet(v, x, S)
```

```
Dom::ImageSet(v, [x1, …], [S1, …])
```

Description

Domain Creation

`solvelib::VectorImageSet` is the domain of all sets of vectors of complex numbers that can be written as the set of all values taken on by some mapping, i.e., sets of the form $\{[f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)] \mid x_i \in S_i\}$ for some complex-valued functions f_j and some sets $S_1, \ldots, S_n$ of complex numbers.

Sets of this type are used by `solve` to express solutions of systems of equations like $\{[k\,\pi, z] \mid k \in \mathbb{Z}, z \in \mathbb{C}\}$.

Element Creation

`solvelib::VectorImageSet(v, x, S)` represents the set of all vectors that can be obtained by substituting some element of `S` for `x` in the vector `v`.

`solvelib::VectorImageSet(v, [x1, ...], [S1, ...])` represents the set of all values that can be obtained by substituting, for each i, the identifier x_i by some element of S_i in the vector `v`.

The user never needs to call `solvelib::VectorImageSet` directly. Instead, `Dom::ImageSet` should be used; it will automatically create a vector image set if its first argument is a list or matrix. The entries and methods of the domain `solvelib::VectorImageSet` are the same as those of `Dom::ImageSet`, and may be found on that help page.

Superdomain

`Dom::ImageSet`

Categories

`Cat::Set`

Examples

Example 1

We create a set of two-dimensional vectors:

```
S:= Dom::ImageSet([k*PI, k*I*PI], k, Z_)
```

$$\left\{ \begin{pmatrix} \pi\, k \\ \pi\, k\, \mathrm{i} \end{pmatrix} \;\middle|\; k \in \mathbb{Z} \right\}$$

Since this is a set of vectors, a `solvelib::VectorImageSet` is created automatically:

```
type(S)
```

DomImageSet

Set-theoretic operations (union, intersection, set difference) may be applied to `S`:

```
S intersect {[0, 0], [1, 1]}, S minus S
```

```
{[0, 0]}, ∅
```

```
delete S:
```

Parameters

v

List or `matrix`

x

Identifier or indexed identifier

S

Set of any type

stats – Statistics

stats::betaCDF
stats::betaPDF
stats::betaQuantile
stats::betaRandom
stats::binomialCDF
stats::binomialPF
stats::binomialQuantile
stats::binomialRandom
stats::calc
stats::cauchyCDF
stats::cauchyPDF
stats::cauchyQuantile
stats::cauchyRandom
stats::chisquareCDF
stats::chisquarePDF
stats::chisquareQuantile
stats::chisquareRandom
stats::col
stats::concatCol
stats::concatRow
stats::correlation
stats::correlationMatrix
stats::covariance
stats::cutoff
stats::winsorize
stats::csGOFT
stats::empiricalCDF
stats::empiricalPF
stats::empiricalQuantile
stats::empiricalRandom
stats::equiprobableCells
stats::erlangCDF

stats::erlangPDF
stats::erlangQuantile
stats::erlangRandom
stats::exponentialCDF
stats::exponentialPDF
stats::exponentialQuantile
stats::exponentialRandom
stats::fCDF
stats::fPDF
stats::fQuantile
stats::fRandom
stats::finiteCDF
stats::finitePF
stats::finiteQuantile
stats::finiteRandom
stats::frequency
stats::gammaCDF
stats::gammaPDF
stats::gammaQuantile
stats::gammaRandom
stats::geometricMean
stats::geometricCDF
stats::geometricPF
stats::geometricQuantile
stats::geometricRandom
stats::harmonicMean
stats::hodrickPrescottFilter
stats::hypergeometricCDF
stats::hypergeometricPF
stats::hypergeometricQuantile
stats::hypergeometricRandom
stats::ksGOFT
stats::kurtosis
stats::linReg
stats::logisticCDF
stats::logisticPDF
stats::logisticQuantile
stats::logisticRandom
stats::lognormalCDF
stats::lognormalPDF

stats::lognormalQuantile
stats::lognormalRandom
stats::mean
stats::meandev
stats::median
stats::modal
stats::moment
stats::normalCDF
stats::normalPDF
stats::normalQuantile
stats::normalRandom
stats::obliquity
stats::poissonCDF
stats::poissonPF
stats::poissonQuantile
stats::poissonRandom
stats::quadraticMean
stats::reg
stats::row
stats::sample
stats::sample2list
stats::selectRow
stats::sortSample
stats::stdev
stats::swGOFT
stats::tabulate
stats::tCDF
stats::tPDF
stats::tQuantile
stats::tRandom
stats::tTest
stats::uniformCDF
stats::uniformPDF
stats::uniformQuantile
stats::uniformRandom
stats::unzipCol
stats::variance
stats::weibullCDF
stats::weibullPDF
stats::weibullQuantile

stats::weibullRandom
stats::zipCol

stats::betaCDF

Cumulative distribution function of the beta distribution

Syntax

```
stats::betaCDF(a, b)
```

Description

`stats::betaCDF(a, b)` returns a procedure representing the cumulative distribution function

$$x \to \begin{cases} 0 & \text{if } x \le 0 \\ \frac{1}{\beta(a,b)} \int_0^x t^{a-1} (1-t)^{b-1} \, \mathrm{d}t & \text{if } 0 < x \wedge x < 1 \\ 1 & \text{if } x \ge 1 \end{cases}$$

of the beta distribution with shape parameters $a > 0$, $b > 0$.

The procedure `f := stats::betaCDF(a, b)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

- If `x` can be converted to a real floating-point number and `a` and `b` can be converted to positive floating-point numbers, then the return value `f(x)` is a floating-point number.
- For numerical values $x \le 0$ and $x \ge 1$, the floating-point numbers `0.0`, respectively `1.0`, are returned even if `a` and `b` are symbolic quantities.
- The call `f(- infinity )` returns `0.0`; the call `f( infinity )` return `1.0`.
- In all other cases, `f(x)` returns the symbolic call `stats::betaCDF(a, b)(x)`.

Numerical values of `a` and `b` are only accepted if they are positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the cumulative distribution function with $a = 5$ and $b = 7$ at various points:

```
f := stats::betaCDF(5, 7):
f(-infinity), f(-PI), f(1/sqrt(10)), f(0.75), f(1), f(infinity)
```

$$0.0,\ 0.0,\ 0.247351489,\ 0.9924387932,\ 1.0,\ 1.0$$

Nonpositive numerical values of a or b lead to an error:

```
stats::betaCDF(-5, 7)(0.75)
```

```
Error: The first shape parameter must be positive. [stats::betaCDF]

Error:
the first shape parameter must be positive [stats::betaCDF]
```

```
delete f:
```

Example 2

For symbolic arguments, symbolic calls of `stats::betaCDF` are returned, unless $x \le 0$ or $x \ge 1$ can be decided:

```
f := stats::betaCDF(a, b):
f(-2), f(0), f(1/3), f(0.4), f(1), f(PI), f(x)
```

$$0.0,\ 0.0,\ \text{stats::betaCDF}(a,\ b)\left(\frac{1}{3}\right),\ \text{stats::betaCDF}(a,\ b)(0.4),\ 1.0,\ 1.0,\ \text{stats::betaCDF}(a,\ b)(x)$$

When positive real numbers are assigned to a and b, the call $f(x)$ returns a floating-point number if x is numerical:

```
a := 2: b := PI:
f(-2), f(1/3), f(0.4),  f(PI)
```

```
     0.0, 0.4272662874, 0.5465772418, 1.0
```

```
delete f, a, b:
```

Parameters

a, b

The shape parameters of the beta distribution: arithmetical expressions representing positive real values.

Return Values

procedure.

See Also

MuPAD Functions

`stats::betaPDF` | `stats::betaQuantile` | `stats::betaRandom`

stats::betaPDF

Probability density function of the beta distribution

Syntax

```
stats::betaPDF(a, b)
```

Description

`stats::betaPDF(a, b)` returns a procedure representing the probability density function

$$x \to \begin{cases} \dfrac{x^{a-1}\,(1-x)^{b-1}}{\beta(a,b)} & \text{if } x > 0 \wedge x < 1 \\ 0 & \text{if } x \le 0 \vee x \ge 1 \end{cases}$$

of the beta distribution with shape parameters $a > 0$ and $b > 0$

The procedure `f := stats::betaPDF(a, b)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

- If `x` is a real floating-point number and `a` and `b` can be converted to positive floating-point numbers, then `f(x)` returns a floating-point number.
- If $0 < x < 1$ can be decided, the expression `x^(a-1)*(1-x)^(b-1)/beta(a, b)` is returned. If $x \le 0$ or $x \ge 1$ can be decided, then `0`, respectively `0.0`, is returned.
- The calls `f(- infinity )` and `f( infinity )` return 0.
- In all other cases, `f(x)` returns the symbolic call `stats::betaPDF(a, b)(x)`.

Numerical values of `a` and `b` are only accepted if they are positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision. The procedure returned by `stats::betaPDF` reacts to properties of its argument.

Examples

Example 1

We evaluate the probability density function with $a = 3$ and $b = 4$ at various points:

```
f := stats::betaPDF(3, 4):
f(-infinity), f(-1), f(1/2), f(0.7), f(infinity)
```

$$0,\ 0,\ \frac{15}{8},\ 0.7938,\ 0$$

```
delete f:
```

Example 2

If `x` is a symbolic object without properties, then it cannot be decided whether $0 < x < 1$ holds. A symbolic function call is returned:

```
f := stats::betaPDF(a, b): f(x)
```

$$\text{stats::betaPDF}(a,\ b)(x)$$

With suitable properties, an explicit expression is returned:

```
assume(0 < x < 1): f(x)
```

$$\frac{x^{a-1}\,(1-x)^{b-1}}{\beta(a,\ b)}$$

```
assume(x > 1): f(x)
```

```
    0
```

```
unassume(x): delete f:
```

Parameters

a, b

The shape parameters of the beta distribution: arithmetical expressions representing positive real values.

Return Values

procedure.

See Also

MuPAD Functions

`stats::betaCDF` | `stats::betaQuantile` | `stats::betaRandom`

stats::betaQuantile

Quantile function of the beta distribution

Syntax

```
stats::betaQuantile(a, b)
```

Description

`stats::betaQuantile(a, b)` returns a procedure representing the quantile function (inverse) of the cumulative distribution function `stats::betaCDF(a, b)`. For $0 \le x \le 1$, the solution of *stats*::*betaCDF*(*a*, *b*)(*y*) = *x* is given by *y* = *stats*::*betaQuantile*(*a*, *b*)(*x*).

The procedure `f := stats::betaQuantile(a, b)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number, or a symbolic expression:

- If `a` and `b` can be converted to positive floating-point numbers and `x` is a real number between 0 and 1, then the return value `f(x)` is a floating-point number between 0.0 and 1.0 approximating the real solution *y* of *stats*::*betaCDF*(*a*, *b*)(*y*) = *x*.
- `f(0)`, `f(0.0)`, `f(1)`, and `f(1.0)` produce 0, 0.0, 1, and 1.0, respectively, for all values of `a` and `b`.
- In all other cases, `f(x)` returns the symbolic call `stats::betaQuantile(a, b)(x)`.

Numerical values of `a` and `b` are only accepted if they are positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the quantile function with $a = \pi$ and $b = 11$ at the point $x = \frac{9}{10}$:

```
f := stats::betaQuantile(PI, 11): f(9/10)
```

$$0.368643722$$

The value `f(x)` satisfies `stats::betaCDF(PI, 11)(f(x)) = x`:

```
stats::betaCDF(PI, 11)(f(0.98765))
```

$$0.98765$$

```
delete f:
```

Example 2

For symbolic arguments, symbolic calls are returned:

```
f := stats::betaQuantile(a, b): f(x), f(0.9)
```

$$\text{stats::betaQuantile}(a, b)(x),\ \text{stats::betaQuantile}(a, b)(0.9)$$

If `a`, `b` evaluate to real numbers and `x` to a real number between 0 and 1, then the call `f(x)` produces a float:

```
a := 17: b := 6: f(0.9)
```

$$0.8499783131$$

Numerical values for x are only accepted if $0 \le x \le 1$:

```
f(2)
```

```
Error: An argument x with 0 <= x <= 1 is expected. [f]
```

```
delete f, a, b:
```

Parameters

a, b

The shape parameters of the beta distribution: arithmetical expressions representing positive real values.

Return Values

procedure.

See Also

MuPAD Functions

`stats::betaCDF` | `stats::betaPDF` | `stats::betaRandom`

stats::betaRandom

Generate a random number generator for beta deviates

Syntax

```
stats::betaRandom(a, b, <Seed = n>)
```

Description

`stats::betaRandom(a, b)` returns a procedure that produces beta deviates (random numbers) with shape parameters $a > 0$, $b > 0$.

The procedure `f := stats::betaRandom(a, b)` can be called in the form `f()`. The return value of `f()` is either a floating-point number or a symbolic expression:

- If `a` and `b` can be converted to positive floating-point numbers, then `f()` returns a random floating-point number between `0.0` and `1.0`.
- In all other cases, *f*() return the symbolic call `stats::betaRandom(a, b)()`.

Numerical values of `a` and `b` are only accepted if they are positive.

The values `X = f()` are distributed randomly according to the beta distribution with parameters `a` and `b`. For any $0 \le x \le 1$, the probability that $X \le x$ is given by

$$\frac{1}{\beta(a, b)} \int_0^x t^{a-1} (1-t)^{b-1} \, \mathrm{d}t$$

.

Without the option `Seed = n`, an initial seed is chosen internally. This initial seed is set to a default value when MuPAD is started. Thus, each time MuPAD is started or re-initialized with the `reset` function, random generators produce the same sequences of numbers.

Note: In contrast to the function `random`, the generators produced by `stats::betaRandom` do not react to the environment variable `SEED`.

For efficiency, it is recommended to produce sequences of K random numbers via

```
f := stats::betaRandom(a, b): f() $ k = 1..K;
```
rather than by

```
stats::betaRandom(a, b)() $ k = 1..K;
```
The latter call produces a sequence of generators each of which is called once. Also note that

```
stats::betaRandom(a, b, Seed = n)() $ k = 1..K;
```
does not produce a random sequence, because a sequence of freshly initialized generators would be created each of them producing the same number.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We generate beta deviates with parameters $a = 2$ and $b = \frac{3}{4}$:

```
f := stats::betaRandom(2, 3/4): f() $ k = 1..4
```

$$0.9454511844,\ 0.8615078721,\ 0.121495159,\ 0.379420364$$

```
delete f:
```

Example 2

With symbolic parameters, no random floating-point numbers can be produced:

```
f := stats::betaRandom(a, b): f()
```

$$\text{stats::betaRandom}(a,\ b)()$$

When a and b evaluate to positive real numbers, the generator starts to produce random numbers:

```
a := 1: b := 2: f() $ k = 1..4
```

```
0.1700845647, 0.1490672548, 0.6022714953, 0.2217977725
```

```
delete f, a, b:
```

Example 3

We use the option `Seed = s` to reproduce a sequence of random numbers:

```
f := stats::betaRandom(1, 3, Seed = 1): f() $ k = 1..4
```

```
0.07584461034, 0.6146360615, 0.2188856232, 0.1020817554
```

```
g := stats::betaRandom(1, 3, Seed = 1): g() $ k = 1..4
```

```
0.07584461034, 0.6146360615, 0.2188856232, 0.1020817554
```

```
f() = g(), f() = g()
```

```
0.3303369551 = 0.3303369551, 0.1975445744 = 0.1975445744
```

```
delete f, g:
```

Parameters

a, b

The shape parameters of the beta distribution: arithmetical expressions representing positive real values.

Options

Seed

Option, specified as `Seed = n`

Initializes the random generator with the integer seed `n`. n can also be the option `CurrentTime`, to make the seed depend on the current time.

This option serves for generating generators that return predictable sequences of pseudo-random numbers. The generator is initialized with the seed `n` which may be an arbitrary integer. Several generators with the same initial seed produce the same sequence of numbers.

When this option is used, the shape parameters `a` and `b` must be convertible to positive floating-point numbers at the time when the random generator is generated.

Return Values

procedure.

Algorithms

The implemented algorithm for the computation of the beta deviates uses gamma deviates `x`, `y` to produce a beta deviate `x/(x + y)`. For more information see: D. Knuth, Seminumerical Algorithms (1998), Vol. 2, p. 134.

See Also

MuPAD Functions

`stats::betaCDF` | `stats::betaPDF` | `stats::betaQuantile`

stats::binomialCDF

The (discrete) cumulative distribution function of the binomial distribution

Syntax

```
stats::binomialCDF(n, p)
```

Description

`stats::binomialCDF(n, p)` returns a procedure representing the (discrete) cumulative distribution function

$$x \to \begin{cases} 0 & \text{if } x < 0 \\ \sum_{i=0}^{\lfloor x \rfloor} \binom{n}{i} p^i (1-p)^{n-i} & \text{if } x \geq 0 \wedge x < 1 \\ 1 & \text{if } x \geq 1 \end{cases}$$

of the binomial distribution with “trial parameter” `n` and “probability parameter” `p`.

The procedure `f := stats::binomialCDF(n, p)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number, an exact numerical value, or a symbolic expression:

- If `x` is a numerical real value and `n` is a positive integer, then an explicit value is returned. If `p` is a numerical value satisfying $0 \leq p \wedge p \leq 1$, this is a numerical value. Otherwise, it is a symbolic expression in `p`.
- If `x` is a numerical value with $x < 0$, then `0`, respectively `0.0`, is returned for any value of `n` and `p`.
- For symbolic values of `n`, explicit results are returned if x is a numerical value with $x < 2$.
- For symbolic values of `n`, explicit results are returned if $n - x$ is a numerical value with $n - x \leq 2$.
- If `n - x` is a numerical value with $n - x \leq 0$, then `1`, respectively `1.0`, is returned for any value of `n` and `p`.

- In all other cases, `f(x)` returns the symbolic call `binomialCDF(n, p)(x)`.

Numerical values for `n` are only accepted if they are positive integers.

Numerical values for `p` are only accepted if they satisfy $0 \le p \le 1$.

If `x` is a real floating-point number, the result is a floating number provided n and p are numerical values. If `x` is an exact numerical value, the result is an exact number.

Note: Note that for large n, floating-point results are computed much faster than exact results. If floating-point approximations are desired, pass a floating-point number `x` to the procedure generated by `stats::binomialCDF`!

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the distribution function with $n = 20$ and $p = \frac{3}{4}$ at various points:

```
f := stats::binomialCDF(5, 3/4):
f(-1), f(2), f(PI), f(5), f(6)
```

$$0,\ \frac{53}{512},\ \frac{47}{128},\ 1,\ 1$$

```
f(-1.2), f(2.0), f(float(PI)), f(5.5)
```

$$0.0,\ 0.103515625,\ 0.3671875,\ 1.0$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::binomialCDF(n, p): f(x), f(8), f(8.0)
```

$$\text{stats::binomialCDF}(n, p)(x),\ \text{stats::binomialCDF}(n, p)(8),\ \text{stats::binomialCDF}(n, p)(8.0)$$

When numerical values are assigned to n and p, the function f starts to produce explicit results if the argument is numerical:

```
n := 3: p := 1/3:
f(2), f(2.5), f(PI +1), f(4.0)
```

$$\frac{26}{27},\ 0.962962963,\ 1,\ 1.0$$

```
delete f, n, p:
```

Example 3

If n and x are numerical, symbolic expressions are returned for symbolic values of p:

```
f := stats::binomialCDF(3, p):
f(-1), f(0), f(3/2), f(1 + sqrt(3)), f(2.999), f(3)
```

$$0,\ -(p-1)^3,\ 3\,p\,(p-1)^2-(p-1)^3,\ 1-p^3,\ 1.0-1.0\,p^3,\ 1$$

```
delete f:
```

Parameters

n

The “trial parameter”: an arithmetical expression representing a positive integer

p

The “probability parameter”: an arithmetical expression representing a real number $0 \le p \le 1$.

Return Values

procedure.

See Also

MuPAD Functions

`stats::binomialPF` | `stats::binomialQuantile` | `stats::binomialRandom`

stats::binomialPF

Probability function of the binomial distribution

Syntax

```
stats::binomialPF(n, p)
```

Description

`stats::binomialPF(n, p)` returns a procedure representing the probability function

$$x \to \binom{n}{x} p^x (1-p)^{n-x}$$

for `x=0,1,..,n` of the binomial distribution with "trial parameter" `n` and "probability parameter" `p`.

The procedure `f := stats::binomialPF(n, p)` can be called in the form `f(x)` with arithmetical expressions `x`. The return value of `f(x)` is either a floating-point number, an exact numerical value, or a symbolic expression:

- If `x` is a non-integer numerical value, $f(x)$ returns `0` or `0.0`, respectively.
- If `x` is an integer or the floating point equivalent of an integer and `n` is a positive integer, then an explicit value is returned. If `p` is a numerical value satisfying $0 \le p \le 1$, this is a numerical value. Otherwise, it is a symbolic expression in `p`.

  For symbolic values of `n`, explicit results are returned if x is a numerical value with$x < 2$.
- For symbolic values of `n`, explicit results are returned if $n - x$ is a numerical value with$n - x < 2$.
- In all other cases, `f(x)` returns the symbolic call `stats::binomialPF(n,p)(x)`.

Numerical values for `n` are only accepted if they are positive integers.

Numerical values for `p` are only accepted if they satisfy $0 \le p \le 1$.

If `x` is a floating-point number, the result is a floating number provided n and p are numerical values. If `x` is an exact value, the result is an exact number.

Note that for large n, floating-point results are computed much faster than exact results. If floating-point approximations are desired, pass a floating-point number `x` to the procedure created by `stats::binomialPF`.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We compute the probability function with $n = 3$ and $p = \frac{3}{4}$ at various points:

```
f := stats::binomialPF(3, 3/4):
f(-1/2), f(0), f(1/2), f(1), f(7/4), f(2), f(3), f(4)
```

$$0,\ \frac{1}{64},\ 0,\ \frac{9}{64},\ 0,\ \frac{27}{64},\ \frac{27}{64},\ 0$$

```
f(-0.2), f(0.0), f(0.7), f(1.0), f(2.0), f(2.7), f(3.0), f(4.0)
```

$$0.0,\ 0.015625,\ 0.0,\ 0.140625,\ 0.421875,\ 0.0,\ 0.421875,\ 0.0$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::binomialPF(n, p): f(x), f(8), f(8.0)
```

$$\text{stats::binomialPF}(n,\ p)(x),\ \text{stats::binomialPF}(n,\ p)(8),\ \text{stats::binomialPF}(n,\ p)(8.0)$$

When real numbers are assigned to n and p, the function f starts to produce explicit results if the argument is numerical:

```
n := 3: p := 1/3:
f(0), f(1), f(2.0), f(3.5), f(4)
```

$$\frac{8}{27},\ \frac{4}{9},\ 0.2222222222,\ 0.0,\ 0$$

```
delete f, n, p, x:
```

Example 3

If n and x are numerical, symbolic expressions are returned for symbolic values of p:

```
f := stats::binomialPF(3, p):
f(-1), f(0), f(3/2), f(2), f(3)
```

$$0,\ -(p-1)^3,\ 0,\ -3\,p^2\,(p-1),\ p^3$$

```
delete f:
```

Parameters

n

The “trial parameter”: an arithmetical expression representing a positive integer

p

The “probability parameter”: an arithmetical expression representing a real number $0 \le p \le 1$.

Return Values

procedure.

See Also

MuPAD Functions

`stats::binomialCDF` | `stats::binomialQuantile` | `stats::binomialRandom`

stats::binomialQuantile

Quantile function of the binomial distribution

Syntax

```
stats::binomialQuantile(n, p)
```

Description

`stats::binomialQuantile(n, p)` returns a procedure representing the quantile function (discrete inverse) of the cumulative distribution function `stats::binomialCDF(n, p)`. For $0 \le x \le 1$, the quantile value k = *stats*::*binomialQuantile*(n, p)(x) satisfies

$$\text{stats::binomialCDF}(n,\ p)(k-1) < x \wedge x \le \text{stats::binomialCDF}(n,\ p)(k)$$

The procedure `f := stats::binomialQuantile(n, p)` can be called in the form `f(x)` with arithmetical expressions `x`. The return value of `f(x)` is either a natural number between 0 and n, or a symbolic expression:

- If n is a positive integer, p a real number satisfying $0 \le p \le 1$, and x a real number satisfying $0 \le x \le 1$, then $f(x)$ returns an integer between 0 and n.
- If $p = 0$, then $f(x)$ returns 0 for any values of n and x.
- If $p = 1$, then $f(x)$ returns n for any values of n and x.
- For $p \ne 1$, the call $f(0)$ returns 0 for any value of n.
- For $p \ne 0$, the call $f(1)$ returns n for any value of n.
- In all other cases, `f(x)` returns the symbolic call `stats::binomialQuantile(n, p)(x)`.

Numerical values for `n` are only accepted if they are positive integers.

Numerical values for `p` are only accepted if they satisfy $0 \le p \le 1$.

If floating-point arguments are passed to the quantile function f, the result is computed with floating-point arithmetic. This is faster than using exact arithmetic, but the result

is subject to internal round-off errors. In particular, round-off may be significant for arguments x close to 1. Cf. “Example 3” on page 30-28.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the quantile function with $n = 30$ and $p = \frac{1}{3}$ at some points:

```
f := stats::binomialQuantile(30, 1/3):
f(0), f((2/3)^30), f(PI/10), f(0.5), f(1 - 1/10^10)
```

```
0, 0, 9, 10, 27
```

The quantile value $f(x)$ satisfies

$$\text{stats::binomialCDF}(n, p)(f(x)-1) < x \wedge x \le \text{stats::binomialCDF}(n, p)(f(x))$$:

```
x := 0.7: f(x)
```

```
11
```

```
stats::binomialCDF(30, 1/3)(float(f(x) - 1)), x,
stats::binomialCDF(30, 1/3)(float(f(x)))
```

```
0.5847595988, 0.7, 0.7238643653
```

```
delete f, x:
```

Example 2

We use symbolic arguments:

```
f := stats::binomialQuantile(n, p): f(x), f(9/10)
```

$$\text{stats::binomialQuantile}(n, p)(x),\ \text{stats::binomialQuantile}(n, p)\left(\frac{9}{10}\right)$$

When `n` and `p` evaluate to suitable numbers, the function `f` starts to produce quantile values:

```
n := 80: p := 1/10:
f(1/2),  f(999/1000), f(1 - 1/10^10), f(1 - 1/10^80)
```

$$8,\ 17,\ 29,\ 79$$

```
delete f, n, p:
```

Example 3

If floating-point arguments are passed to the quantile function, the result is computed with floating-point arithmetic. This is faster than using exact arithmetic, but the result is subject to internal round-off errors:

```
f := stats::binomialQuantile(1000, 1/30):
f(1 - 1/10^16) <> f(float(1 - 1/10^16))
```

$$89 \neq 88$$

```
delete f:
```

Parameters

n

The “trial parameter”: an arithmetical expression representing a positive integer

p

The “probability parameter”: an arithmetical expression representing a real number $0 \leq p \leq 1$.

Return Values

procedure.

See Also

MuPAD Functions

`stats::binomialCDF` | `stats::binomialPF` | `stats::binomialRandom`

stats::binomialRandom

Generate a random number generator for binomial deviates

Syntax

```
stats::binomialRandom(n, p, <Seed = s>)
```

Description

`stats::binomialRandom(n, p)` returns a procedure that produces binomial-deviates (random numbers) with trial parameter `n` and probability parameter `p`.

The procedure `f := stats::binomialRandom(n, p)` can be called in the form `f()`. The return value of `f()` is an integer between 0 and n or a symbolic expression:

- If `n` is a positive integer and `p` is a real value satisfying $0 \le p \le 1$, then `f()` returns an integer between 0 and n.
- If $p = 0$ or $p = 0.0$, then `f()` returns `0` for any value of n.
- If $p = 1$ or $p = 1.0$, then $f()$ returns `n` for any value of n.

  In all other cases, $f()$ return the symbolic call `stats::binomialRandom(n, p)()`.

Numerical values for `n` are only accepted if they are positive integers.

Numerical values for `p` are only accepted if they satisfy $0 \le p \le 1$.

The values `X = f()` are distributed randomly according to the binomial distribution with trial parameter `n` and probability parameter `p`. For any $x \in [0, 1]$, the probability of $X \le x$ is given by

$$\sum_{i=0}^{\lfloor x \rfloor} \binom{n}{i} p^i (1-p)^{n-i}$$

.

Without the option `Seed = s`, an initial seed is chosen internally. This initial seed is set to a default value when MuPAD is started. Thus, each time MuPAD is started or re-

initialized with the `reset` function, random generators produce the same sequences of numbers.

Note: With this option, the parameters `n` and `p` must evaluate to suitable numerical values at the time, when the generator is created.

Note: In contrast to the function `random`, the generators produced by `stats::binomialRandom` do not react to the environment variable `SEED`.

For efficiency, it is recommended to produce sequences of K random numbers via

```
f := stats::binomialRandom(n, p): f() $k = 1..K;
```

rather than by

```
stats::binomialRandom(n, p)() $k = 1..K;
```

The latter call produces a sequence of generators each of which is called once. Also note that

```
stats::binomialRandom(n, p, Seed = s)() $k = 1..K;
```

does not produce a random sequence, because a sequence of freshly initialized generators would be created each of them producing the same number.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We generate binomial deviates with parameters $n = 80$ and $p = \frac{1}{7}$:

```
f := stats::binomialRandom(80, 1/7): f() $ k = 1..10
```

$$9, 9, 16, 11, 8, 5, 11, 13, 14, 12$$

```
delete f:
```

Example 2

With symbolic parameters, no random numbers can be produced:

```
f := stats::binomialRandom(n, p): f()
```

$$\text{stats::binomialRandom}(n, p)()$$

When n and p evaluate to suitable numbers, the generator starts to produce random numbers:

```
n := 200: p := 1/PI: f() $ k= 1..10
```

$$79, 69, 69, 64, 77, 70, 80, 66, 62, 69$$

```
delete f, n, p:
```

Example 3

We use the option `Seed = s` to reproduce a sequence of random numbers:

```
f := stats::binomialRandom(70, 7/8, Seed = 1): f() $ k = 1..10
```

$$63, 65, 60, 65, 60, 57, 58, 63, 57, 61$$

```
g := stats::binomialRandom(70, 7/8, Seed = 1): g() $ k = 1..10
```

$$63, 65, 60, 65, 60, 57, 58, 63, 57, 61$$

```
f() = g(), f() = g()
```

$$61 = 61, 63 = 63$$

```
delete f, g:
```

Parameters

n

The “trial parameter”: an arithmetical expression representing a positive integer

p

The “probability parameter”: an arithmetical expression representing a real number $0 \leq p \leq 1$.

Options

Seed

Option, specified as `Seed = s`

Initializes the random generator with the integer seed `s`. s can also be the option `CurrentTime`, to make the seed depend on the current time.

This option serves for generating generators that return predictable sequences of pseudo-random numbers. The generator is initialized with the seed `s` which may be an arbitrary integer. Several generators with the same initial seed produce the same sequence of numbers.

When this option is used, the parameters `n` and `p` must be numerical values at the time when the random generator is generated.

Return Values

procedure.

See Also

MuPAD Functions

`stats::binomialCDF` | `stats::binomialPF` | `stats::binomialQuantile`

stats::calc

Apply functions to a sample

Syntax

`stats::calc(s, c, f1, f2, …)`

`stats::calc(s, [c1, c2, …], f1, f2, …)`

Description

`stats::calc` applies functions to columns of the sample `s`.

In a call such as `stats::calc(s, c, f1)` the function f_1 is applied to the elements of the column c of s. This generates a new column which is appended to s. If present, the next function f_2 is applied to the new sample etc. Thus, a call of `stats::calc` with *m* functions appends *m* new columns to s.

Each function must accept exactly one parameter.

In a call such as `stats::calc(s, [c1, c2, …], f1)` the *i*-th element of the new column is given by `f1(si, c1, si, c2, …)`.

Each function must accept as many parameters as specified by the second argument of `stats::calc`.

Examples

Example 1

We create a sample of three rows and three columns:

```
stats::sample([[1, a1, b1], [2, a2, b2], [3, a3, b3]])
```

```
1  a1  b1
2  a2  b2
3  a3  b3
```

We add and multiply the elements of the columns 2 and 3 by applying the system functions `_plus` and `_mult`:

```
stats::calc(%, [2, 3], _plus, _mult)
```

```
1  a1  b1  a1 + b1  a1*b1
2  a2  b2  a2 + b2  a2*b2
3  a3  b3  a3 + b3  a3*b3
```

The following call maps each element of the second column of the original sample to its fourth power:

```
stats::calc(%2, 2, x -> x^4)
```

```
1  a1  b1  a1^4
2  a2  b2  a2^4
3  a3  b3  a3^4
```

The following call computes the mean values of the rows of the last sample:

```
stats::calc(%, [1, 2, 3, 4],
            (x1, x2, x3, x4) -> (x1 + x2 + x3 + x4)/4)
```

```
1  a1  b1  a1^4  a1^4/4 + a1/4 + b1/4 + 1/4
2  a2  b2  a2^4  a2^4/4 + a2/4 + b2/4 + 1/2
3  a3  b3  a3^4  a3^4/4 + a3/4 + b3/4 + 3/4
```

The same is achieved by the following call:

```
stats::calc(%2, [1, 2, 3, 4], stats::mean)
```

```
1  a1  b1  a1^4  a1^4/4 + a1/4 + b1/4 + 1/4
2  a2  b2  a2^4  a2^4/4 + a2/4 + b2/4 + 1/2
3  a3  b3  a3^4  a3^4/4 + a3/4 + b3/4 + 3/4
```

Parameters

s

A sample of domain type `stats::sample`

`c, c1, c2, …`

Positive integers representing column indices of the sample

`f1, f2, …`

Procedures

Return Values

Sample of domain type `stats::sample`.

See Also

MuPAD Functions
`stats::tabulate`

stats::cauchyCDF

Cumulative distribution function of the Cauchy distribution

Syntax

```
stats::cauchyCDF(a, b)
```

Description

`stats::cauchyCDF(a, b)` returns a procedure representing the cumulative distribution function

$$x \to \frac{1}{\pi} \arctan\left(\frac{x-a}{b}\right) + \frac{1}{2}$$

of the Cauchy distribution with median a and scale parameter $b > 0$.

The procedure `f := stats::cauchyCDF(a, b)` can be called in the form `f(x)` with arithmetical expressions `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

- If `x` is a floating-point number and `a` and `b` can be converted to suitable floating-point numbers, then `f(x)` returns a floating-point number.
- In all other cases, the symbolic expression `arctan((x - a)/b)/PI + 1/2` is returned.

Numerical values of `a` and `b` are only accepted if they are real and `b` is positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the cumulative distribution function with $a = 2$ and $b = \frac{3}{4}$ at various points:

```
f := stats::cauchyCDF(2, 3/4):
f(-infinity), f(-10), f(0.8), f(2), f(10.0^4), f(infinity)
```

$$0,\ \frac{1}{2} - \frac{\arctan(16)}{\pi},\ 0.1778076845,\ \frac{1}{2},\ 0.999976122,\ 1$$

```
delete f, x:
```

Example 2

We use symbolic arguments:

```
f := stats::cauchyCDF(a, b):
f(x), f(sqrt(2)), f(0.9)
```

$$\frac{1}{2} - \frac{\arctan\left(\frac{a-x}{b}\right)}{\pi},\ \frac{1}{2} - \frac{\arctan\left(\frac{a-\sqrt{2}}{b}\right)}{\pi},\ \frac{1}{2} - \frac{\arctan\left(\frac{a-0.9}{b}\right)}{\pi}$$

When numbers are assigned to a and b, the function f starts to produce corresponding numerical values:

```
a := PI:
b := 1/8:
f(sqrt(2)), f(0.9)
```

$$\frac{\arctan(8\sqrt{2} - 8\pi)}{\pi} + \frac{1}{2},\ 0.01773184344$$

Parameters

a

The median: an arithmetical expression representing a real value

b

The scale parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`stats::cauchyPDF` | `stats::cauchyQuantile` | `stats::cauchyRandom`

stats::cauchyPDF

Probability density function of the Cauchy distribution

Syntax

```
stats::cauchyPDF(a, b)
```

Description

`stats::cauchyPDF(a, b)` returns a procedure representing the probability density function

$$x \to \frac{b}{\pi} \frac{1}{(x-a)^2 + b^2}$$

of the Cauchy distribution with median a and scale parameter $b > 0$.

The procedure `f := stats::cauchyPDF(a, b)` can be called in the form `f(x)` with arithmetical expressions `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

- If `x` is a floating-point number and `a` and `b` can be converted to suitable floating-point numbers, then `f(x)` returns a floating-point number.
- In all other cases, the symbolic expression `b/PI * 1/((x-a)^2 + b^2)` is returned.

Numerical values of `a` and `b` are only accepted if they are real and `b` is positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We calculate the Cauchy density with $a = 2$ and $b = \frac{3}{4}$ at various points:

```
f := stats::cauchyPDF(2, 3/4):
f(-infinity), f(9/10), f(0.9), f(2), f(infinity)
```

$$0,\ \frac{300}{709\,\pi},\ 0.1346868348,\ \frac{4}{3\,\pi},\ 0$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::cauchyPDF(a, b):
f(x), f(2), f(2.0)
```

$$\frac{b}{\pi\,(b^2+(a-x)^2)},\ \frac{b}{\pi\,((a-2)^2+b^2)},\ \frac{b}{\pi\,(b^2+(a-2.0)^2)}$$

When a and b evaluate to numbers, the function f starts to produce numerical values:

```
a := PI:
b:= 1/8:
f(2), f(2.0)
```

$$\frac{1}{8\,\pi\left((\pi-2)^2+\frac{1}{64}\right)},\ 0.03016906448$$

```
delete f, a, b:
```

Parameters

a

The median: an arithmetical expression representing a real value

b

The scale parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`stats::cauchyCDF` | `stats::cauchyQuantile` | `stats::cauchyRandom`

stats::cauchyQuantile

Quantile function of the Cauchy distribution

Syntax

```
stats::cauchyQuantile(a, b)
```

Description

`stats::cauchyQuantile(a, b)` returns a procedure representing the quantile function (inverse) of the cumulative distribution function `stats::cauchyCDF(a, b)`: for $0 \le x \le 1$, the solution of *stats*::*cauchyCDF*(*a*, *b*)(*y*) = *x* is given by *y* = *stats*::*cauchyQuantile*(*a*, *b*)(*x*).

The procedure `f := stats::cauchyQuantile(a, b)` can be called in the form `f(x)` with arithmetical expressions `x`. The return value of `f(x)` is either a floating-point number, $\pm\infty$ or a symbolic expression:

If `x` is a floating-point number between `0` and `1` and `a` and `b` can be converted to suitable floating-point numbers, then `f(x)` returns a floating-point number approximating the real solution *y* of *stats*::*cauchyCDF*(*a*, *b*)(*y*) = *x*.

For any value of `a` and `b`, the calls `f(0)` and `f(0.0)` produce $-\infty$. The calls `f(1)` and `f(1.0)` produce *infinity*.

In all other cases, the symbolic expression `a + b*tan(PI*(x - 1/2))` is returned.

Numerical values of `a` and `b` are only accepted, if they are real and `b` is positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the quantile function with $a = 2$ and $b = \frac{3}{4}$ at various points:

```
f := stats::cauchyQuantile(2, 3/4):
f(0), f(4/5), f(0.8), f(1)
```

$$-\infty,\ \frac{3\sqrt{5}\sqrt{2\sqrt{5}+5}}{20}+2,\ 3.03228644,\ \infty$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::cauchyQuantile(a, b):
f(0), f(x), f(1/sqrt(2)), f(0.9), f(1)
```

$$-\infty,\ a+b\tan\left(\pi\left(x-\frac{1}{2}\right)\right),\ a+b\tan\left(\pi\left(\frac{\sqrt{2}}{2}-\frac{1}{2}\right)\right),\ a+b\tan(0.4\,\pi),\ \infty$$

When numbers are assigned to a and b, the function f starts to produce numerical values when called with arguments between 0 and 1:

```
a := PI: b := 1/8:
f(0), f(1/sqrt(2)), f(0.9), f(1)
```

$$-\infty,\ \pi+\frac{\tan\left(\pi\left(\frac{\sqrt{2}}{2}-\frac{1}{2}\right)\right)}{8},\ 3.526303096,\ \infty$$

Parameters

a

The median: an arithmetical expression representing a real value

b

The scale parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`stats::cauchyCDF` | `stats::cauchyPDF` | `stats::cauchyRandom`

stats::cauchyRandom

Generate a random number generator for Cauchy deviates

Syntax

```
stats::cauchyRandom(a, b, <Seed = n>)
```

Description

`stats::cauchyRandom(a, b)` returns a procedure that produces Cauchy deviates (random numbers) with median a and scale parameter $b > 0$.

The procedure `f := stats::cauchyRandom(a, b)` can be called in the form `f()`. The return value of `f()` is either a floating-point number or a symbolic expression:

If `a` can be converted to a real floating point number and `b` to a positive real floating point number, then `f()` returns a real floating point number.

In all other cases, $f()$ returns the symbolic call `stats::cauchyRandom(a, b)()`.

Numerical values of `a` and `b` are only accepted, if they are real and b is positive.

The values `X = f()` are distributed randomly according to the the Cauchy distribution with parameters `a` and `b`. For any real `x`, the probability that $X \leq x$ is given by

$$\frac{1}{\pi}\arctan\left(\frac{x-a}{b}\right)+\frac{1}{2}.$$

Without the option `Seed = n`, an initial seed is chosen internally. This initial seed is set to a default value when MuPAD is started. Thus, each time MuPAD is started or re-initialized with the `reset` function, random generators produce the same sequences of numbers.

Note: In contrast to the function `random`, the generators produced by `stats::cauchyRandom` do not react to the environment variable `SEED`.

For efficiency, it is recommended to produce sequences of K random numbers via

```
f := stats::cauchyRandom(a, b): f() $ k = 1..K;
```
rather than by

```
stats::cauchyRandom(a, b)() $k = 1..K;
```
The latter call produces a sequence of generators each of which is called once. Also note that

```
stats::cauchyRandom(a, b, Seed = n)() $ k = 1..K;
```
does not produce a random sequence, because a sequence of freshly initialized generators would be created each of them producing the same number.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We generate Cauchy deviates with parameters $a = 2$ and $b = \frac{3}{4}$:

```
f := stats::cauchyRandom(2, 3/4): f() $ k = 1..4
```

$$1.340284406,\ 3.277664042,\ 0.5634392829,\ 48.01912393$$

```
delete f:
```

Example 2

With symbolic parameters, no random floating-point numbers can be produced:

```
f := stats::cauchyRandom(a, b): f()
```

$$\text{stats::cauchyRandom}(a,\ b)()$$

When a and b evaluate to suitable real numbers, the generator starts to produce random numbers:

```
a := -PI: b := 1/2: f() $ k = 1..4
```

$$-3.592915903,\ -3.928568815,\ -3.217434162,\ -2.82696038$$

```
delete f, a, b:
```

Example 3

We use the option `Seed = n` to reproduce a sequence of random numbers:

```
f := stats::cauchyRandom(PI, 3, Seed = 1): f() $ k = 1..4
```

$$3.786179405,\ 7.050017894,\ -4.775376375,\ -1.791650747$$

```
g := stats::cauchyRandom(PI, 3, Seed = 1): g() $ k = 1..4
```

$$3.786179405,\ 7.050017894,\ -4.775376375,\ -1.791650747$$

```
f() = g(), f() = g()
```

$$8.362838563 = 8.362838563,\ -302.9342996 = -302.9342996$$

```
delete f, g:
```

Parameters

a

The median: an arithmetical expression representing a real value

b

The scale parameter: an arithmetical expression representing a positive real value

Options

Seed

Option, specified as `Seed = n`

Initializes the random generator with the integer seed `n`. n can also be the option `CurrentTime`, to make the seed depend on the current time.

This option serves for generating generators that return predictable sequences of pseudo-random numbers. The generator is initialized with the seed `n` which may be an arbitrary integer. Several generators with the same initial seed produce the same sequence of numbers.

When this option is used, the parameters `a` and `b` must be convertible to suitable floating-point numbers at the time when the random generator is generated.

Return Values

procedure.

Algorithms

The implemented algorithm for the computation of the Cauchy deviates uses the quantile function of the Cauchy distribution applied to uniformly distributed random numbers between 0 and 1.

See Also

MuPAD Functions

`stats::cauchyCDF` | `stats::cauchyPDF` | `stats::cauchyQuantile`

stats::chisquareCDF

Cumulative distribution function of the chi-square distribution

Syntax

```
stats::chisquareCDF(m)
```

Description

`stats::chisquareCDF(m)` returns a procedure representing the cumulative distribution function

$$x \to \begin{cases} \int_0^x \frac{t^{\frac{m}{2}-1}\, e^{-\frac{t}{2}}}{2^{m/2}\,\Gamma\left(\frac{m}{2}\right)}\, dt & \text{if } x > 0 \\ 0 & \text{if } x \le 0 \end{cases}$$

of the chi-square distribution with mean $m > 0$.

The procedure `f := stats::chisquareCDF(m)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

If $x \le 0$ can be decided, then `f(x)` returns 0. If $x > 0$ can be decided, then `f(x)` returns the value $1 - \frac{\Gamma\left(\frac{m}{2}, \frac{x}{2}\right)}{\Gamma\left(\frac{m}{2}\right)}$.

If x is a floating-point number and m can be converted to a positive floating-point number, then these values are returned as floating-point numbers. Otherwise, symbolic expressions are returned.

The function `f` reacts to properties of identifiers set via `assume`. If x is a symbolic expression with the property $x \le 0$ or $x \ge 0$, the corresponding values are returned.

`f(x)` returns the symbolic call `stats::chisquareCDF(m)(x)` if neither $x \leq 0$ nor $x > 0$ can be decided.

Numerical values for `m` are only accepted if they are real and positive.

Note that, for large `m`, exact results may be costly to compute. If floating-point values are desired, it is recommended to pass floating-point arguments `x` to `f` rather than to compute exact results `f(x)` and convert them via `float`. Cf. "Example 4" on page 30-53.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the cumulative distribution function with mean $m = 2$ at various points:

```
f := stats::chisquareCDF(2):
f(-infinity), f(-3), f(1/2), f(0.5), f(PI), f(infinity)
```

$$0,\ 0,\ 1 - e^{-\frac{1}{4}},\ 0.2211992169,\ 1 - e^{-\frac{\pi}{2}},\ 1$$

```
delete f:
```

Example 2

If `x` is a symbolic object without properties, then it cannot be decided whether $x \geq 0$ holds. A symbolic function call is returned:

```
f := stats::chisquareCDF(m):
f(x)
```

$$\text{stats::chisquareCDF}(m)(x)$$

With suitable properties, it can be decided whether $x \geq 0$ holds. An explicit expression is returned:

```
assume(0 <= x):
f(x)
```

$$1 - \frac{\Gamma\left(\frac{m}{2}, \frac{x}{2}\right)}{\Gamma\left(\frac{m}{2}\right)}$$

For integer values of `m`, the special function `igamma` can be expressed in terms of more elementary functions:

```
m := 6:
f(x)
```

$$1 - \frac{e^{-\frac{x}{2}} \left(\frac{x^2}{4} + x + 2\right)}{2}$$

```
m := 5:
f(x)
```

$$1 - \frac{4\left(\frac{3\sqrt{\pi}\,\mathrm{erfc}\left(\sqrt{\frac{x}{2}}\right)}{4} + e^{-\frac{x}{2}}\left(\frac{3\sqrt{\frac{x}{2}}}{2} + \left(\frac{x}{2}\right)^{3/2}\right)\right)}{3\sqrt{\pi}}$$

```
unassume(x): delete f, m:
```

Example 3

We use a symbolic mean `m`:

```
f := stats::chisquareCDF(m):
f(3), f(3.0)
```

$$1 - \frac{\Gamma\left(\frac{m}{2}, \frac{3}{2}\right)}{\Gamma\left(\frac{m}{2}\right)},\ 1.0 - \frac{1.0\,\Gamma(0.5\,m, 1.5)}{\Gamma(0.5\,m)}$$

When a numerical value is assigned to `m`, the function `f` starts to produce numerical values:

```
m := PI:
f(3), f(3.0)
```

$$1 - \frac{\Gamma\left(\frac{\pi}{2}, \frac{3}{2}\right)}{\Gamma\left(\frac{\pi}{2}\right)}, \quad 0.5840678031$$

```
delete f, m:
```

Example 4

We consider a chi-square distribution with large mean `m` = 1000:

```
f := stats::chisquareCDF(1000):
```

For floating-point approximations, one should not compute an exact result and convert it via `float`. For large mean m, it is faster to pass a floating-point argument to `f`. The following call takes some time, because an exact computation of the huge integer `gamma(m/2) = gamma(500) = 499!` is involved:

```
float(f(1023))
```

$$0.7003071959$$

The following call is much faster:

```
f(float(1023))
```

$$0.7003071959$$

```
delete f:
```

Parameters

`m`

The mean: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`gamma` | `igamma` | `stats::chisquarePDF` | `stats::chisquareQuantile` | `stats::chisquareRandom`

stats::chisquarePDF

Probability density function of the chi-square distribution

Syntax

```
stats::chisquarePDF(m)
```

Description

`stats::chisquarePDF(m)` returns a procedure representing the probability density function

$$x \to \begin{cases} \dfrac{x^{\frac{m}{2}-1}\, e^{-\frac{x}{2}}}{2^{m/2}\,\Gamma\left(\frac{m}{2}\right)} & \text{if } x > 0 \\ 0 & \text{if } x \le 0 \end{cases}$$

of the chi-square distribution with mean $m > 0$.

The procedure `f := stats::chisquarePDF(m)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

If $x \le 0$ can be decided, then `f(x)` returns 0. If $x > 0$ can be decided, then `f(x)` returns the value $\dfrac{x^{\frac{m}{2}-1}\, e^{-\frac{x}{2}}}{2^{m/2}\,\Gamma\left(\frac{m}{2}\right)}$.

If x is a floating-point number and m can be converted to a positive floating-point number, then these values are returned as floating-point numbers. Otherwise, symbolic expressions are returned.

The function `f` reacts to properties of identifiers set via `assume`. If x is a symbolic expression with the property $x \le 0$ or $x \ge 0$, the corresponding values are returned.

`f(x)` returns the symbolic call `stats::chisquarePDF(m)(x)` if neither $x \leq 0$ nor $x > 0$ can be decided,

Numerical values of `m` are only accepted if they are positive.

Note that, for large m, exact results may be costly to compute. If floating-point values are desired, it is recommended to pass floating-point arguments `x` to `f` rather than to compute exact results `f(x)` and convert them via `float`. Cf. “Example 4” on page 30-57.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the probability density function with $m = 2$ at various points:

```
f := stats::chisquarePDF(2):
f(-infinity), f(-PI), f(1/2), f(0.5), f(PI), f(infinity)
```

$$0,\ 0,\ \frac{e^{-\frac{1}{4}}}{2},\ 0.3894003915,\ \frac{e^{-\frac{\pi}{2}}}{2},\ 0$$

```
delete f:
```

Example 2

If `x` is a symbolic object without properties, then it cannot be decided whether $x \geq 0$ holds. A symbolic function call is returned:

```
f := stats::chisquarePDF(m): f(x)
```

$$\text{stats::chisquarePDF}(m)(x)$$

With suitable properties, it can be decided whether $x \geq 0$ holds. An explicit expression is returned:

```
assume(0 <= x): f(x)
```

$$\frac{x^{\frac{m}{2}-1}\, e^{-\frac{x}{2}}}{2^{m/2}\,\Gamma\left(\frac{m}{2}\right)}$$

```
unassume(x): delete f:
```

Example 3

We use symbolic a symbolic mean `m`:

```
f := stats::chisquarePDF(m): f(x)
```

$$\text{stats::chisquarePDF}(m)(x)$$

When a numerical value is assigned to `m`, the function `f` starts to produce numerical values:

```
m := PI: f(3), f(3.0)
```

$$\frac{3^{\frac{\pi}{2}-1}\, e^{-\frac{3}{2}}}{2^{\pi/2}\,\Gamma\left(\frac{\pi}{2}\right)},\ 0.1578981008$$

```
delete f, m:
```

Example 4

We consider a chi-square distribution with large mean `m` = 1000:

```
f := stats::chisquarePDF(1000):
```

For floating-point approximations, one should not compute an exact result and convert it via `float`. For large mean m, it is faster to pass a floating-point argument to `f`.

The following call takes some time, because an exact computation of the huge integer `gamma(m/2) = gamma(500) = 499!` is involved:

```
float(f(1023))
```

```
0.00765380452
```

The following call is much faster:

```
f(float(1023))
```

```
0.00765380452
```

```
delete f:
```

Parameters

`m`

The mean: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`gamma` | `stats::chisquareCDF` | `stats::chisquareQuantile` | `stats::chisquareRandom`

stats::chisquareQuantile

Quantile function of the chi-square distribution

Syntax

```
stats::chisquareQuantile(m)
```

Description

`stats::chisquareQuantile(m)` returns a procedure representing the quantile function (inverse) of the cumulative distribution function `stats::chisquareCDF(m)`. For $0 \le x \le 1$, the solution of *stats*::*chisquareCDF*(*m*)(*y*) = *x* is given by *y* = *stats*::*chisquareQuantile*(*m*)(*x*).

The procedure `f := stats::chisquareQuantile(m)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number, *infinity*, or a symbolic expression:

If `x` is a real number between `0` and `1` and `m` can be converted to a positive floating-point number, then `f(x)` returns a positive floating-point number approximating the solution *y* of *stats*::*chisquareCDF*(*m*)(*y*) = *x*.

The calls `f(0)` and `f(0.0)` produce `0.0` for all values of `m`.

The calls `f(1)` and `f(1.0)` produce *infinity* for all values of `m`.

In all other cases, `f(x)` returns the symbolic call `stats::chisquareQuantile(m)(x)`.

Numerical values of `m` are only accepted if they are real and positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the quantile function with $m = \pi$ at various points:

```
f := stats::chisquareQuantile(PI):
f(0), f(1/10), f(0.5), f(1 - 10^(-10)), f(1)
```

$$0.0,\ 0.6469867417,\ 2.505845123,\ 50.00263604,\ \infty$$

The value `f(x)` satisfies `stats::chisquareCDF(PI)(f(x)) = x`:

Der Wert `f(x)` erfüllt `stats::chisquareCDF(PI)(f(x)) = x`:

```
stats::chisquareCDF(PI)(f(0.987654))
```

$$0.987654$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::chisquareQuantile(m): f(x), f(9/10)
```

$$\text{stats::chisquareQuantile}(m)(x),\ \text{stats::chisquareQuantile}(m)\left(\frac{9}{10}\right)$$

When a positive real value is assigned to `m`, the function `f` starts to produce floating-point values:

```
m := PI + 1: f(0.999), f(1 - sqrt(2)/10^5)
```

$$18.76468483,\ 28.07485542$$

Numerical values for x are only accepted if $0 \le x \le 1$:

```
f(2)
```

```
Error: An argument x with 0 <= x <= 1 is expected. [f]
```

```
delete f, m:
```

Parameters

m

The mean: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`stats::chisquareCDF` | `stats::chisquarePDF` | `stats::chisquareRandom`

stats::chisquareRandom

Generate a random number generator for chi-square deviates

Syntax

```
stats::chisquareRandom(m, <Seed = n>)
```

Description

`stats::chisquareRandom(m)` returns a procedure that produces chi-square deviates (random numbers) with mean $m > 0$.

The procedure `f := stats::chisquareRandom(m)` can be called in the form `f()`. The return value of `f()` is either a floating-point number or a symbolic expression:

If `m` can be converted to a positive floating point number, then `f()` returns a nonnegative floating point number.

In all other cases, `stats::chisquareRandom(m)()` is returned symbolically.

A numerical value of `m` is only accepted if it is positive.

The values `X = f()` are distributed randomly according to the cumulative distribution function of the chi-square distribution with mean `m`. For any $x \geq 0$, the probability that $X \leq x$ is given by

$$\frac{1}{\Gamma\left(\frac{m}{2}\right) 2^{m/2}} \int_0^x t^{\frac{m}{2}-1} e^{-\frac{t}{2}} \, dt$$

.

Without the option `Seed = n`, an initial seed is chosen internally. This initial seed is set to a default value when MuPAD is started. Thus, each time MuPAD is started or re-initialized with the `reset` function, random generators produce the same sequences of numbers.

Note: In contrast to the function `random`, the generators produced by `stats::chisquareRandom` do not react to the environment variable `SEED`.

For efficiency, it is recommended to produce sequences of K random numbers via

```
f := stats::chisquareRandom(m): f() $k = 1..K;
```

rather than by

```
stats::chisquareRandom(m)() $k = 1..K;
```

The latter call produces a sequence of generators each of which is called once. Also note that

```
stats::chisquareRandom(m, Seed = n)() $k = 1..K;
```

does not produce a random sequence, because a sequence of freshly initialized generators would be created each of them producing the same number.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We generate chi-square deviates with mean $m = 12$:

```
f := stats::chisquareRandom(12): f() $ k = 1..4
```

```
17.54103319, 13.4630887, 17.34866815, 4.820644436
```

```
delete f:
```

Example 2

With symbolic parameters, no random floating-point numbers can be produced:

```
f := stats::chisquareRandom(m): f()
```

$$\text{stats::chisquareRandom}(m)()$$

When m evaluates to a positive real number, the generator starts to produce random numbers:

```
m := PI: f() $ k = 1..4
```

$$1.557180623,\ 0.3840766601,\ 0.5560473903,\ 3.185747198$$

```
delete f, m:
```

Example 3

We use the option `Seed = n` to reproduce a sequence of random numbers:

```
f := stats::chisquareRandom(70, Seed = 1): f() $ k = 1..4
```

$$55.24812677,\ 78.10283482,\ 68.16283459,\ 80.01866787$$

```
g := stats::chisquareRandom(70, Seed = 1): g() $ k = 1..4
```

$$55.24812677,\ 78.10283482,\ 68.16283459,\ 80.01866787$$

```
f() = g(), f() = g()
```

$$94.28358259 = 94.28358259,\ 57.54456 = 57.54456$$

```
delete f, g:
```

Parameters

`m`

The mean: an arithmetical expression representing a positive real value

Options

Seed

Option, specified as `Seed = n`

Initializes the random generator with the integer seed `n`. n can also be the option `CurrentTime`, to make the seed depend on the current time.

This option serves for generating generators that return predictable sequences of pseudo-random numbers. The generator is initialized with the seed `n` which may be an arbitrary integer. Several generators with the same initial seed produce the same sequence of numbers.

When this option is used, the mean `m` must be convertible to a positive floating-point number at the time when the random generator is generated.

Return Values

procedure.

Algorithms

The implemented algorithm for the computation of the chi-square deviates uses a gamma deviate `x` with shape parameters `m/2` and `2`. For more information see: D. Knuth, Seminumerical Algorithms (1998), Vol. 2, p. 135.

See Also

MuPAD Functions

`stats::chisquareCDF` | `stats::chisquarePDF` | `stats::chisquareQuantile`

stats::col

Select and rearrange columns of a sample

Syntax

```
stats::col(s, c1, <c2, …>)
stats::col(s, c1 .. c2, <c3 .. c4, …>)
```

Description

`stats::col(s, ...)` creates a new sample from selected columns of the sample `s`.

`stats::col` is useful for selecting columns of interest or for rearranging columns.

The columns of `s` specified by the remaining arguments of `stats::col` are used to build a new sample. The new sample contains the columns of `s` in the order specified by the call to `stats::col`. Columns can be duplicated by specifying the column index more than once.

Examples

Example 1

The following sample contains columns for “gender”, “age”, “height”, the “number of yellow socks” and “eye color” of a person:

```
stats::sample([["m", 26, 180, 3, "blue"],
               ["f", 22, 160, 0, "brown"],
               ["f", 48, 155, 2, "green"],
               ["m", 30, 172, 1, "brown"]])
```

```
"m"  26  180  3  "blue"
"f"  22  160  0  "brown"
"f"  48  155  2  "green"
"m"  30  172  1  "brown"
```

Since nobody is really interested in the yellow socks, we create a new sample without that column:

```
stats::col(%, 1..3, 5)
```

```
"m"  26  180  "blue"
"f"  22  160  "brown"
"f"  48  155  "green"
"m"  30  172  "brown"
```

We can use `stats::col` to rearrange the sample. As an illustrating example, we duplicate the first column:

```
stats::col(%, 1, 3, 2, 1, 4)
```

```
"m"  180  26  "m"  "blue"
"f"  160  22  "f"  "brown"
"f"  155  48  "f"  "green"
"m"  172  30  "m"  "brown"
```

Parameters

s

A sample of domain type `stats::sample`.

c_1, c_2, …

Positive integers representing column indices of the sample `s`. A range `c_1 .. c_2` represents all columns from c_1 through c_2.

Return Values

Sample of domain type `stats::sample`.

See Also

MuPAD Functions

`stats::concatCol` | `stats::concatRow` | `stats::row`

stats::concatCol

Concatenate samples column-wise

Syntax

`stats::concatCol(`s_1`, `s_2`, <`s_3`, …>)`

Description

`stats::concatCol(`s_1`, `s_2`, …)` creates a new sample consisting of the columns of the samples s_1, s_2 etc.

If the samples s_1, s_2 etc. have different numbers of rows, then the number of rows in the resulting sample is given by the "shortest" sample with the minimal number of rows. Elements below this row in "longer" samples are ignored.

Examples

Example 1

We create two samples:

```
s1 := stats::sample([[a1, a2], [b1, b2]]);
s2 := stats::sample([[a3, a4], [b3, b4]])
```

```
a1  a2
b1  b2
```

```
a3  a4
b3  b4
```

Concatenation of the columns yields:

```
stats::concatCol(s1, s2)
```

```
a1  a2  a3  a4
b1  b2  b3  b4

delete s1, s2:
```

Example 2

The following sample contains columns for “gender”, “age” and “height” of a person:

```
stats::sample([["m", 26, 180], ["f", 22, 160],
               ["f", 48, 155], ["m", 30, 172]])


"m"  26  180
"f"  22  160
"f"  48  155
"m"  30  172
```

We append a further column “nationality”, specified by a list:

```
stats::concatCol(%, ["German", "French", "Italian",
                     "British", "German"])


"m"  26  180  "German"
"f"  22  160  "French"
"f"  48  155  "Italian"
"m"  30  172  "British"
```

Parameters

s_1, s_2, …

Samples of domain type `stats::sample`. Alternatively, lists may be entered, which are treated as columns of a sample.

Return Values

Sample of domain type `stats::sample`.

See Also

MuPAD Functions

`stats::col` | `stats::concatRow` | `stats::row`

stats::concatRow

Concatenate samples row-wise

Syntax

`stats::concatRow(`s_1`, `s_2`, <`s_3`, …>)`

Description

`stats::concatRow(`s_1`, `s_2`, …)` creates a new sample consisting of the rows of the samples s_1, s_2 etc.

All samples must have the same number of columns.

Examples

Example 1

We create a small sample:

```
stats::sample([[123, g], [442, f]])
```

```
123  g
442  f
```

A list is concatenated to the sample as a row:

```
stats::concatRow(%, [x, y])
```

```
123  g
442  f
  x  y
```

Example 2

The following samples contain columns for “gender” and “age”:

```
s1 := stats::sample([["f", 36], ["m", 25]]);
s2 := stats::sample([["m", 26], ["f", 22]])
```

```
"f"  36
"m"  25
```

```
"m"  26
"f"  22
```

We build a larger sample:

```
stats::concatRow(s1, s2)
```

```
"f"  36
"m"  25
"m"  26
"f"  22
```

```
delete s1, s2:
```

Parameters

s_1, s_2, …

Samples of domain type `stats::sample`. Alternatively, lists may be entered, which are treated as rows of a sample.

Return Values

Sample of domain type `stats::sample`.

See Also

MuPAD Functions

`stats::col` | `stats::concatCol` | `stats::row`

stats::correlation

Correlation between data samples

Syntax

```
stats::correlation([x₁, x₂, …], [y₁, y₂, …], <BravaisPearson | Fechner>)
stats::correlation([[x₁, y₁], [x₂, y₂], …], <BravaisPearson | Fechner>)
stats::correlation(s, <c₁, c₂>, <BravaisPearson | Fechner>)
stats::correlation(s, <[c₁, c₂]>, <BravaisPearson | Fechner>)
stats::correlation(s₁, <c₁>, s₂, <c₂>, <BravaisPearson | Fechner>)
```

Description

`stats::correlation([x₁, x₂, …], [y₁, y₂, …])` returns the linear (Bravais-Pearson) correlation coefficient

$$\frac{\sum_i (x_i(y_i-\overline{y})-\overline{x}(y_i-\overline{y}))}{\sqrt{\left(\sum_i (x_i-\overline{x})^2\right)\left(\sum_i (y_i-\overline{y})^2\right)}},$$

where $\overline{x}$ and $\overline{y}$ are the means of the data x_i and y_i.

`stats::correlation([x₁, x₂, …], [y₁, y₂, …], Fechner)` returns the Fechner correlation $\frac{2}{n}\left(\sum_{i=1}^{n} v_i\right)-1$, where n is the sample size. The number v_i is 1, if $x_i-\overline{x}$ and $y_i-\overline{y}$ have the same sign or are both 0. It is $\frac{1}{2}$, if either $x_i-\overline{x}$ or $y_i-\overline{y}$ is 0. Otherwise, $v_i = 0$.

Both the Bravais-Pearson correlation as well as the Fechner correlation are numbers between - 1 and 1.

The Bravais-Pearson correlation is close to 1 if the data pairs x_i, y_i are approximately related by a 'positive' linear relation (i.e., $y_i \approx a\,x_i + b$ with some positive coefficient a). It is close to - 1 if there is a 'negative' linear relation (with some negative coefficient a).

Correlation coefficients close to 0 correspond to non-linear relations or to unrelated data, respectively.

If the input data are floating-point numbers, the sums defining the Bravais-Pearson correlation are computed in a numerically stable way. If a floating-point result is desired, it is recommended to make sure that all input data are floats.

The Fechner correlation is always returned as a rational number.

The column indices `c1`, `c2` are optional if the data are given by a `stats::sample` object `s` containing only two non-string data columns. If the data are provided by two samples `s1`, `s2`, the column indices are optional for samples containing only one non-string data column.

Note: The Fechner correlation should not be computed for symbolic data. This may lead to unexpected results, if the sign of symbolic parameters cannot be determined.

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

Examples

Example 1

We compute the correlation of samples passed as lists:

```
X := [7, 33/7, 3, 5, 2]: Y := [3, 5, 1, 7, 2]:
stats::correlation(X, Y)
```

$$\frac{147\sqrt{26506}}{53012}$$

Alternatively, the data may be passed as a list of data pairs:

```
stats::correlation([[7, 3], [33/7, 5], [3, 1], [5, 7], [2, 2]])
```

$$\frac{147\sqrt{26506}}{53012}$$

If all data are floating-point numbers, the result is a float:

```
stats::correlation(float(X), float(Y))
```

$$0.4514558056$$

The Fechner correlation of the data is always returned as a rational number:

```
stats::correlation(X, Y, Fechner),
stats::correlation(float(X), float(Y), Fechner)
```

$$\frac{3}{5}, \frac{3}{5}$$

The following exact result indicates an exact linear between the data pairs:

```
stats::correlation([0, 1, 2, 3], [7, 5, 3, 1])
```

$$-1$$

Indeed, there is the `negative' linear relation $y = 7 - 2\,x$ between the data pairs.

```
delete X, Y:
```

Example 2

We create a sample of type `stats::sample`:

```
s := stats::sample([[1.0, 2.4, 3.0],
                    [7.0, 4.8, 4.0],
                    [3.3, 3.0, 5.0]])
```

```
1.0  2.4  3.0
```

```
7.0  4.8  4.0
3.3  3.0  5.0
```

We compute the correlation between the data of the first and the third column in several equivalent ways:

```
stats::correlation(s, 1, 3),
stats::correlation(s, [1, 3]),
stats::correlation(s, 1, s, 3)
```

$$0.3799015783,\ 0.3799015783,\ 0.3799015783$$

```
stats::correlation(s, 1, 3, Fechner),
stats::correlation(s, [1, 3], Fechner),
stats::correlation(s, 1, s, 3, Fechner)
```

$$0,\ 0,\ 0$$

```
delete s:
```

Example 3

With symbolic data, the Bravais-Pearson correlation is returned as a symbolic expression:

```
stats::correlation([x1, x2], [y1, y2])
```

$$\frac{x1\ y1 + x2\ y2 - 2\left(\frac{x1}{2} + \frac{x2}{2}\right)\left(\frac{y1}{2} + \frac{y2}{2}\right)}{\sqrt{\left(x1^2 - 2\left(\frac{x1}{2} + \frac{x2}{2}\right)^2 + x2^2\right)\left(y1^2 - 2\left(\frac{y1}{2} + \frac{y2}{2}\right)^2 + y2^2\right)}}$$

```
simplify(%)
```

$$\frac{(x1 - x2)\ (y1 - y2)}{\sqrt{(x1 - x2)^2\ (y1 - y2)^2}}$$

Parameters

x_1, y_1, x_2, y_2, …

The statistical data: arithmetical expressions. The number of data x_i must coincide with the number of data y_i.

s, s_1, s_2

Samples of type stats::sample

c_1, c_2

Column indices: positive integers. Column c_1 of `s` or s_1, respectively, provides the data x_i. Column c_2 of `s` or `s_2`, respectively, provides the data y_i.

mode

Either `BravaisPearson` or `Fechner`. The default is the linear (Bravais-Pearson) correlation.

Options

BravaisPearson, Fechner

Linear (Bravais-Pearson) or Fechner's correlation coefficient. Bravais-Pearson coefficient is the default, but may in some cases where the data is not normally distributed be less useful than Fechner's correlation.

Return Values

The Bravais-Pearson correlation is returned as an arithmetical expression. `FAIL` is returned if the variance of one of the data samples vanishes (the Bravais-Pearson correlation does not exist).

The Fechner correlation is returned as a rational number.

`FAIL` is returned if the data samples are empty.

See Also

MuPAD Functions
`stats::correlationMatrix` | `stats::covariance` | `stats::stdev`

MuPAD Graphical Primitives
`plot::Scatterplot`

stats::correlationMatrix

Compute the correlation matrix associated with a covariance matrix

Syntax

```
stats::correlationMatrix(cov)
```

Description

`stats::correlationMatrix(cov)` returns to correlation matrix `cor` of the variance-covariance matrix `cov`. It is given by:

$$\mathrm{cor}_{i,j} = \frac{\mathrm{cov}_{i,j}}{\sqrt{\mathrm{cov}_{i,i}}\sqrt{\mathrm{cov}_{j,j}}}.$$

A covariance matrix C should be positive (semi-)definite and hence satisfies $|C_{ij}|^2 \le C_{ii}\,C_{jj}$ for all indices i, j. Consequently, the absolute values of the entries of the corresponding correlation matrix do not exceed 1.

With the option `CovarianceMatrix`, the routine `stats::reg` returns the variance-covariance matrix of the fit parameters in a regression analysis. The corresponding correlation matrix of the fit parameters is computed conveniently by applying `stats::correlationMatrix` to this matrix. Cf. “Example 2” on page 30-80.

Examples

Example 1

We generate a positive definite matrix that may serve as a covariance matrix:

```
A := matrix([[4, -3, 2], [-1, 2, 1], [0, 1, 1]]):
cov := A*linalg::transpose(A)
```

$$\begin{pmatrix} 29 & -8 & -1 \\ -8 & 6 & 3 \\ -1 & 3 & 2 \end{pmatrix}$$

The corresponding correlation matrix is:

```
stats::correlationMatrix(cov)
```

$$\begin{pmatrix} 1 & -\frac{4\sqrt{6}\sqrt{29}}{87} & -\frac{\sqrt{2}\sqrt{29}}{58} \\ -\frac{4\sqrt{6}\sqrt{29}}{87} & 1 & \frac{\sqrt{2}\sqrt{6}}{4} \\ -\frac{\sqrt{2}\sqrt{29}}{58} & \frac{\sqrt{2}\sqrt{6}}{4} & 1 \end{pmatrix}$$

If the input matrix consists of floating-point data, the result is a matrix of floats:

```
stats::correlationMatrix(float(cov))
```

$$\begin{pmatrix} 1.0 & -0.6064784349 & -0.1313064329 \\ -0.6064784349 & 1.0 & 0.8660254038 \\ -0.1313064329 & 0.8660254038 & 1.0 \end{pmatrix}$$

```
delete A, cov:
```

Example 2

We consider a covariance matrix arising in a non-linear regression problem. The model function $y = a + b\,cos(x - c)$ is to be fit to the following randomized data:

```
r := stats::uniformRandom(-0.1, 0.1):
xdata := [i $i = 1..100]:
ydata := [1 + 2*cos(x - 3) + r() $ x in xdata]:
```

By construction, the variance of the y values is the variance of the uniformly distributed random data on the interval [- 0.1, 0.1] generated by the random generator r. This variance is $\sigma^2 = \frac{1}{300}$. We use `stats::reg` to obtain estimates of the fit parameters a, b,

c of the model. Appropriate weights for the regression are given by $\frac{1}{\sigma^2} = 300$. The option `CovarianceMatrix` makes `stats::reg` include the covariance matrix `cov` of the fit parameters in its return list:

```
weights := [300 $ i = 1..100]:
DIGITS:= 4:
[abc, chisquare, cov] :=
   stats::reg(xdata, ydata, weights, a + b*cos(x - c),
              [x], [a, b, c], StartingValues = [1, 2, 3],
              CovarianceMatrix)
```

$$\left[[1.004,\ 1.998,\ 3.0],\ 107.4,\ \begin{pmatrix} 0.00003334 & -0.0000003414 & -0.00000006749 \\ -0.0000003414 & 0.00006697 & 0.0000001379 \\ -0.00000006749 & 0.0000001379 & 0.00001663 \end{pmatrix} \right]$$

The correlation matrix of the parameters a, b, c is obtained via `stats::correlationMatrix` applied to the covariance matrix `cov` returned by `stats::reg`:

```
stats::correlationMatrix(cov)
```

$$\begin{pmatrix} 1.0 & -0.007226 & -0.002866 \\ -0.007226 & 1.0 & 0.004132 \\ -0.002866 & 0.004132 & 1.0 \end{pmatrix}$$

```
delete r, xdata, ydata, weights, DIGITS, abc, chisquare, cov:
```

Parameters

`cov`

The covariance matrix: a square matrix of category `Cat::Matrix`, or an array.

Return Values

Matrix of the same dimension and type as the input matrix `cov`. `FAIL` is returned if at least one of the diagonal elements of the input matrix `cov` is zero.

See Also

MuPAD Functions

`stats::correlation` | `stats::covariance` | `stats::reg` | `stats::stdev`

stats::covariance

Covariance of data samples

Syntax

```
stats::covariance([x₁, x₂, …], [y₁, y₂, …], <Sample | Population>)
stats::covariance([[x₁, y₁], [x₂, y₂], …], <Sample | Population>)
stats::covariance(s, <c₁, c₂>, <Sample | Population>)
stats::covariance(s, <[c₁, c₂]>, <Sample | Population>)
stats::covariance(s₁, <c₁>, s₂, <c₂>, <Sample | Population>)
```

Description

`stats::covariance([x₁, x₂, …, xₙ], [y₁, y₂, …, yₙ])` returns the covariance

$$\frac{1}{n-1}\left(\sum_{i=1}^{n}(x_i-\overline{x})\,(y_i-\overline{y})\right)=\frac{1}{n-1}\left(\sum_{i=1}^{n}x_i\,y_i\right)-\frac{n}{n-1}\,\overline{x}\,\overline{y}\,,$$

where $\overline{x}$ and $\overline{y}$ are the arithmetic means of the data x_i and y_i, respectively.

`stats::covariance([x₁, x₂, …, xₙ], [y₁, y₂, …, yₙ], Population)` returns

$$\frac{1}{n}\left(\sum_{i=1}^{n}(x_i-\overline{x})\,(y_i-\overline{y})\right)=\frac{1}{n}\left(\sum_{i=1}^{n}x_i\,y_i\right)-\overline{x}\,\overline{y}\,.$$

If the input data are floating-point numbers, the sums defining the covariance are computed in a numerically stable way. If a floating point result is desired, it is recommended to make sure that all input data are floats.

For exact input data, exact symbolic expressions are returned.

The column indices c_1, c_2 are optional if the data are given by a `stats::sample` object `s` containing only two non-string data columns. If the data are provided by two samples s_1, s_2, the column indices are optional for samples containing only one non-string data column.

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

Examples

Example 1

We compute the covariance of samples passed as lists:

```
X := [2, 33/7, 21/9, PI]: Y :=  [3, 5, 1, 7]:
stats::covariance(X, Y)
```

$$\pi - \frac{10}{7}$$

Alternatively, the data may be passed as a list of data pairs:

```
stats::covariance([[2, 3], [33/7, 5], [21/9, 1], [PI, 7]])
```

$$\pi - \frac{10}{7}$$

If all data are floating-point numbers, the result is a float:

```
stats::covariance(float(X), float(Y))
```

$$1.713021225$$

```
delete X, Y:
```

Example 2

We create a sample of type `stats::sample`:

```
s := stats::sample([[1.0, 2.4, 3.0],
                    [7.0, 4.8, 4.0],
                    [3.3, 3.0, 5.0]])
```

```
1.0  2.4  3.0
7.0  4.8  4.0
3.3  3.0  5.0
```

We compute the covariance of the first column and the third column in several equivalent ways:

```
stats::covariance(s, 1, 3),
stats::covariance(s, [1, 3]),
stats::covariance(s, 1, s, 3)
```

$$1.15,\ 1.15,\ 1.15$$

```
delete s:
```

Example 3

The covariance of symbolic data is returned as a symbolic expression:

```
stats::covariance([x1, x2], [y1, y2])
```

$$x1\ y1 + x2\ y2 - 2\left(\frac{x1}{2} + \frac{x2}{2}\right)\left(\frac{y1}{2} + \frac{y2}{2}\right)$$

```
expand(%)
```

$$\frac{x1\ y1}{2} - \frac{x1\ y2}{2} - \frac{x2\ y1}{2} + \frac{x2\ y2}{2}$$

Parameters

x_1, y_1, x_2, y_2, …

The statistical data: arithmetical expressions. The number of data x_i must coincide with the number of data y_i.

`s`, `s`$_1$, `s`$_2$

Samples of type stats::sample

`c`$_1$, `c`$_2$

Column indices: positive integers. Column `c`$_1$ of `s` or `s`$_1$, respectively, provides the data x_i. Column `c`$_2$ of `s` or `s`$_2$, respectively, provides the data y_i.

Options

`Sample`

The data are regarded as a “sample”, not as a full population. This is the default.

`Population`

The data are regarded as the whole population, not as a sample.

Return Values

arithmetical expression.

See Also

MuPAD Functions

`stats::correlation` | `stats::correlationMatrix` | `stats::stdev`

stats::cutoff

Discard outliers

Syntax

```
stats::cutoff([x1, x2, …], α)

stats::cutoff([[x11, x12, …], [x21, x22, …], …], α, i)

stats::cutoff(s, α, i)
```

Description

`stats::cutoff([x1, x2, …], α)` returns those elements of [x_1, x_2, …] larger than the α quantile and smaller than the 1 - α quantile of this list.

`stats::cutoff([[x11, x12, …], [x21, x22, …], …], α, i)` and `stats::cutoff(stats::sample([[x11, x12, …], [x21, x22, …], …]), α, i)` perform the operations described above on the i-th entries of the input rows.

Measurement data often contains "outliers," sample points rather far outside the range containing the majority of the points. While expected both from theory and experience, these outliers, for small or medium-sized samples, tend to distort statistical data such as the mean value.

One of the standard methods dealing with this problem for (real) continuous scales is discarding the outliers. `stats::cutoff` discards all data points below or above a given quantile.

Examples

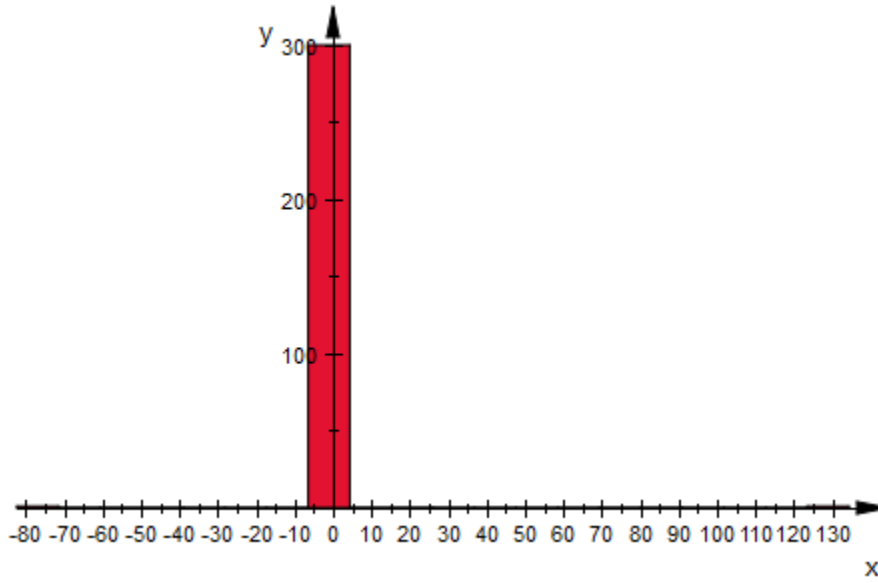
Example 1

We create a normally distributed sample, slightly contaminated:

```
r := stats::normalRandom(0, 1, Seed=2):
data := [r() $ i = 1..300, 100*r() $ i = 1..2]:
```

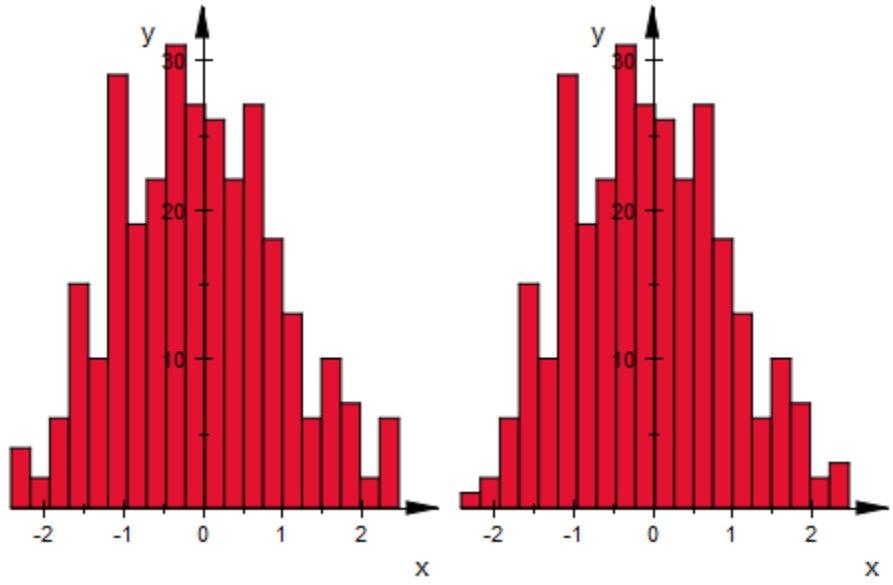
The two extra points distort the data significantly:

```
plot(plot::Histogram2d(data, Cells=20))
```



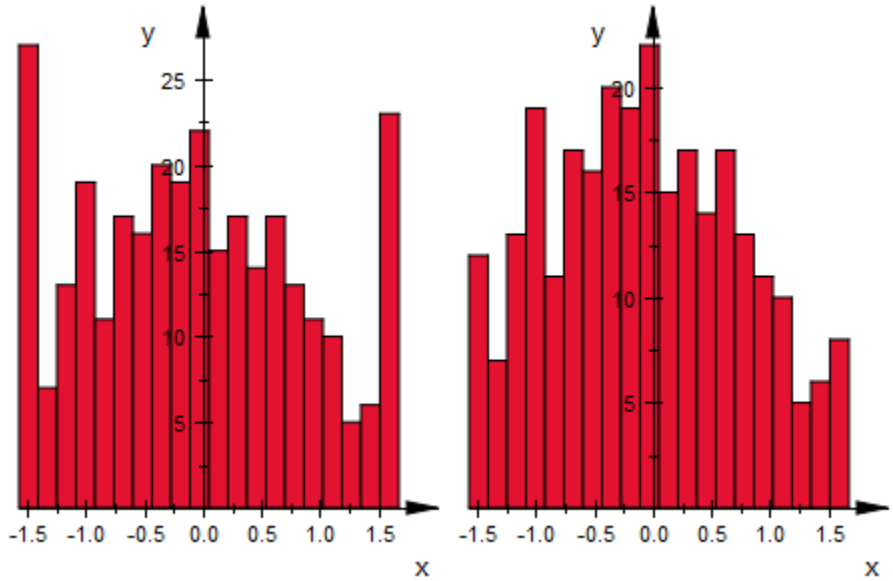
Using either `stats::winsorize` or `stats::cutoff` removes this noise and the image shows more detail:

```
plot(plot::Scene2d(plot::Histogram2d
         (stats::winsorize(data, 1/100), Cells=20)),
     plot::Scene2d(plot::Histogram2d
         (stats::cutoff(data, 1/100), Cells=20)))
```



With larger values of α, the difference between the two is easier to see:

```
plot(plot::Scene2d(plot::Histogram2d
         (stats::winsorize(data, 1/20), Cells=20)),
     plot::Scene2d(plot::Histogram2d
         (stats::cutoff(data, 1/20), Cells=20)))
```



Both `stats::winsorize` and `stats::cutoff` reduce the standard deviation of the sample. This effect is considerably stronger for `stats::cutoff`, though. Keeping in mind that the standard deviation of our random number generator is 1, we compute that of the data in its various forms:

```
stats::stdev(data),
stats::stdev(stats::winsorize(data, 1/20)),
stats::stdev(stats::cutoff(data, 1/20))
```

```
9.133931298, 0.9276576788, 0.8142529511
```

Parameters

x_1, x_2, x_{11}, …

The statistical data: arithmetical expressions. The data to filter on must be real-valued.

s

Sample of type `stats::sample`

α

Cutoff parameter: a real-valued expression $0 \leq \alpha \leq \frac{1}{2}$.

i

Column index: positive integer. The nested list or the sample is filtered on its *i*-th column.

Return Values

The input data with outliers being removed.

See Also

MuPAD Functions

`stats::sample` | `stats::winsorize`

More About

- "Handle Outliers"

stats::winsorize

Clamp (winsorize) extremal values

Syntax

```
stats::winsorize([x1, x2, …], α)
```

```
stats::winsorize([[x11, x12, …], [x21, x22, …], …], α, i)
```

```
stats::winsorize(s, α, i)
```

Description

`stats::winsorize([x1, x2, …], α)` returns a copy of [x_1, x_2, …] in which all entries smaller than the a quantile have been replaced by this value and likewise for all entries larger than the 1 - a quantile.

`stats::winsorize([[x11, x12, …], [x21, x22, …], …], α, i)` and `stats::winsorize(stats::sample([[x11, x12, …], [x21, x22, …], …]), α, i)` perform the operations described above on the i-th entries of the input rows.

Measurement data often contains "outliers," sample points rather far outside the range containing the majority of the points. While expected both from theory and experience, these outliers, for small or medium-sized samples, tend to distort statistical data such as the mean value.

One of the standard methods dealing with this problem for (real) continuous scales is clamping the outliers. `stats::winsorize` sets all data points below or above a given quantile to these quantiles. (This operation is named after its inventor, Charles P. Winsor.)

Examples

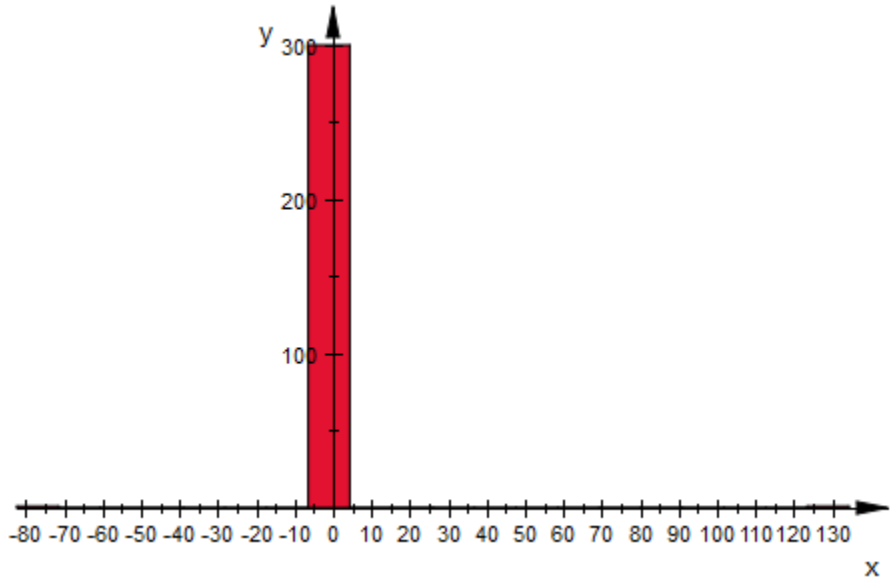
Example 1

We create a normally distributed sample, slightly contaminated:

```
r := stats::normalRandom(0, 1, Seed=2):
data := [r() $ i = 1..300, 100*r() $ i = 1..2]:
```

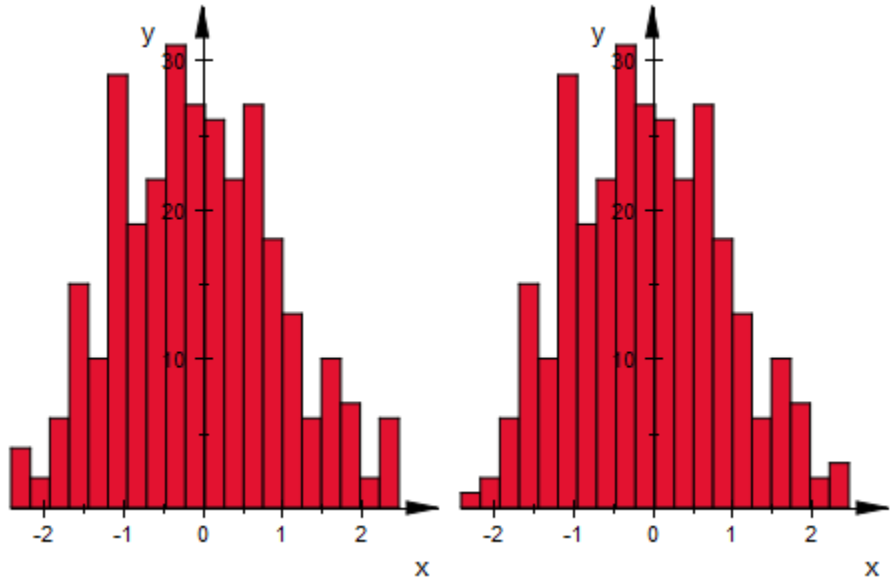
The two extra points distort the data significantly:

```
plot(plot::Histogram2d(data, Cells=20))
```



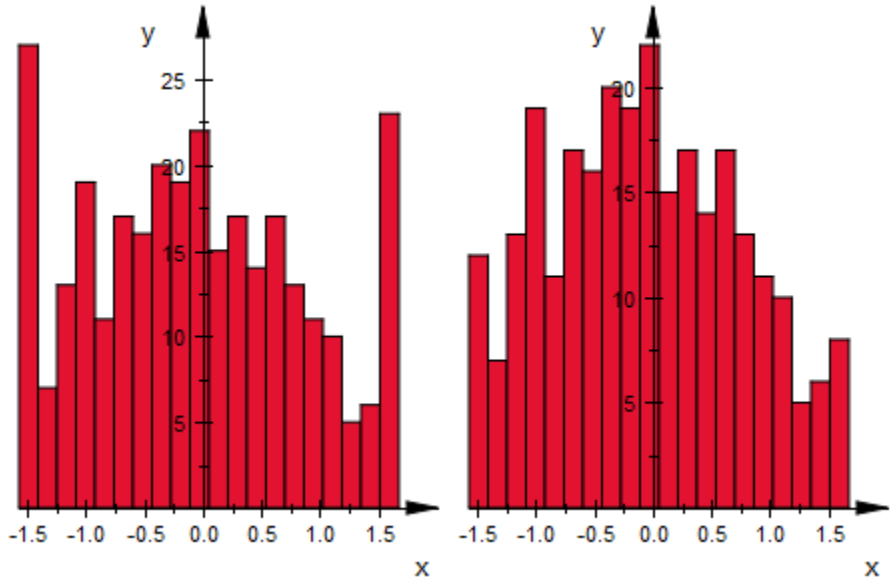
Using either `stats::winsorize` or `stats::cutoff` removes this noise and the image shows more detail:

```
plot(plot::Scene2d(plot::Histogram2d
         (stats::winsorize(data, 1/100), Cells=20)),
     plot::Scene2d(plot::Histogram2d
         (stats::cutoff(data, 1/100), Cells=20)))
```



With larger values of α, the difference between the two is easier to see:

```
plot(plot::Scene2d(plot::Histogram2d
         (stats::winsorize(data, 1/20), Cells=20)),
     plot::Scene2d(plot::Histogram2d
         (stats::cutoff(data, 1/20), Cells=20)))
```



Both `stats::winsorize` and `stats::cutoff` reduce the standard deviation of the sample. This effect is considerably stronger for `stats::cutoff`, though. Keeping in mind that the standard deviation of our random number generator is 1, we compute that of the data in its various forms:

```
stats::stdev(data),
stats::stdev(stats::winsorize(data, 1/20)),
stats::stdev(stats::cutoff(data, 1/20))
```

```
9.133931298, 0.9276576788, 0.8142529511
```

Parameters

x_1, x_2, x_{11}, …

The statistical data: arithmetical expressions. The data to filter on must be real-valued.

s

Sample of type `stats::sample`

α

Cut-off parameter: a real-valued expression $0 \le \alpha \le \frac{1}{2}$.

i

Column index: positive integer. The nested list or the sample is winsorized on its *i*-th column.

Return Values

The input data with outliers being replaced by the values of quantiles.

See Also

MuPAD Functions

`stats::cutoff` | `stats::sample`

More About

- "Handle Outliers"

stats::csGOFT

Classical chi-square goodness-of-fit test

Syntax

```
stats::csGOFT(x1, x2, …, [[a1, b1], [a2, b2], …], CDF = f | PDF = f | PF = f)
```

```
stats::csGOFT([x1, x2, …], [[a1, b1], [a2, b2], …], CDF = f | PDF = f | PF = f)
```

```
stats::csGOFT(s, <c>, [[a1, b1], [a2, b2], …], CDF = f | PDF = f | PF = f)
```

Description

`stats::csGOFT(data, cells, CDF = f)` applies the classical chi-square goodness-of-fit test for the null hypothesis: "the data are f-distributed".

The chi-square goodness-of-fit test divides the real line into k intervals $c_i = (a_i, b_i]$ ('the cells'). It computes the number of data x_j falling into the cells c_i and compares these 'empirical cell frequencies' with the 'expected cell frequencies' $n\,p_i$, where n is the sample size and $p_i = Pr(a_i < x \le b_i)$ are the 'cell probabilities' of a random variable with the hypothesized distribution specified by `X = f`.

All data x_1, x_2 etc. must be convertible to real floating-point numbers. The data do not have to be sorted on input: `stats::csGOFT` automatically converts the data to floats and sorts them internally.

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

Finite cell boundaries a_i, b_i must be convertible to real floating-point numbers satisfying $a_1 < b_1 \le a_2 < b_2 \le a_3 < \ldots$. They define *semiopen* intervalls $c_i = (a_i, b_i]$.

When the hypothesized distribution f is specified as a cumulative distribution function (`CDF = f`), the left boundary of the first cell and the right boundary of the last cell are ignored. They are replaced by $-\infty$ and *infinity*, respectively, i.e., the cell partitioning

$$(-\infty, b_1], (a_2, b_2], \ldots, \left(a_{k-1}, b_{k-1}\right], (a_k, \infty)$$

is used internally.

The cells must be disjoint. Their union must cover the support area of the distribution, i.e., the 'cell probabilities' $p_i = Pr(a_i < x \le b_i)$ must add up to 1 for a random variable x with the hypothesized distribution given by `f`. For continuous distributions, adjacent cells with $b_1 = a_2$, $b_2 = a_3$, … are appropriate.

You may use $a_1 = -\infty$ and $b_k = \infty$ for distributions supported on the entire real line.

Note: The cells must be chosen such that no cell probability p_i vanishes!

See the `Background' section of this help page for recommendations on the cell partitioning. In particular, the use of equiprobable cells (with constant $p_i$) is recommended. For convenience, a utility function `stats::equiprobableCells` is provided to generate such cells. See “Example 1” on page 30-100, “Example 3” on page 30-103, and “Example 4” on page 30-106.

The distribution the data are tested for is specified by the equation `X = f`, where `X` is one of the flags `CDF`, `PDF` or `PF`.

For efficiency, it is recommended to specify a cumulative distribution function (`CDF = f`).

The function `f` can be a procedure provided by the MuPAD `stats` library. Specifications such as `CDF = stats::normalCDF(m, v)` or `CDF = stats::poissonCDF(m)` with suitable numerical values of `m`, `v` are possible and recommended.

Distributions that are not provided by the `stats`-package can be implemented easily by the user. A user defined procedure *f* can implement any distribution function. In the CDF case, `stats::csGOFT` calls `f` with the boundary values a_i, b_i of the cells to compute the cell probabilities via $p_i = f(b_i) - f(a_i)$ (automatically setting $f(a_1) = 0$ and $f(b_k) = 1$).

The function `f` must return a numerical real value between 0 and 1. See “Example 5” on page 30-107 and “Example 6” on page 30-108.

Alternatively, the function `f` can be specified by a univariate arithmetical expression $g(x)$ depending on a symbolic variable `x`. It is interpreted as the function $f: x \to g(x)$. Cf. “Example 6” on page 30-108.

See the `Background' section of this help page for further information on the specification of the distribution via `CDF = f`, `PDF = f` or `PF = f`.

The call `stats::csGOFT(data, cells, X = f)` returns the list [*PValue* = *p*, *StatValue* = *s*, *MinimalExpectedCellFrequency* = *m*]:

- `s` is the observed value of the chi-square statistic

$$s = \sum_{i=1}^{k} \frac{(y_i - n\,p_i)^2}{n\,p_i},$$

  where n is the sample size, k is the number of cells, y_i is the observed cell frequency of the data (i.e., $y_i$is the number of data x_j falling into the cell c_i), and p_i is the cell probabilitiy corresponding to the hypothesized distribution f.
- p is the observed significance level of the chi-square statistic with k - 1 degrees of freedom, i.e., p = 1 - *stats*::*chisquareCDF*(k - 1)(s)
- $m = \min(n\,p_i,\ i = 1..k)$ is the minimum of the expected cell frequencies $n\,p_i$. This information is provided by the test to make sure that the boundary conditions for a "reasonable" cell partitioning are met (see the "Background" section of this help page).

The most relevant information returned by `stats::csGOFT` is the observed significance level `PValue = p`. It has to be interpreted in the following way: Under the null hypothesis, the chi-square statistic

$$S = \sum_{i=1}^{k} \frac{(y_i - n\,p_i)^2}{n\,p_i}$$

is approximately chi-square distributed (for large samples):

$$\Pr(S \le s) = \text{stats::chisquareCDF}(k-1)(s).$$

Under the null hypothesis, the probability $p = Pr(S > s)$ should not be small, where `s` is the value of the statistic attained by the sample.

Specifically, $p = Pr(S > s) \ge \alpha$ should hold for a given significance level $0 < \alpha < 1$, If this condition is violated, the hypothesis may be rejected at level α.

Thus, if the PValue (observed significance level) `p` = $Pr(S > s)$ satisfies $p < \alpha$, the sample leading to the observed value `s` of the statistic S represents an unlikely event, and the null hypothesis may be rejected at level α.

On the other hand, values of p close to 1 should raise suspicion about the randomness of the data: they indicate a fit that is *too good*.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We consider random data that should be normally distributed with mean 15 and variance 2:

```
f := stats::normalRandom(15, 2, Seed = 0):
data := [f() $ i = 1..1000]:
```

According to the recommendations in the `Background' section of this help page, the number of cells should be approximately $2\,n^{2/5} \approx 31.7$, where $n = 1000$ is the sample size.

We wish to use 32 cells that are equiprobable with respect to the hypothesized normal distribution. We estimate the mean m and the variance v of the data:

```
[m, v] := [stats::mean(data), stats::variance(data, Sample)]
```

$$[14.94011963, 2.014256118]$$

The utility function `stats::equiprobableCells` is used to compute an equiprobable cell partitioning via the quantile function of the normal distribution with the empirical parameters:

```
cells := stats::equiprobableCells(32, stats::normalQuantile(m, v)):
```

```
stats::csGOFT(data, cells, CDF = stats::normalCDF(m, v))
```

[PValue = 0.6359912784, StatValue = 27.712, MinimalExpectedCellFrequency = 31.25]

The observed significance level 0.635... attained by the sample is not small. Hence, one should not reject the hypothesis that the sample is normally distributed with mean 14.94... and variance 2.014....

In the following, we impurify the sample by appending some uniformly distributed numbers. A new equiprobable cell partitioning appropriate for the new data is computed:

```
r := stats::uniformRandom(10, 20, Seed = 0):
data := append(data, r() $ 40):
[m, v] := [stats::mean(data), stats::variance(data, Sample)]:
k := round(2*nops(data)^(2/5)):
cells := stats::equiprobableCells(k, stats::normalQuantile(m, v)):
stats::csGOFT(data, cells, CDF = stats::normalCDF(m, v))
```

[PValue = 0.007653320709, StatValue = 53.29230769, MinimalExpectedCellFrequency = 32.5]

The impure data may be rejected as a normally distributed sample at levels as small as 0.0076....

```
delete f, data, m, v, k, cells, r:
```

Example 2

We create a sample of random data that should be binomially distributed with trial parameter 70 and probability parameter $\frac{1}{2}$:

```
r := stats::binomialRandom(70, 1/2, Seed = 123):
data := [r() $ k = 1..1000]:
```

With the expectation value of 35 and the standard deviation of $\frac{\sqrt{70}}{2} \approx 4.18$ of this distribution, we expect most of the data to have values between 30 and 40. Thus, a cell partitioning consisting of 12 cells corresponding to the intervals

(0, 30], (30, 31], (31, 32], ..., (39, 40], (40, 70]

should be appropriate. Note that all cells are interpreted as the intervals $(a_i, b_i]$, i.e., the left boundary is not included in the interval. Strictly speaking, the value 0 is not covered by these cells. However, with a `CDF` specification, `stats::csGOFT` ignores the leftmost boundary and replaces it by `-infinity`. Thus, the union of the cells does cover all integers 0, …, 70 that can be attained by the hypothesized binomial distribution with `trial parameter' 70:

```
cells := [[0, 30], [i, i + 1] $ i = 30..39, [40, 70]]
```

```
[[0, 30], [30, 31], [31, 32], [32, 33], [33, 34], [34, 35], [35, 36], [36, 37], [37, 38], [38, 39],
  [39, 40], [40, 70]]
```

We apply the χ^2 test with various specifications of the binomial distribution. They all produce the same result. However, the first call using a `CDF` specification is the most efficient (fastest) call:

```
stats::csGOFT(data, cells, CDF = stats::binomialCDF(70, 1/2));
```

```
[PValue = 0.4810610726, StatValue = 10.55729362,
  MinimalExpectedCellFrequency = 46.88135939]
```

```
stats::csGOFT(data, cells, PF = stats::binomialPF(70, 1/2));
```

```
[PValue = 0.4810610726, StatValue = 10.55729362,
  MinimalExpectedCellFrequency = 46.88135939]
```

```
f := binomial(70, x)*(1/2)^x*(1/2)^(70 - x):
stats::csGOFT(data, cells, PF = f)
```

```
[PValue = 0.4810610726, StatValue = 10.55729362,
  MinimalExpectedCellFrequency = 46.88135939]
```

The observed significance level 0.481… indicates that the data pass the test well.

Next, we dote the sample by appending the value 35 forty times:

```
data := data . [35 $ 40]:
```

```
stats::csGOFT(data, cells, CDF = stats::binomialCDF(70, 1/2));
```

```
[PValue = 0.0098099163, StatValue = 24.78215227,
  MinimalExpectedCellFrequency = 48.75661376]
```

Now, the data may be rejected as a binomial sample with the specified parameters at levels as small as 0.0098....

```
delete r, data, cells, f:
```

Example 3

We test data that purport to be a sample of beta distributed numbers with scale parameters 3 and 2. Since beta deviates attain values between 0 and 1, we choose an equidistant cell partitioning of the interval [0, 1] consisting of 10 cells. Various equivalent calls to `stats::csGOFT` are demonstrated:

```
r := stats::betaRandom(3, 2, Seed = 1):
data := [r() $ i = 1..100]:
cells := [[(i - 1)/10, i/10] $ i = 1..10]:
stats::csGOFT(data, cells, CDF = stats::betaCDF(3, 2));
stats::csGOFT(data, cells, CDF = (x -> stats::betaCDF(3, 2)(x)))
```

```
[PValue = 0.7329969624, StatValue = 6.068961653, MinimalExpectedCellFrequency = 0.37]
```

```
[PValue = 0.7329969624, StatValue = 6.068961653, MinimalExpectedCellFrequency = 0.37]
```

Alternatively, the beta destribution may be passed by a `PDF` specification. This, however, is less efficient than the `CDF` specification used before:

```
stats::csGOFT(data, cells, PDF = stats::betaPDF(3, 2));
stats::csGOFT(data, cells, PDF = (x -> stats::betaPDF(3, 2)(x)));
```

```
[PValue = 0.7329969624, StatValue = 6.068961653, MinimalExpectedCellFrequency = 0.37]
```

```
[PValue = 0.7329969624, StatValue = 6.068961653, MinimalExpectedCellFrequency = 0.37]
```

The observed significance level 0.732... is not small. Hence, this test does not indicate that the data should be rejected as a beta distributed sample with the specified parameters. Note, however, that the minimal expected cell frequency given by the third element of the returned list is rather small. This indicates that the cell partitioning is not very fortunate. We investigate the expected cell frequencies by computing $n\,p_i = n\,(f(b_i) - f(a_i))$, where f is the cumulative distribution function of the beta distribution and n is the sample size:

```
f:= stats::betaCDF(3, 2):
map(cells, cell -> 100*(f(cell[2]) - f(cell[1])))
```

[0.37, 2.35, 5.65, 9.55, 13.33, 16.27, 17.65, 16.75, 12.85, 5.23]

These values show that the first two or three cells should be joined to a single cell. We modify the cell partitioning by joining the first three and the last two cells:

```
cells := [[0, 3/10], [(i - 1)/10, i/10] $ i = 4..8, [8/10, 1]]
```

$$\left[\left[0, \frac{3}{10}\right], \left[\frac{3}{10}, \frac{2}{5}\right], \left[\frac{2}{5}, \frac{1}{2}\right], \left[\frac{1}{2}, \frac{3}{5}\right], \left[\frac{3}{5}, \frac{7}{10}\right], \left[\frac{7}{10}, \frac{4}{5}\right], \left[\frac{4}{5}, 1\right]\right]$$

For this cell partitioning, the expected frequencies in a random sample of size 100 are sufficiently large for all cells:

```
map(cells, cell -> 100*(f(cell[2]) - f(cell[1])))
```

[8.37, 9.55, 13.33, 16.27, 17.65, 16.75, 18.08]

We apply another χ^2 test with this improved partitioning:

```
stats::csGOFT(data, cells, CDF = f)
```

[PValue = 0.9023533657, StatValue = 2.180685972, MinimalExpectedCellFrequency = 8.37]

Again, with the observed significance level 0.902..., the test does not give any hint that the data are not beta distributed with the specified parameters.

Now, we test whether the data can be regarded as being normally distributed. First, we estimate the parameters (mean and variance) required for the normal distribution:

```
[m, v] := [stats::mean(data), stats::variance(data, Sample)]
```

$$[0.6101560142,\ 0.03595065085]$$

The cell partitioning used before was a partitioning of the interval [0, 1], because beta deviates attain values in this interval. Now we construct a partitioning of 7 equiprobable cells using the quantile function of the normal distribution:

```
k := 7:
cells := stats::equiprobableCells(7, stats::normalQuantile(m, v))
```

$$[[-\infty,\ 0.4077376304],\ [0.4077376304,\ 0.5028484001],\ [0.5028484001,\ 0.5760244864],\ [0.5760244864,\ 0.6442875419],\ [0.6442875419,\ 0.7174636283],\ [0.7174636283,\ 0.8125743979],\ [0.8125743979,\ \infty]]$$

Indeed, theses cells are equiprobable:

```
f:= stats::normalCDF(m, v):
map(cells, cell -> f(cell[2]) - f(cell[1]))
```

$$[0.1428571429,\ 0.1428571429,\ 0.1428571429,\ 0.1428571429,\ 0.1428571429,\ 0.1428571429,\ 0.1428571429]$$

We test for normality with the estimated mean and variance:

```
stats::csGOFT(data, cells, CDF = f)
```

$$[\text{PValue} = 0.7118124259,\ \text{StatValue} = 3.74,\ \text{MinimalExpectedCellFrequency} = 14.28571429]$$

With the observed significance level of $0.711\ldots$, the data should not be rejected as a normally distributed sample. We note that the nonparametric Shapiro-Wilk test implemented in `stats::swGOFT` does detect nonnormality of the sample:

```
stats::swGOFT(data)
```

$$[\text{PValue} = 0.04753801335,\ \text{StatValue} = 0.97429647]$$

With the observed significance level of 0.0475..., normality can be rejected at levels as low as 0.0475....

```
delete r, data, cells, f, m, v, k, boundaries:
```

Example 4

We demonstrate the use of samples of type `stats::sample`. We create a sample consisting of one string column and two non-string columns:

```
s := stats::sample(
  [["1996", 1242, 156], ["1997", 1353, 162], ["1998", 1142, 168],
   ["1999", 1201, 182], ["2001", 1201, 190], ["2001", 1201, 190],
   ["2001", 1201, 205], ["2001", 1201, 210], ["2001", 1201, 220],
   ["2001", 1201, 213], ["2001", 1201, 236], ["2001", 1201, 260],
   ["2001", 1201, 198], ["2001", 1201, 236], ["2001", 1201, 245],
   ["2001", 1201, 188], ["2001", 1201, 177], ["2001", 1201, 233],
   ["2001", 1201, 270]])
```

```
"1996"  1242  156
"1997"  1353  162
"1998"  1142  168
"1999"  1201  182
"2001"  1201  190
"2001"  1201  190
"2001"  1201  205
"2001"  1201  210
"2001"  1201  220
"2001"  1201  213
"2001"  1201  236
"2001"  1201  260
"2001"  1201  198
"2001"  1201  236
"2001"  1201  245
"2001"  1201  188
"2001"  1201  177
"2001"  1201  233
"2001"  1201  270
```

We consider the data in the third column. The mean and the variance of these data are computed:

```
[m, v] := float([stats::mean(s, 3),
```

```
                stats::variance(s, 3, Sample)])
```

```
[207.3157895, 1082.672515]
```

We check whether the data of the third column are normally distributed with the empirical mean and variance computed above. We compute an appropriate cell partitioning in the same way as explained in “Example 1” on page 30-100:

```
samplesize := s::dom::size(s):
k := round(2*samplesize^(2/5)):
cells := stats::equiprobableCells(k, stats::normalQuantile(m, v)):
stats::csGOFT(s, 3, cells, CDF = stats::normalCDF(m, v))
```

```
[PValue = 0.9100131461, StatValue = 1.526315789,
  MinimalExpectedCellFrequency = 3.166666667]
```

Thus, the data pass the test.

```
delete s, m, v, samplesize, k, cells:
```

Example 5

We demonstrate how user-defined distribution functions can be used. A die is rolled 60 times. The following frequencies of the scores 1, 2, …, 6 are observed:

```
             score  | 1 |  2 | 3
|  4 | 5 | 6          ----------+---+----+---+----+---+--
   frequency | 7 | 16 | 8 | 17 | 3 | 9
```

We test the null hypothesis that the dice is fair. Under this hypothesis, the variable X given by the score of a single roll attains the values 1 through 6 with constant probability $\frac{1}{6}$. Presently, the `stats`-package does not provide a discrete uniform distribution, so we implement a corresponding cumulative discrete distribution function `f`:

```
f := proc(x)
     begin
          if x < 0 then
               0
          elif x <= 6 then
               trunc(x)/6
          else 1
          end_if;
```

```
    end_proc:
```

We create the data representing the 60 rolls:

```
data := [ 1 $ 7, 2 $ 16, 3 $ 8, 4 $ 17, 5 $ 3, 6 $ 9]:
```

We choose a collection of cells, each of which contains exactly one of the integers 1, …, 6:

Wir wählen sodann eine Zellzerlegung, so dass jede Zelle genau eine der ganzen Zahlen 1, …, 6 enthält:

```
cells := [[i - 1/2, i + 1/2] $ i = 1..6]
```

$$\left[\left[\frac{1}{2}, \frac{3}{2}\right], \left[\frac{3}{2}, \frac{5}{2}\right], \left[\frac{5}{2}, \frac{7}{2}\right], \left[\frac{7}{2}, \frac{9}{2}\right], \left[\frac{9}{2}, \frac{11}{2}\right], \left[\frac{11}{2}, \frac{13}{2}\right]\right]$$

```
stats::csGOFT(data, cells, CDF = f)
```

$$[\text{PValue} = 0.01125197903,\ \text{StatValue} = 14.8,\ \text{MinimalExpectedCellFrequency} = 10.0]$$

At a significance level as small as $0.011\ldots$, the null hypothesis `the dice is fair' should be rejected.

```
delete f, data, cells:
```

Example 6

We give a further demonstration of user-defined distribution functions. The following procedure represents the cumulative distribution function $f: x \to \Pr(X \le x) = x^2$ of a variable X supported on the interval $[0, 1]$. It will be called with values from the cell boundaries and must return numerical values between 0 and 1:

```
f := proc(x)
     begin
       if x <= 0 then return(0)
       elif x <= 1 then return(x^2)
       else return(1)
       end_if
     end_proc:
```

We test the hypothesis that the following data are *f*-distributed. The cells form an equidistant partitioning of the interval [0, 1]:

```
data := [sqrt(frandom()) $ i = 1..10^3]:
k := 10:
cells := [[(i - 1)/k, i/k] $ i = 1..k]:
stats::csGOFT(data, cells, CDF = f)
```

[PValue = 0.5637573509, StatValue = 7.708606165, MinimalExpectedCellFrequency = 10.0]

The test does not disqualify the sample as being *f*-distributed. Indeed, for a uniform deviate Y on the interval [0, 1] (as produced by `frandom`), the cumulative distribution function of $\sqrt{Y}$ is indeed given by *f*.

We note that the previous function yields the correct CDF values for all real arguments. The chosen cell partitioning indicates that only values from the interval $(0, 1]$ are considered. Since `stats::csGOFT` just evaluates the CDF on the cell boundaries to compute the cell probability of the cell $(a, b]$ by *f*(*b*) - *f*(*a*), it suffices to restrict *f* to the interval $(0, 1]$. Hence, for the chosen cells, the symbolic expression `f = x^2` can also be used to specify the distribution:

```
stats::csGOFT(data, cells, CDF = x^2)
```

[PValue = 0.5637573509, StatValue = 7.708606165, MinimalExpectedCellFrequency = 10.0]

```
delete f, data, k, cells:
```

Parameters

x_1, x_2, …

The statistical data: real numerical values

s

A sample of domain type `stats::sample`

c

An integer representing a column index of the sample `s`. This column provides the data x_1, x_2 etc. There is no need to specify a column `c` if the sample has only one column.

a_1, b_1, a_2, b_2, …

Cell boundaries: real numbers satisfying $a_1 < b_1 \le a_2 < b_2 \le a_3 < \ldots$. Also $\pm\infty$ is admitted as a cell boundary. At least 3 cells have to be specified.

f

A procedure representing the hypothesized distribution: either a cumulative distribution function (`CDF = f`), a probability density function (`PDF = f`), or a (discrete) probability function (`PF = f`). Typically, `f` is one of the distribution functions of the `stats` package such as `stats::normalCDF(m, v)` etc. Instead of a procedure, also an arithmetical expression in some indeterminate `x` may be specified which will be interpreted as a function of `x`.

Options

CDF, PDF, PF

This determines how the procedure `f` is interpreted by `stats::csGOFT`.

Return Values

a list of three equations

`[PValue = p, StatValue = s, MinimalExpectedCellFrequency = m]`

with floating-point values `p`, `s`, `m`. See the "Details" section below for the interpretation of these values.

Algorithms

In R.B. D'Agostino and M.A. Stephens, "Goodness-Of-Fit Techniques", Marcel Dekker, 1986, p. 70-71, one finds the following recommendations for choosing the cell partitioning:

- The number of cells used should be approximately $2\,n^{2/5}$, where n is the sample size.
- The cells should have equal probabilities p_i under the hypothesized distribution.

- With equiprobable cells, the average of the expected cell frequencies $n\,p_i$ should be at least 1 when testing at the significance level α = 0.05. For α = 0.01, the average expected cell frequency should be at least 2. When cells are not approximately equiprobable, the average expected cell frequency for the significance levels above should be doubled. For example, the average expected cell frequency at the significance level α = 0.01 should be at least 4.

The distribution function *f* passed to `stats::csGOFT` via `X = f` is only used to compute the cell probabilities $p_i = Pr(a_i < x \le b_i)$ of the cells $c_i = (a_i, b_i]$.

A cumulative distribution function `f` specified by `CDF = f` is used to compute the cell probabilities via $p_i = f(b_i) - f(a_i)$.

A probability density function `f` specified via `PDF = f` is used to compute the cell probabilities via numerical integration: $p_i = \text{numeric::int}(f(x), x = a_i .. b_i)$. This is rather expensive!

A discrete probability function specified via `PF = f` is used to compute the cell probabilities via the summation $p_i = \sum_{x=\lfloor a_i \rfloor + 1}^{b_i} f(x)$.

Note: Thus, with the specification `PF = f`, the distribution is implicitly supposed to be supported on the integers in the cells $c_i = (a_i, b_i]$. Do not use `PF = f` if the discrete probability function is not supported on the integers! Use `CDF = f` with an appropriate (discrete) cumulative distribution function instead!

With the specification `PF = f`, the value -∞ is not admitted for the left boundary a_1 of the first cell $c_1 = Intval([a_1], [b_1])$.

See Also

MuPAD Functions

`stats::equiprobableCells` | `stats::ksGOFT` | `stats::swGOFT` | `stats::tTest`

stats::empiricalCDF

Empirical (discrete) cumulative distribution function of a finite data sample

Syntax

```
stats::empiricalCDF(x1, x2, …)
```

```
stats::empiricalCDF([x1, x2, …])
```

```
stats::empiricalCDF(s, c)
```

Description

`stats::empiricalCDF(x1, x2, …, xn)` returns a procedure representing the empirical (discrete) cumulative distribution function $x \to \frac{1}{n}\left|\left\{i \;\middle|\; x_i \le x\right\}\right|$ (the relative frequency of data elements x_i less than or equal to x).

All data `x1, x2, …` must be convertible to real floating-point numbers.

The procedure `f := stats::empiricalCDF(x1, x2, …)` can be called in the form `f(x)` with an arithmetical expression `x`.

If `x` is a numerical value, `f(x)` returns a rational number from the interval [0, 1].

The call `f(- infinity )` produces 0; the call `f( infinity )` produces 1.

Otherwise, if `x` is a symbolic expression that cannot be converted to a real floating-point number, `f(x)` returns the symbolic call `stats::empiricalCDF([x1, x2, …])(x)` with the data `x1, x2, …` in ascending order.

For a sample of size n, the call `f := stats::empiricalCDF(x1, x2, …)` needs a run time of $O(n\,ln(n))$ due to internal sorting of the data. Each call to `f` needs a run time of $O(ln(n))$. If several evaluations of the distribution function are needed, a calling sequence such as

```
f := stats::empiricalCDF(x1, x2, …); f(a1); f(a2); dots
```

is more efficient than

`stats::empiricalCDF(`x_1`, `x_2`, …)(`a_1`);`

`stats::empiricalCDF(`x_1`, `x_2`, …)(`a_2`);`

`dots`.

`stats::empiricalCDF` is generalized by `stats::finiteCDF`, which allows to specify different probabilities for the elements of the sample. The call `stats::empiricalCDF([`x_1`, …, `x_n`])` corresponds to `stats::finiteCDF([x_1, dots, x_n], [1/n, dots, 1/n])`.

Further, `stats::finiteCDF` does not only allow numerical values `x1, x2, ...`, but arbitrary MuPAD objects.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision. Note, however, that this function is implemented with `option remember`. After the first call it does not react to changes of `DIGITS` unless the input parameters are changed.

Examples

Example 1

We evaluate the empirical distribution function of the data `-1, 0, 2.3, PI, 8` at various points:

```
f := stats::empiricalCDF(-1, 0, 2.3, PI, 8):
f(-infinity), f(-3), f(2.4), f(PI), f(10), f(infinity)
```

$$0,\ 0,\ \frac{3}{5},\ \frac{4}{5},\ 1,\ 1$$

Alternatively, the data may be passed as a list:

```
f := stats::empiricalCDF([-1, 0, 2.3, PI, 8]):
f(-infinity), f(-3), f(2.4), f(PI), f(10), f(infinity)
```

$$0,\ 0,\ \frac{3}{5},\ \frac{4}{5},\ 1,\ 1$$

```
delete f:
```

Example 2

We use a symbolic argument. In the symbolic return value, the input data appear as a sorted list:

```
stats::empiricalCDF(PI, -3, 25, PI, 4/3)(x)
```

$$\text{stats::empiricalCDF}\left(\left[-3,\ \frac{4}{3},\ \pi,\ \pi,\ 25\right]\right)(x)$$

Example 3

We create a sample consisting of one string column and two non-string columns:

```
s := stats::sample(
  [["1996", 1242, PI - 1/2], ["1997", 1353, PI + 0.3],
   ["1998", 1142, PI + 0.5], ["1999", 1201, PI - 1],
   ["2001", 1201, PI]])
```

```
"1996"   1242   PI - 1/2
"1997"   1353   PI + 0.3
"1998"   1142   PI + 0.5
"1999"   1201     PI - 1
"2001"   1201         PI
```

We compute values of the empirical distributions of the data in the second and third column, respectively:

```
f2 := stats::empiricalCDF(s, 2): f2(1000), f2(1200), f2(1201)
```

$$0,\ \frac{1}{5},\ \frac{3}{5}$$

```
f3 := stats::empiricalCDF(s, 3): f3(0.7), f3(3), f3(PI), f3(4)
```

$$0,\ \frac{2}{5},\ \frac{3}{5},\ 1$$

```
delete s, f2, f3:
```

Parameters

x_1, x_2, …

The statistical data: real numerical values

s

A sample of domain type `stats::sample`

c

A column index of the sample `s`: a positive integer. This column provides the data x_1, x_2 etc. There is no need to specify a column number `c` if the sample has only one non-string column.

Return Values

procedure.

See Also

MuPAD Functions

`stats::empiricalPF` | `stats::empiricalQuantile` | `stats::empiricalRandom` | `stats::finiteCDF` | `stats::finitePF` | `stats::finiteQuantile` | `stats::finiteRandom`

stats::empiricalPF

Probability function of a finite data sample

Syntax

```
stats::empiricalPF(x1, x2, …)
```

```
stats::empiricalPF([x1, x2, …])
```

```
stats::empiricalPF(n, <c>)
```

```
stats::empiricalPF(n, <[c]>)
```

Description

`stats::empiricalPF([x1, x2, …, xn])` returns a procedure representing the probability function

$$x \to \begin{cases} \frac{1}{n} & \text{if } x = x_i \\ 0 & \text{otherwise} \end{cases}$$

of the sample given by the data `x1, x2, ….`

The procedure `f := stats::empiricalPF([x1, x2, …])` can be called in the form `f(x)` with an arithmetical expression `x` or sets of lists of such expressions.

If `x` is a numerical expression that is contained in the data `x1, x2, ….`, then the corresponding probability value $\frac{1}{n}$ is returned (n is the size of the sample).

If `x` is a numerical expression that is not contained in the data `x1, x2, ….`, then `0` is returned.

If `x` is a symbolic expression that cannot be converted to a real floating-point number, `f(x)` returns the symbolic call `stats::empiricalPF([x1, x2, …])(x)` with the data `x1, x2, …` in ascending order.

If `x` is a set, the sum of the probability values of its elements is returned.

If `x` is a list, it is treated like a set (i.e., duplicate entries in `x` are eliminated). The sum of the probability values of the elements in `x` is returned.

Duplicate data elements are automatically combined to a single data element, adding up the corresponding probability values. Cf. “Example 4” on page 30-119.

`stats::empiricalPF` is generalized by `stats::finitePF`, which allows to specify different probabilities for the elements of the sample. The call `stats::empiricalPF([x_1, dots, x_n], [1/n, dots, 1/n])` corresponds to `stats::empiricalPF([`x_1`, …, `x_n`])`.

Further, `stats::finitePF` does not only allow numerical values x_1, x_2, …, but arbitrary MuPAD objects.

Examples

Example 1

We demonstrate the basic usage of this function:

```
f := stats::empiricalPF(1, 3, PI, 4.0):
f(0), f(1), f(1.0), f(3), f(PI), f(float(PI)), f(4), f(4.0)
```

$$0,\ \frac{1}{4},\ 0,\ \frac{1}{4},\ \frac{1}{4},\ 0,\ 0,\ \frac{1}{4}$$

Alternatively, the data may be passed as a list:

```
f := stats::empiricalPF(1, 3, PI, 4.0):
f(0), f(1), f(1.0), f(3), f(PI), f(float(PI)), f(4), f(4.0)
```

$$0,\ \frac{1}{4},\ 0,\ \frac{1}{4},\ \frac{1}{4},\ 0,\ 0,\ \frac{1}{4}$$

A symbolic value of the argument in `f` leads to a symbolic return value:

```
f(x)
```

$$\text{stats::empiricalPF}([1, 3, \pi, 4.0])(x)$$

Symbolic data are not accepted:

```
stats::empiricalPF(1, 3, x, 4.0):
```

```
Error: Some data cannot be converted to floating-point numbers. [stats::empiricalPF]
```

```
delete f:
```

Example 2

We create a sample of type `stats::sample` consisting of one string column and two non-string columns:

```
s := stats::sample(
  [["1996", 1242, 2/5],
   ["1997", 1353, 0.1],
   ["1998", 1142, 0.2],
   ["1999", 1201, 0.2],
   ["2001", 1201, 0.1]])
```

```
"1996"  1242  2/5
"1997"  1353  0.1
"1998"  1142  0.2
"1999"  1201  0.2
"2001"  1201  0.1
```

We use the data in the first and third column:

```
f := stats::empiricalPF(s, 2):
f(1242), f(1353), f(1200), f(1201)
```

$$\frac{1}{5}, \frac{1}{5}, 0, \frac{2}{5}$$

```
delete s, f:
```

Example 3

We consider a fair die:

```
f:= stats::empiricalPF([1, 2, 3, 4, 5, 6]):
```

What is the probabiliy that tossing the die produces a score more than or equal to 4?

```
f({4, 5, 6})
```

$$\frac{1}{2}$$

```
delete f:
```

Example 4

Duplicate data elements are automatically combined to a single data element, adding up the corresponding probability values:

```
f:= stats::empiricalPF([1, 2, 1, 1, 2]):
f(1), f(2)
```

$$\frac{3}{5}, \frac{2}{5}$$

```
delete f:
```

Parameters

x_1, x_2, …

The statistical data: real numerical values

s

A sample of domain type `stats::sample`

c

A column index of the sample `s`: a positive integer. This column provides the data x_1, x_2 etc. There is no need to specify a column number `c` if the sample has only one non-string column.

Return Values

procedure.

See Also

MuPAD Functions

`stats::empiricalCDF` | `stats::empiricalQuantile` | `stats::empiricalRandom` | `stats::finiteCDF` | `stats::finitePF` | `stats::finiteQuantile` | `stats::finiteRandom`

stats::empiricalQuantile

Quantile function of the empirical distribution

Syntax

```
stats::empiricalQuantile(x₁, x₂, …)
stats::empiricalQuantile([x₁, x₂, …])
stats::empiricalQuantile(s, c)
```

Description

`stats::empiricalQuantile(x₁, x₂, …)` returns a procedure representing the quantile function of the data `x₁, x₂` etc. It is the (discrete) inverse of the empirical cumulative distribution function `stats::empiricalCDF(x₁, x₂, …)`. For $0 \le x \le 1$, the `x`-quantile $y = stats::empiricalQuantile(x_1, x_2, \ldots)(x)$ is the smallest of the data elements `x₁, x₂, …` satisfying

$$\text{stats::empiricalCDF}(x_1, x_2, \ldots)(y) \ge x$$

.

All data `x₁, x₂, …` must be convertible to real floating-point numbers. The data do not have to be sorted on input.

The procedure `f := stats::empiricalQuantile(x₁, x₂, …)` can be called in the form `f(x)` or `f(x, Averaged)` with an arithmetical expression `x`.

If x is a real number satisfying $0 \le x \le 1$, then `f(x)` returns one of the data elements; `f(x, Averaged)` uses interpolation of adjacent data elements:

The x-quantile of n sorted values $x_1 \le \ldots \le x_n$ is computed as follows.

- `f(x)` returns x_k with `k = ceil(n*x)`.
- `f(x, Averaged)` returns x_k with `k = ceil(n*x)` if $n\,x$ is not an integer. Otherwise, it returns $\frac{\left(x_k + x_{k+1}\right)}{2}$.

If `x` is a symbolic expression that cannot be converted to a real floating-point number, `f( x , <Averaged>)` returns the symbolic call `stats::empiricalQuantile([x1, x2, …])(x, <Averaged>)` with the data `x1, x2, …` in ascending order.

Numerical values of x are only accepted if $0 \le x \le 1$.

$y = stats::empiricalQuantile(x_1, x_2, \ldots)(x)$ satisfies

$$\text{stats::empiricalCDF}(x_1, x_2, \ldots)(z) < x \le \text{stats::empiricalCDF}(x_1, x_2, \ldots)(y)$$

for all data elements z in the sample satisfying $z < y$.

For a sample of size n, the call `f := stats::empiricalQuantile(x1, x2, …)` needs a run time of $O(n\,ln(n))$ due to internal sorting of the data. The costs of a call to `f` are essentially dependent of n. If several evaluations of the quantile function are needed, a calling sequence such as

```
f := stats::empiricalQuantile(x1, x2, …); f(a1); f(a2); dots
```

is more efficient than

```
stats::empiricalQuantile(x1, x2, …)(a1);

stats::empiricalQuantile(x1, x2, …)(a2);

dots.
```

The $\frac{1}{2}$-quantile is called “median”. The function `stats::median` implements this special quantile.

`stats::empiricalQuantile` is generalized by `stats::finiteQuantile`, which allows to specify different probabilities for the elements of the sample. The call `stats::empiricalQuantile([x1, …, xn])` corresponds to `stats::finiteQuantile([x_1, dots, x_n], [1/n, dots, 1/n])`.

Further, `stats::finiteQuantile` does not only allow numerical values `x1, x2, …`, but arbitrary MuPAD objects.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision. Note, however, that this function is implemented with `option remember`. After the first call it does not react to changes of `DIGITS` unless the input parameters are changed.

Examples

Example 1

We compute various quantiles of the data `-1, 0, 0, 2.3, PI, PI, 8`:

```
f := stats::empiricalQuantile(-1, 0, 0, 2.3, PI, PI, 8):
f(0), f(0.1), f(3/10), f(0.5), f(1/sqrt(2)), f(99/100), f(1)
```

$$-1,\ -1,\ 0,\ 2.3,\ \pi,\ 8,\ 8$$

Alternatively, the data may be passed as a list:

```
f := stats::empiricalQuantile([-1, 0, 2.3, PI, 8]):
f(0), f(0.1), f(3/10), f(0.5), f(1/sqrt(2)), f(99/100), f(1)
```

$$-1,\ -1,\ 0,\ 2.3,\ \pi,\ 8,\ 8$$

```
delete f:
```

Example 2

We use a symbolic argument. In the symbolic return value, the input data appear as a sorted list:

```
f := stats::empiricalQuantile(3, 25, PI, 4/3): f(x)
```

$$\text{stats::empiricalQuantile}\left(\left[\frac{4}{3},\ 3,\ \pi,\ 25\right]\right)(x)$$

Numerical values for x are only accepted if $0 \le x \le 1$:

```
f(0.5)
```

$$3$$

```
f(2)
```

```
Error: An argument x with 0 <= x <= 1 is expected. [f]
```

```
delete f:
```

Example 3

We create a sample of type `stats::sample` consisting of one string column and two non-string columns:

```
s := stats::sample(
  [["1996", 1242, PI - 1/2], ["1997", 1353, PI + 0.3],
   ["1998", 1142, PI + 0.5], ["1999", 1201, PI - 1/3],
   ["2001", 1201, PI + 0.5]])
```

```
"1996"  1242  PI - 1/2
"1997"  1353  PI + 0.3
"1998"  1142  PI + 0.5
"1999"  1201  PI - 1/3
"2001"  1201  PI + 0.5
```

We compute quantile values of the data in the second and third column, respectively:

```
f2 := stats::empiricalQuantile(s, 2):
f2(0.1), f2(1/4), f2(0.7), f2(99/100)
```

$$1142,\ 1201,\ 1242,\ 1353$$

```
f3 := stats::empiricalQuantile(s, 3):
f3(0.1), f3(1/4), f3(0.7), f3(99/100)
```

$$\pi - \frac{1}{2},\ \pi - \frac{1}{3},\ \pi + 0.5,\ \pi + 0.5$$

```
delete s, f2, f3:
```

Parameters

x_1, x_2, …

The statistical data: real numerical values

s

A sample of domain type `stats::sample`

c

A column index of the sample `s`: a positive integer. This column provides the data x_1, x_2 etc. There is no need to specify a column number `c` if the sample has only one non-string column.

Return Values

procedure.

See Also

MuPAD Functions

`stats::empiricalCDF` | `stats::empiricalPF` | `stats::empiricalRandom` | `stats::finiteCDF` | `stats::finitePF` | `stats::finiteQuantile` | `stats::finiteRandom` | `stats::median`

stats::empiricalRandom

Generate a random generator for uniformly distributed elements of a data sample

Syntax

```
stats::empiricalRandom(x1, x2, …, <Seed = n>)
```

```
stats::empiricalRandom([x1, x2, …], <Seed = n>)
```

```
stats::empiricalRandom(n, <c>, <Seed = n>)
```

```
stats::empiricalRandom(n, <[c]>, <Seed = n>)
```

Description

`stats::empiricalRandom([x1, x2, …, xn])` returns a procedure that picks out random elements from the data `x1, x2` etc.

All data `x1, x2, …` must be convertible to real floating-point numbers.

The procedure `f := stats::empiricalRandom([x1, x2, …])` can be called in the form `f()`. The call `f()` returns one of the data elements `x1, x2, ….`

The values produced by `f()` are distributed randomly. Each element of the sample is chosen with the same probability.

Without the option `Seed = n`, an initial seed is chosen internally. This initial seed is set to a default value when MuPAD is started. Thus, each time MuPAD is started or re-initialized with the `reset` function, random generators produce the same sequences of numbers.

Note: In contrast to the function `random`, the generators produced by `stats::empiricalRandom` do not react to the environment variable `SEED`.

For efficiency, it is recommended to produce sequences of K random elements via

```
f := stats::empiricalRandom([x_1, x_2, …]):

f() $k = 1..K;
```

rather than by

```
stats::empiricalRandom([x_1, x_2, dots])() $k = 1..K;
```

The latter call produces a sequence of generators each of which is called once. Also note that

```
stats::empiricalRandom([x_1, x_2, dots], Seed = s)() $k = 1..K;
```

does not produce a random sequence, because a sequence of freshly initialized generators would be created each of them producing the same number.

`stats::empiricalRandom` is generalized by `stats::finiteRandom`, which allows to specify different probabilities for the elements of the sample. The call `stats::empiricalRandom([x_1, …, x_n])` corresponds to `stats::finiteRandom([x_1, dots, x_n], [1/n, dots, 1/n])`.

Further, `stats::finiteRandom` does not only allow numerical values `x_1, x_2, …`, but arbitrary MuPAD objects.

Examples

Example 1

We pick out random elements of some data:

```
f := stats::empiricalRandom(1, 7, 4, PI, Seed = 234):
f(), f(), f(), f(), f(), f(), f(), f(), f()
```

$$1,\ 1,\ \pi,\ 7,\ 7,\ \pi,\ \pi,\ 7,\ 7$$

Alternatively, the data may be passed as a list:

```
f := stats::empiricalRandom([1, 7, 4, PI], Seed = 234):
f(), f(), f(), f(), f(), f(), f(), f(), f()
```

$$1, 1, \pi, 7, 7, \pi, \pi, 7, 7$$

Symbolic data are not accepted:

```
stats::empiricalRandom(1, 7, 4, x):
```

```
Error: Some data cannot be converted to floating-point numbers. [stats::empiricalRandor
```

```
delete f:
```

Example 2

We create a sample of type `stats::sample` consisting of one string column and two non-string columns:

```
s := stats::sample(
  [["1996", 1242, 2/5],
   ["1997", 1353, 0.1],
   ["1998", 1142, 0.2],
   ["1999", 1201, 0.2],
   ["2001", 1201, 0.1]])
```

```
"1996"  1242  2/5
"1997"  1353  0.1
"1998"  1142  0.2
"1999"  1201  0.2
"2001"  1201  0.1
```

We pick random values using the data in the second and third column, respectively:

```
f := stats::empiricalRandom(s, 2, Seed = 12345):
f(), f(), f(), f(), f(), f(), f()
```

$$1353, 1142, 1142, 1201, 1142, 1142, 1353$$

```
f := stats::empiricalRandom(s, 3, Seed = 12345):
f(), f(), f(), f(), f(), f(), f()
```

$$\frac{2}{5}, 0.1, 0.1, 0.1, 0.1, 0.1, \frac{2}{5}$$

```
delete s, f:
```

Example 3

We toss a fair die:

```
f:= stats::empiricalRandom([1, 2, 3, 4, 5, 6], Seed = 12345):
f(), f(), f(), f(), f(), f(), f(), f(), f(), f()
```

$$5, 1, 2, 3, 1, 1, 6, 3, 1, 2$$

We toss the die 6000 times and count the frequencies of the scores 1 through 6:

```
t := [f() $ k = 1..6000]:
i = nops(select(t, _equal, i)) $ i = 1..6
```

$$1 = 982, 2 = 1006, 3 = 911, 4 = 1037, 5 = 1021, 6 = 1043$$

The routine `stats::finiteRandom` allows to model a loaded die:

```
f:= stats::finiteRandom(
    [[1, 0.1],
     [2, 0.1],
     [3, 0.1],
     [4, 0.1],
     [5, 0.1],
     [6, 0.5]],
     Seed = 12345):
t := [f() $ k = 1..6000]:
i = nops(select(t, _equal, i)) $ i = 1..6
```

$$1 = 572, 2 = 611, 3 = 614, 4 = 548, 5 = 554, 6 = 3101$$

```
delete f, t:
```

Parameters

x_1, x_2, …

The statistical data: real numerical values

s

A sample of domain type `stats::sample`

c

A column index of the sample `s`: a positive integer. This column provides the data x_1, x_2 etc. There is no need to specify a column number `c` if the sample has only one non-string column.

Options

Seed

Option, specified as `Seed = n`

Initializes the random generator with the integer seed `n`. n can also be the option `CurrentTime`, to make the seed depend on the current time.

This option serves for generating generators that return predictable sequences of pseudo-random values. The generator is initialized with the seed `n` which may be an arbitrary integer. Several generators with the same initial seed produce the same sequence of values.

Return Values

procedure.

See Also

MuPAD Functions

`stats::empiricalCDF` | `stats::empiricalPF` | `stats::empiricalQuantile` | `stats::finiteCDF` | `stats::finitePF` | `stats::finiteQuantile` | `stats::finiteRandom` | `stats::median`

stats::equiprobableCells

Divide the real line into equiprobable intervals

Syntax

```
stats::equiprobableCells(k, q, <NoWarning>)
```

Description

`stats::equiprobableCells` is a utility function for the classical chi-square test implemented by `stats::csGOFT`. The call `stats::equiprobableCells(k, q)` creates a list of intervals ("cells") that are equiprobable with respect to the statistical distribution corresponding to the quantile function `q`.

The chi-square goodness-of-fit test needs a cell partitioning of the real line to compare the empirical frequencies of data falling into the cells with the expected frequencies corresponding to a hypothesized statistical distribution. It is recommended to use equiprobable cells in this test. `stats::equiprobableCells` is a utility function to compute such a partitioning.

The cell boundaries b_i of the returned cell partitioning $[[b_0, b_1], \ldots, [b_{k-1}, b_k]]$ are computed via $b_i = \mathrm{float}\left(q\left(\mathrm{float}\left(\frac{i}{k}\right)\right)\right)$. Mathematically, each cell $[b_{i-1}, b_i]$ corresponds to a semi-open interval $\left(b_{i-1}, b_i\right]$.

If `q` is the quantile function of a *continuous* statistical distribution, all cells have the same cell probability $\Pr\left(b_{i-1} < x \le b_i\right) = \frac{1}{k}$.

The function `q` can be a quantile procedure provided by the MuPAD `stats`-library.

Quantile functions not provided by the `stats`-package can be implemented easily by the user. A user defined quantile procedure q can correspond to any statistical distribution. Quantile functions must accept one numerical floating-point parameter x satisfying $0.0 \le x \le 1.0$. The call `q(x)` must produce a real value. In particular, the return values `q(0.0) = -infinity` and `q(1.0) = infinity` are allowed.

Quantile functions must be monotonically increasing. `stats::equiprobableCells` issues warnings if the computed quantile values $b_i = \text{float}\left(q\left(\text{float}\left(\frac{i}{k}\right)\right)\right)$ are not real or $\pm\infty$, or if these values do not increase monotonically.

`stats::equiprobableCells` also accepts quantile functions of *discrete* distributions such as `stats::empiricalQuantile(data)` or `stats::binomialQuantile(n, p)`.

Note: Note, however, that in general, there are no equiprobable cell partitionings for discrete distributions. Consequently, equiprobability of the cells returned by `stats::equiprobableCells` is not guaranteed if q is not a continuous function.

In particular, it may happen for large k, that $q\left(\frac{i-1}{k}\right)$ coincides with $q\left(\frac{i}{k}\right)$, i.e., the corresponding cell is empty. This will always happen, when k exceeds the number of possible discrete values the random variable can attain.

In such a case, a warning is issued. Passing such a cell partitioning to `stats::csGOFT` raises an error.

Further to the examples on this help page, see also the examples on the help page of `stats::csGOFT`.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We divide the real line into 4 intervals that are equiprobable with respect to the standard normal distribution:

```
k:= 4: q := stats::normalQuantile(0, 1):
cells := stats::equiprobableCells(k, q)
```

$$[[-\infty, -0.6744897502], [-0.6744897502, 0.0], [0.0, 0.6744897502], [0.6744897502, \infty]]$$

We check equiprobability by applying the function `stats::normalCDF(0, 1)` to the cell boundaries:

```
cdf := stats::normalCDF(0, 1):
p := map(cells, map, cdf)
```

$$[[0, 0.25], [0.25, 0.5], [0.5, 0.75], [0.75, 1]]$$

The cell probabilities are given by the differences of the CDF function applied to the cell boundaries:

```
(p[i][2] - p[i][1]) $ i = 1..k
```

$$0.25, 0.25, 0.25, 0.25$$

We use these cells for a chi-square test for normality of some random data:

```
r := stats::normalRandom(0, 1, Seed = 0):
data := [r() $ i = 1..1000]:
stats::csGOFT(data, cells, CDF = cdf)
```

[PValue = 0.8398775533, StatValue = 0.84, MinimalExpectedCellFrequency = 250.0]

With the observed significance level 0.839..., the data pass this test well. We experiment with other equiprobable cell partitionings:

```
for k in [20, 30, 40, 50] do
    cells := stats::equiprobableCells(k, q);
    print(stats::csGOFT(data, cells, CDF = cdf));
end_for:
```

[PValue = 0.1798122173, StatValue = 24.44, MinimalExpectedCellFrequency = 50.0]

[PValue = 0.713000696, StatValue = 24.32, MinimalExpectedCellFrequency = 33.33333333]

[PValue = 0.7039429342, StatValue = 33.84, MinimalExpectedCellFrequency = 25.0]

```
[PValue = 0.932574698, StatValue = 35.1, MinimalExpectedCellFrequency = 20.0]
```

```
delete k, cells, p, cdf, r, data:
```

Example 2

We create a sample of 1000 random integers between 0 and 100:

```
SEED := 10^2: r := random(0 .. 100):
data := [r() $ i = 1..1000]:
```

We construct an `equiprobable' cell partitioning of 10 cells using the (discrete) empirical distribution of the data. I.e., each of the following cells should contain approximately the same number of data from the random sample:

```
k := 10:
quantile := stats::empiricalQuantile(data):
cells := stats::equiprobableCells(k, quantile)
```

```
[[0.0, 9.0], [9.0, 19.0], [19.0, 31.0], [31.0, 38.0], [38.0, 49.0], [49.0, 59.0], [59.0, 69.0],
  [69.0, 78.0], [78.0, 91.0], [91.0, 100.0]]
```

For discrete distributions, `equiprobability' can only be achieved approximately. We compute the cell probabilities with respect to the empirical cumulative distribution function (CDF), by subtracting the CDF value of the left boundary from the CDF value of the right boundary:

```
cdf := stats::empiricalCDF(data):
map(cells, cell -> cdf(cell[2]) - cdf(cell[1]))
```

$$\left[\frac{12}{125}, \frac{99}{1000}, \frac{14}{125}, \frac{89}{1000}, \frac{51}{500}, \frac{103}{1000}, \frac{103}{1000}, \frac{23}{250}, \frac{21}{200}, \frac{93}{1000}\right]$$

The actual empirical frequency of the data in each cell is the cell probability times the sample size (1000):

```
map(cells, cell -> 1000*(cdf(cell[2]) - cdf(cell[1])))
```

```
[96, 99, 112, 89, 102, 103, 103, 92, 105, 93]
```

When computing the probability of the cell `[b[i-1], b[i]]` via $cdf(b_i)$ - $cdf(b_{i-1})$, the cell is regarded as the semiopen interval $\left(b_{i-1}, b_i\right]$ mathematically. For this reason, the data points `0` contained in the sample are not counted, and the cell frequencies do not quite add up to the sample size:

```
_plus(op(%))
```

$$994$$

For the Symbol::chi^2 test, this does not matter because it replaces the left boundary of the first cell by `-infinity`, anyway. With an observed significance level of $0.161\ldots$, the data pass the test for a uniform distribution at levels as high as $0.161\ldots$:

```
stats::csGOFT(data, cells, CDF = stats::uniformCDF(0, 100))
```

$$[\text{PValue} = 0.1619543558,\ \text{StatValue} = 13.01443112,\ \text{MinimalExpectedCellFrequency} = 70.0]$$

We test whether the data fit a normal distribution with the empirical mean and variance:

```
[m, v] := [stats::mean(data), stats::variance(data)];
stats::csGOFT(data, cells, CDF = stats::normalCDF(m, v))
```

$$\left[\frac{9863}{200}, \frac{1221413}{1480}\right]$$

$$[\text{PValue} = 0.00000000207226197,\ \text{StatValue} = 59.0175707,\ \text{MinimalExpectedCellFrequency} = 65.3983756]$$

With the observed significance level $0.00000000207\ldots$, the hypothesis of a normal distribution clearly has to be rejected.

```
delete r, data, k, quantile, cells, cdf, m, v:
```

Example 3

We consider a binomial distribution with `trial parameter' $n = 100$ and `probability parameter' $p = \frac{1}{2}$. It is the distribution of the number of successes in $n = 100$

independent Bernoulli experiments, each with success probability $p = \frac{1}{2}$. This random variable can attain the discrete values 0, 1, …, 100. We create a cell partitioning of 4 cells:

```
n := 100: p := 1/2:
quantile := stats::binomialQuantile(n, p):
cells := stats::equiprobableCells(4, quantile)
```

```
[[0.0, 47.0], [47.0, 50.0], [50.0, 53.0], [53.0, 100.0]]
```

Because of discreteness, an exact equiprobable cell partitioning does not exist. We compute the expected cell frequencies in the same way as in the previous example:

```
cdf := stats::binomialCDF(n, p):
map(cells, cell -> n*(cdf(cell[2]) - cdf(cell[1])))
```

```
[30.86497068, 23.11449119, 21.81461745, 24.20592068]
```

We create a random sample and apply the Symbol::chi^2 test:

```
r := stats::binomialRandom(n, p, Seed = 123):
data := [r() $ i = 1..100]:
stats::csGOFT(data, cells, CDF = cdf)
```

```
[PValue = 0.3394635837, StatValue = 3.359377148,
  MinimalExpectedCellFrequency = 21.81461745]
```

The observed significance level 0.339… is not small, i.e., the data pass the test well.

The `trial parameter' $n = 100$ is large enough for the binomial distribution to be approximated by a normal distribution with mean $n\,p$ and variance $n\,p\,(1 - p)$. The data pass the test for a normal distribution, too:

```
cdf := stats::normalCDF(n*p, n*p*(1 - p)):
stats::csGOFT(data, cells, CDF = cdf)
```

```
[PValue = 0.1547938521, StatValue = 5.243756673,
  MinimalExpectedCellFrequency = 22.57468822]
```

We repeat the test with another cell partitioning:

```
quantile :=  stats::normalQuantile(n*p, n*p*(1 - p)):
cells := stats::equiprobableCells(4, quantile)
```

$$[[-\infty, 46.62755125], [46.62755125, 50.0], [50.0, 53.37244875], [53.37244875, \infty]]$$

```
stats::csGOFT(data, cells, CDF = cdf)
```

$$[PValue = 0.1422716505, StatValue = 5.44, MinimalExpectedCellFrequency = 25.0]$$

```
delete k, quantile, cells, cdf, r, data:
```

Example 4

We demonstrate user-defined quantile functions. We consider the following distribution of a random variable X supported on the interval [0, 1]:

$$\Pr(X \le x) = \begin{cases} 0 & \text{if } x < 0 \\ x^2 & \text{if } 0 \le x \le 1 \\ 1 & \text{if } x > 1 \end{cases}$$

The quantile function q is given by $q(x) = \sqrt{x}$ for $0 \le x \le 1$:

```
quantile := x -> sqrt(x):
```

We test the hypothesis that the following data are distributed as defined above.

```
cells := stats::equiprobableCells(6, quantile)
```

$$[[0.0, 0.4082482905], [0.4082482905, 0.5773502692], [0.5773502692, 0.7071067812], [0.7071067812, 0.8164965809], [0.8164965809, 0.9128709292], [0.9128709292, 1.0]]$$

```
data := [sqrt(frandom()) $ i = 1..10^3]:
cdf := proc(x)
       begin
         if x <= 0 then return(0)
         elif x <= 1 then return(x^2)
         else return(1)
         end_if
```

```
        end_proc:
stats::csGOFT(data, cells, CDF = cdf)
```

```
[PValue = 0.2230310886, StatValue = 6.968, MinimalExpectedCellFrequency = 166.6666667]
```

The data pass the test well. In fact, for a uniform deviate Y on the interval [0, 1] (as produced by `frandom`), the cumulative distribution function of $\sqrt{Y}$ is indeed given by *cdf*.

```
delete quantile, cells, data, cdf:
```

Parameters

k

The number of cells: a positive integer

q

A procedure representing a quantile function of a statistical distribution. Typically, `q` is one of the quantile functions of the `stats`-package such as `stats::normalQuantile(m, v)`, `stats::empiricalQuantile(data)` etc. Alternatively, user defined procedures may be passed if the `stats`-package does not provide a suitable quantile function.

Options

NoWarning

`stats::equiprobableCells` issues warnings if the computed cell partitioning is not suitable for `stats::csGOFT`. These warnings may be switched off with this option.

Return Values

List of k "cells"

$$\left[[b_0, b_1], [b_1, b_2], \ldots, \left[b_{k-1}, b_k\right]\right]$$

with floating-point values $b_i = q\left(\frac{i}{k}\right)$, $i = 0, \ldots, k$. This `cell partitioning' is suitable as input parameter for `stats::csGOFT`.

See Also

MuPAD Functions

`stats::csGOFT`

stats::erlangCDF

Cumulative distribution function of the Erlang distribution

Syntax

```
stats::erlangCDF(a, b)
```

Description

`stats::erlangCDF(a, b)` returns a procedure representing the cumulative distribution function

$$x \to \begin{cases} \frac{b^a}{\Gamma(a)} \int_0^x t^{a-1} \, e^{-t\,b} \, dt & \text{if } x > 0 \\ 0 & \text{if } x \le 0 \end{cases}$$

of the Erlang distribution with shape parameter $a > 0$ and scale parameter $b > 0$.

The procedure `f := stats::erlangCDF(a, b)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

If $x \le 0$ can be decided, then `f(x)` returns 0. If $x \ge 0$ can be decided, then `f(x)` returns the value $1 - \frac{\Gamma(a, x\,b)}{\Gamma(a)}$.

If x is a floating-point number and both a and b can be converted to positive floating-point numbers, then these values are returned as floating-point numbers. Otherwise, symbolic expressions are returned.

The function `f` reacts to properties of identifiers set via `assume`. If x is a symbolic expression with the property $x \le 0$ or $x \ge 0$, the corresponding values are returned.

The call `f(- infinity )` returns 0.

The call `f( infinity )` returns 1.

`f(x)` returns the symbolic call `stats::erlangCDF(a, b)(x)` if neither $x \leq 0$ nor $x \geq 0$ can be decided.

Numerical values for `a` and `b` are only accepted if they are real and positive.

Note that, for large a, exact results may be costly to compute. If floating-point values are desired, it is recommended to pass floating-point arguments `x` to `f` rather than to compute exact results `f(x)` and convert them via `float`. Cf. “Example 4” on page 30-142.

Note that $\text{stats::erlangCDF}(a, b) = \text{stats::gammaCDF}\left(a, \frac{1}{b}\right)$.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision. The procedure generated by `stats::erlangCDF` reacts to properties of identifiers set via `assume`.

Examples

Example 1

We evaluate the cumulative probability function with $a = 2$ and $b = 1$ at various points:

```
f := stats::erlangCDF(2, 1):
f(-infinity), f(-3), f(0.5), f(2/3), f(PI), f(infinity)
```

$$0,\ 0,\ 0.09020401043,\ 1 - \frac{5\,e^{-\frac{2}{3}}}{3},\ 1 - e^{-\pi}\,(\pi + 1),\ 1$$

```
delete f:
```

Example 2

If `x` is a symbolic object without properties, then it cannot be decided whether $x \geq 0$ holds. A symbolic function call is returned:

```
f := stats::erlangCDF(a, b): f(x)
```

$$\text{stats::erlangCDF}(a, b)(x)$$

With suitable properties, it can be decided whether $x \geq 0$ holds. An explicit expression is returned:

```
assume(0 <= x): f(x)
```

$$1 - \frac{\Gamma(a, b\,x)}{\Gamma(a)}$$

```
unassume(x): delete f:
```

Example 3

We use symbolic arguments:

```
f := stats::erlangCDF(a, b): f(3), f(3.0)
```

$$1 - \frac{\Gamma(a, 3\,b)}{\Gamma(a)},\ 1.0 - \frac{1.0\,\Gamma(a, 3.0\,b)}{\Gamma(a)}$$

When numerical values are assigned to `a` and `b`, the function `f` starts to produce numerical values:

```
a := 2: b := 4: f(3), f(3.0)
```

$$1 - 13\,e^{-12},\ 0.9999201252$$

```
delete f, a, b:
```

Example 4

We consider an Erlang distribution with large shape parameter:

```
f := stats::erlangCDF(2000, 2):
```

For floating-point approximations, one should not compute an exact result and convert it via `float`. For large shape parameter, it is faster to pass a floating-point argument to `f`. The following call takes some time, because an exact computation of the huge integer $\Gamma(2000) = 1999!$ is involved:

```
float(f(1010))
```

```
0.6747900654
```

The following call is much faster:

```
f(float(1010))
```

```
0.6747900654
```

```
delete f:
```

Parameters

a

The shape parameter: an arithmetical expression representing a positive real value

b

The scale parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`gamma` | `igamma` | `stats::erlangPDF` | `stats::erlangQuantile` | `stats::erlangRandom` | `stats::gammaCDF` | `stats::gammaPDF` | `stats::gammaQuantile` | `stats::gammaRandom`

stats::erlangPDF

Probability density function of the Erlang distribution

Syntax

```
stats::erlangPDF(a, b)
```

Description

`stats::erlangPDF(a, b)` returns a procedure representing the probability density function

$$x \to \begin{cases} \frac{b^a}{\Gamma(a)}\, x^{a-1}\, e^{-x\,b} & \text{if } x > 0 \\ 0 & \text{if } x \le 0 \end{cases}$$

of the Erlang distribution with shape parameter $a > 0$ and scale parameter $b > 0$.

The procedure `f := stats::erlangPDF(a, b)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

If $x \le 0$ can be decided, then `f(x)` returns 0. If $x > 0$ can be decided, then `f(x)` returns the value $\frac{x^{a-1}\, b^a}{\Gamma(a)\, e^{x\,b}}$.

If x is a floating-point number and both a and b can be converted to positive floating-point numbers, then these values are returned as floating-point numbers. Otherwise, symbolic expressions are returned.

The function `f` reacts to properties of identifiers set via `assume`. If x is a symbolic expression with the property $x \le 0$ or $x > 0$, the corresponding values are returned.

`f(- infinity )` and `f( infinity )` return 0.

`f(x)` returns the symbolic call `stats::erlangPDF(a, b)(x)` if neither $x \leq 0$ nor $x > 0$ can be decided.

Numerical values for `a` and `b` are only accepted if they are real and positive.

Note that, for large a, exact results may be costly to compute. If floating-point values are desired, it is recommended to pass floating-point arguments `x` to `f` rather than to compute exact results `f(x)` and convert them via `float`. Cf. "Example 4" on page 30-146.

Note that $\text{stats::erlangPDF}(a, b) = \text{stats::gammaPDF}\left(a, \frac{1}{b}\right)$.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision. The procedure generated by `stats::erlangPDF` reacts to properties of identifiers set via `assume`.

Examples

Example 1

We evaluate the probability density function with $a = 2$ and $b = 1$ at various points:

```
f := stats::erlangPDF(2, 1):
f(-infinity), f(-PI), f(1/2), f(0.5), f(PI), f(infinity)
```

$$0,\ 0,\ \frac{e^{-\frac{1}{2}}}{2},\ 0.3032653299,\ \pi\, e^{-\pi},\ 0$$

```
delete f:
```

Example 2

If `x` is a symbolic object without properties, then it cannot be decided whether $x > 0$ holds. A symbolic function call is returned:

```
f := stats::erlangPDF(a, b): f(x)
```

$$\text{stats::erlangPDF}(a, b)(x)$$

With suitable properties, it can be decided whether $x > 0$ holds. An explicit expression is returned:

```
assume(0 < x): f(x)
```

$$\frac{b^a\, x^{a-1}\, e^{-b\,x}}{\Gamma(a)}$$

```
unassume(x): delete f:
```

Example 3

We use symbolic arguments:

```
f := stats::erlangPDF(a, b): f(x), f(3)
```

$$\text{stats::erlangPDF}(a, b)(x),\ \frac{3^{a-1}\, b^a\, e^{-3\,b}}{\Gamma(a)}$$

When numerical values are assigned to `a` and `b`, the function `f` starts to produce numerical values:

```
a := 2: b := 1: f(3), f(3.0)
```

$$3\, e^{-3},\ 0.1493612051$$

```
delete f, a, b:
```

Example 4

We consider an Erlang distribution with large shape parameter:

```
f := stats::erlangPDF(2000, 1):
```

For floating-point approximations, one should not compute an exact result and convert it via `float`. For large shape parameter, it is faster to pass a floating-point argument to `f`. The following call takes some time, because an exact computation of the huge integer $\Gamma(2000) = 1999!$ is involved:

```
float(f(2010))
```

```
0.008657442277
```

The following call is much faster:

```
f(float(2010))
```

```
0.008657442277
```

```
delete f:
```

Parameters

a

The shape parameter: an arithmetical expression representing a positive real value

b

The scale parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`gamma` | `stats::erlangCDF` | `stats::erlangQuantile` | `stats::erlangRandom` | `stats::gammaCDF` | `stats::gammaPDF` | `stats::gammaQuantile` | `stats::gammaRandom`

stats::erlangQuantile

Quantile function of the Erlang distribution

Syntax

```
stats::erlangQuantile(a, b)
```

Description

`stats::erlangQuantile(a, b)` returns a procedure representing the quantile function (inverse) of the cumulative distribution function `stats::erlangCDF(a, b)`. For $0 \le x \le 1$, the solution of *stats*::*erlangCDF*(a, b)(y) = x is given by

$$y = \text{stats::erlangQuantile}(a, b)(x)$$

.

The procedure `f := stats::erlangQuantile(a, b)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number, *infinity*, or a symbolic expression:

If `x` is a real number between 0 and 1 and `a` and `b` can be converted to positive floating-point numbers, then `f(x)` returns a positive floating-point number approximating the solution y of `stats::erlangCDF(a, b)(y) = x`.

The calls `f(0)` and `f(0.0)` produce `0.0` for all values of `a` and `b`.

The calls `f(1)` and `f(1.0)` produce *infinity* for all values of `a` and `b`.

In all other cases, `f(x)` returns the symbolic call `stats::erlangQuantile(a, b)(x)`.

Numerical values of x are only accepted if $0 \le x \le 1$.

Numerical values of `a` and `b` are only accepted if they are real and positive.

Note that $\text{stats::erlangQuantile}(a, b) = \text{stats::gammaQuantile}\left(a, \frac{1}{b}\right)$.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the quantile function with $a = \pi$ and $b = 11$ at various points:

```
f := stats::erlangQuantile(PI, 1/11):
f(0), f(1/10), f(0.5), f(1 - 10^(-10)), f(1)
```

$$0.0,\ 13.08489993,\ 30.96813726,\ 324.7230043,\ \infty$$

The value `f(x)` satisfies `stats::erlangCDF(PI, 1/11)(f(x)) = x`:

```
stats::erlangCDF(PI, 1/11)(f(0.987654))
```

$$0.987654$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::erlangQuantile(a, b): f(x), f(9/10)
```

$$\text{stats::erlangQuantile}(a, b)(x),\ \text{stats::erlangQuantile}(a, b)\left(\frac{9}{10}\right)$$

When positive real values are assigned to `a` and `b`, the function `f` starts to produce floating-point values:

```
a := 17: b := 1/6: f(0.999), f(1 - sqrt(2)/10^5)
```

```
      195.7416524, 240.0294477
```

Numerical values for x are only accepted if $0 \le x \le 1$:

```
f(0.5)
```

```
      100.0071221
```

```
f(2)
```

```
Error: An argument x with 0 <= x <= 1 is expected. [f]
```

```
delete f, a, b:
```

Parameters

a

The shape parameter: an arithmetical expression representing a positive real value

b

The scale parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`stats::erlangCDF` | `stats::erlangPDF` | `stats::erlangRandom` | `stats::gammaCDF` | `stats::gammaPDF` | `stats::gammaQuantile` | `stats::gammaRandom`

stats::erlangRandom

Generate a random number generator for Erlang deviates

Syntax

```
stats::erlangRandom(a, b, <Seed = n>)
```

Description

`stats::erlangRandom(a, b)` returns a procedure that produces Erlang deviates (random numbers) with shape parameter $a > 0$ and scale parameter $b > 0$.

The procedure `f := stats::erlangRandom(a, b)` can be called in the form `f()`. The return value of `f()` is either a floating-point number or a symbolic expression:

If `a` and `b` can be converted to positive floating-point numbers, then `f()` returns a nonnegative floating-point number.

In all other cases, `stats::erlangRandom(a, b)()` is returned symbolically.

Numerical values of `a` and `b` are only accepted if they are real and positive.

The values `X = f()` are distributed randomly according to the cumulative distribution function of the Erlang distribution with parameters a and b. For any $0 \le x$, the probability that $X \le x$ is given by

$$\frac{b^a}{\Gamma(a)} \int_0^x t^{a-1} \, \mathrm{e}^{-t\,b} \, \mathrm{d}t$$

.

Without the option `Seed = n`, an initial seed is chosen internally. This initial seed is set to a default value when MuPAD is started. Thus, each time MuPAD is started or re-initialized with the `reset` function, random generators produce the same sequences of numbers.

Note: In contrast to the function `random`, the generators produced by `stats::erlangRandom` do not react to the environment variable `SEED`.

For efficiency, it is recommended to produce sequences of K random numbers via

```
f := stats::erlangRandom(a, b): f() $k = 1..K;
```

rather than by

```
stats::erlangRandom(a, b)() $k = 1..K;
```

The latter call produces a sequence of generators each of which is called once. Also note that

```
stats::erlangRandom(a, b, Seed = n)() $k = 1..K;
```

does not produce a random sequence, because a sequence of freshly initialized generators would be created each of them producing the same number.

Note that `stats::erlangRandom(a, b)` = `stats::gammaRandom(a, 1/b)`.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We generate Erlang deviates with parameters $a = 2$ and $b = \frac{3}{4}$:

```
f := stats::erlangRandom(2, 3/4): f() $ k = 1..4
```

```
3.958784095, 3.891811185, 6.046842446, 3.142485711
```

```
delete f:
```

Example 2

With symbolic parameters, no random floating-point numbers can be produced:

```
f := stats::erlangRandom(a, b): f()
```

$$\text{stats::erlangRandom}(a, b)()$$

When positive real numbers are assigned to `a` and `b`, the function `f` starts to produce random floating point numbers:

```
a := PI: b := 1/8: f() $ k = 1..4
```

$$19.74371462,\ 12.37357049,\ 13.40137346,\ 29.97534861$$

```
delete f, a, b:
```

Example 3

We use the option `Seed = n` to reproduce a sequence of random numbers:

```
f := stats::erlangRandom(PI, 3, Seed = 1): f() $ k = 1..4
```

$$0.125771079,\ 1.179788536,\ 0.7213738523,\ 1.268143263$$

```
g := stats::erlangRandom(PI, 3, Seed = 1): g() $ k = 1..4
```

$$0.125771079,\ 1.179788536,\ 0.7213738523,\ 1.268143263$$

```
f() = g(), f() = g()
```

$$1.926015116 = 1.926015116,\ 1.1178812 = 1.1178812$$

```
delete f, g:
```

Parameters

a

The shape parameter: an arithmetical expression representing a positive real value

b

The scale parameter: an arithmetical expression representing a positive real value

Options

Seed

Option, specified as `Seed = n`

Initializes the random generator with the integer seed `n`. n can also be the option `CurrentTime`, to make the seed depend on the current time.

This option serves for generating generators that return predictable sequences of pseudo-random numbers. The generator is initialized with the seed `n` which may be an arbitrary integer. Several generators with the same initial seed produce the same sequence of numbers.

When this option is used, the parameters `a` and `b` must be convertible to positive floating-point numbers at the time when the random generator is generated.

Return Values

procedure.

See Also

MuPAD Functions

`stats::erlangCDF` | `stats::erlangPDF` | `stats::erlangQuantile` | `stats::gammaCDF` | `stats::gammaPDF` | `stats::gammaQuantile` | `stats::gammaRandom`

stats::exponentialCDF

Cumulative distribution function of the exponential distribution

Syntax

```
stats::exponentialCDF(a, b)
```

Description

`stats::exponentialCDF(a, b)` returns a procedure representing the cumulative distribution function

$$x \to \begin{cases} 1 - e^{-b\,(x-a)} & \text{if } x > a \\ 0 & \text{if } x \le 0 \end{cases}$$

of the exponential distribution with real location parameter a and scale parameter $b > 0$.

The procedure `f := stats::exponentialCDF(a, b)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

If $x \le a$ can be decided, then `f(x)` returns 0. If $x > a$ can be decided, then `f(x)` returns the value $1 - e^{b\,(a-x)}$.

If x is a floating-point number and both a and b can be converted to floating-point numbers, then these values are returned as floating-point numbers. Otherwise, symbolic expressions are returned.

The function `f` reacts to properties of identifiers set via `assume`. If x is a symbolic expression with the property $x \le a$ or $x \le a$, the corresponding values are returned.

`f(x)` returns the symbolic call `stats::exponentialCDF(a, b)(x)` if neither $x \le a$ nor $x > a$ can be decided.

Numerical values for a and b are only accepted if they are real and b is positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the cumulative distribution function with $a = 0$ and $b = 1$ at various points:

```
f := stats::exponentialCDF(0, 1):
f(-infinity), f(-PI), f(1/2), f(0.5), f(PI), f(infinity)
```

$$0,\ 0,\ 1 - e^{-\frac{1}{2}},\ 0.3934693403,\ 1 - e^{-\pi},\ 1$$

```
delete f:
```

Example 2

If `a` or `x` are symbolic objects without properties, then it cannot be decided whether $x \geq a$ holds. A symbolic function call is returned:

```
f := stats::exponentialCDF(a, b): f(x)
```

$$\text{stats::exponentialCDF}(a, b)(x)$$

With suitable properties, it can be decided whether $x \geq a$ holds. An explicit expression is returned:

```
assume(a <= x): f(x)
```

$$1 - e^{b\,(a-x)}$$

Note that `assume(a <= x)` attached properties both to `a` and `x`. When cleaning up, the properties have to be removed separately for `a` and `x` via `unassume`:

```
unassume(a): unassume(x): delete f:
```

Example 3

We use symbolic arguments:

```
f := stats::exponentialCDF(a, b): f(x)
```

$$\text{stats::exponentialCDF}(a, b)(x)$$

When numerical values are assigned to a and b, the function `f` starts to produce numerical values:

```
a := 0: b := 2: f(3), f(3.0)
```

$$1 - e^{-6},\ 0.9975212478$$

```
delete f, a, b:
```

Parameters

a

The location parameter: an arithmetical expression representing a real value

b

The scale parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`exp` | `stats::exponentialPDF` | `stats::exponentialQuantile` | `stats::exponentialRandom`

stats::exponentialPDF

Probability density function of the exponential distribution

Syntax

```
stats::exponentialPDF(a, b)
```

Description

`stats::exponentialPDF(a, b)` returns a procedure representing the probability density function

$$x \to \begin{cases} b\, e^{-b\,(x-a)} & \text{if } x \geq a \\ 0 & \text{if } x < a \end{cases}$$

of the exponential distribution with real location parameter a and scale parameter $b > 0$.

The procedure `f := stats::exponentialPDF(a, b)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

If $x < a$ can be decided, then `f(x)` returns 0. If $x \geq a$ can be decided, then `f(x)` returns the value $b\, e^{b\,(a-x)}$.

If x is a floating-point number and both a and b can be converted to floating-point numbers, then these values are returned as floating-point numbers. Otherwise, symbolic expressions are returned.

The function `f` reacts to properties of identifiers set via `assume`. If x is a symbolic expression with the property $x < a$ or $x \geq a$, the corresponding values are returned.

`f(x)` returns the symbolic call `stats::exponentialPDF(a, b)(x)` if neither $x < a$ nor $x \geq a$ can be decided.

Numerical values for a and b are only accepted if they are real and b is positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the probability density function with $a = 0$ and $b = 1$ at various points:

```
f := stats::exponentialPDF(0, 1):
f(-infinity), f(-PI), f(1/2), f(0.5), f(PI), f(infinity)
```

$$0,\ 0,\ e^{-\frac{1}{2}},\ 0.6065306597,\ e^{-\pi},\ 0$$

```
delete f:
```

Example 2

If `a` or `x` are symbolic objects without properties, then it cannot be decided whether $x \geq a$ holds. A symbolic function call is returned:

```
f := stats::exponentialPDF(a, b): f(x)
```

$$\text{stats::exponentialPDF}(a,\ b)(x)$$

With suitable properties, it can be decided whether $x \geq a$ holds. An explicit expression is returned:

```
assume(a <= x): f(x)
```

$$b\ e^{b\ (a-x)}$$

Note that `assume(a <= x)` attached properties both to `a` and `x`. When cleaning up, the properties have to be removed separately for `a` and `x` via `unassume`:

```
unassume(a): unassume(x): delete f:
```

Example 3

We use symbolic arguments:

```
f := stats::exponentialPDF(a, b): f(x)
```

$$\text{stats::exponentialPDF}(a, b)(x)$$

When numerical values are assigned to a and b, the function `f` starts to produce numerical values:

```
a := 0: b := 2: f(3), f(3.0)
```

$$2\,e^{-6},\ 0.004957504353$$

```
delete f, a, b:
```

Parameters

a

The location parameter: an arithmetical expression representing a real value

b

The scale parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`stats::exponentialCDF` | `stats::exponentialQuantile` | `stats::exponentialRandom`

stats::exponentialQuantile

Quantile function of the exponential distribution

Syntax

```
stats::exponentialQuantile(a, b)
```

Description

`stats::exponentialQuantile(a, b)` returns a procedure representing the quantile function (inverse)

$$x \to a - \frac{\ln(1-x)}{b}$$

of the cumulative distribution function `stats::exponentialCDF(a, b)`. For $0 \le x \le 1$, the solution of *stats*::*exponentialCDF*(a, b)(y) = x is given by

$$y = \text{stats::exponentialQuantile}(a, b)(x)$$.

The procedure `f := stats::exponentialQuantile(a, b)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number, `infinity`, or a symbolic expression:

If `x` is a real floating-point number between 0 and 1 and `a` and `b` can be converted to suitable real floating-point numbers, then `f(x)` returns a floating-point number.

The calls `f(1)` and `f(1.0)` produce *infinity*.

In all other cases, $f(x)$ returns the symbolic expression `a-ln(1-x)/b`.

Numerical values of x are only accepted if $0 \le x \le 1$.

Numerical values of a and b are only accepted if they are real and b is positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`. which determines the numerical working precision.

Examples

Example 1

We evaluate the quantile function with $a = 2$ and $b = 3$ at various points:

```
f := stats::exponentialQuantile(2, 3):
f(0), f(1/10), f(0.5), f(1 - 10^(-10)), f(1)
```

$$2,\ 2 - \frac{\ln\left(\frac{9}{10}\right)}{3},\ 2.23104906,\ \frac{\ln(10000000000)}{3} + 2,\ \infty$$

The value `f(x)` satisfies `stats::exponentialCDF(2, 3)(f(x))` = `x`:

```
stats::exponentialCDF(2, 3)(f(0.987654))
```

$$0.987654$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::exponentialQuantile(a, b): f(x), f(1/3), f(0.4)
```

$$a - \frac{\ln(1-x)}{b},\ a - \frac{\ln\left(\frac{2}{3}\right)}{b},\ a + \frac{0.5108256238}{b}$$

When suitable numerical values are assigned to `a` and `b`, the function `f` starts to produce numerical values:

```
a := 7: b := 1/8: f(0.999), f(999/1000)
```

$$62.26204223,\ 8\ln(1000) + 7$$

Numerical values for x are only accepted if $0 \le x \le 1$:

```
f(0.5)
```

$$12.54517744$$

```
f(2)
```

```
Error: An argument x with 0 <= x <= 1 is expected. [f]
```

```
delete f, a, b:
```

Parameters

a

The location parameter: an arithmetical expression representing a real value

b

The scale parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`stats::exponentialCDF` | `stats::exponentialPDF` | `stats::exponentialRandom`

stats::exponentialRandom

Generate a random number generator for exponential deviates

Syntax

```
stats::exponentialRandom(a, b, <Seed = n>)
```

Description

`stats::exponentialRandom(a, b)` returns a procedure that produces exponential deviates (random numbers) with real location parameter a and scale parameter $b > 0$.

The procedure `f := stats::exponentialRandom(a, b)` can be called in the form `f()`. The return value of `f()` is either a floating-point number or a symbolic expression:

If `a` can be converted to a real floating point number and `b` to a positive floating-point number, then `f()` returns nonnegative floating-point number.

In all other cases, `stats::exponentialRandom(a, b)()` is returned symbolically.

Numerical values of `a` and `b` are only accepted if they are real and `b` is positive.

The values `X = f()` are distributed randomly according to the cumulative distribution function of the exponential distribution with parameters `a` and `b`. For real $x \geq a$, the probability that $X \leq x$ is given by

$$1 - \frac{1}{e^{b\,(-a+x)}}.$$

Without the option `Seed = n`, an initial seed is chosen internally. This initial seed is set to a default value when MuPAD is started. Thus, each time MuPAD is started or re-initialized with the `reset` function, random generators produce the same sequences of numbers.

Note: In contrast to the function `random`, the generators produced by `stats::exponentialRandom` do not react to the environment variable `SEED`.

For efficiency, it is recommended to produce sequences of K random numbers via

```
f := stats::exponentialRandom(a, b): f() $k = 1..K;
```

rather than by

```
stats::exponentialRandom(a, b)() $k = 1..K;
```

The latter call produces a sequence of generators each of which is called once. Also note that

```
stats::exponentialRandom(a, b, Seed = n)() $k = 1..K;
```

does not produce a random sequence, because a sequence of freshly initialized generators would be created each of them producing the same number.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We generate exponential deviates with parameters $a = 2$ and $b = \frac{3}{4}$:

```
f := stats::exponentialRandom(2, 3/4): f() $ k = 1..4
```

```
3.744010213, 2.246774327, 4.501726533, 2.006934293
```

```
delete f:
```

Example 2

With symbolic parameters, no random floating-point numbers can be produced:

```
f := stats::exponentialRandom(a, b): f()
```

stats::exponentialRandom(a, b)()

When a and b evaluate to suitable real numbers, `f` starts to produce random floating-point numbers:

```
a := PI: b := 1/8: f() $ k = 1..4
```

13.72746104, 16.8526844, 9.492707582, 6.241235276

```
delete f, a, b:
```

Example 3

We use the option `Seed = n` to reproduce a sequence of random numbers:

```
f := stats::exponentialRandom(PI, 1/2, Seed = 1): f() $ k = 1..4
```

4.275085081, 3.608946643, 7.462091361, 6.63997707

```
g := stats::exponentialRandom(PI, 1/2, Seed = 1): g() $ k = 1..4
```

4.275085081, 3.608946643, 7.462091361, 6.63997707

```
f() = g(), f() = g()
```

3.504644667 = 3.504644667, 14.68155806 = 14.68155806

```
delete f, g:
```

Parameters

a

The location parameter: an arithmetical expression representing a real value

b

The scale parameter: an arithmetical expression representing a positive real value

Options

Seed

Option, specified as `Seed = n`

Initializes the random generator with the integer seed `n`. n can also be the option `CurrentTime`, to make the seed depend on the current time.

This option serves for generating generators that return predictable sequences of pseudo-random numbers. The generator is initialized with the seed `n` which may be an arbitrary integer. Several generators with the same initial seed produce the same sequence of numbers.

When this option is used, the parameters `a` and `b` must be convertible to suitable floating-point numbers at the time when the random generator is generated.

Return Values

procedure.

Algorithms

The implemented algorithm for the computation of the exponential deviates uses the quantile function of the exponential distribution applied to uniformly distributed random numbers between 0 and 1.

See Also

MuPAD Functions

`stats::exponentialCDF` | `stats::exponentialPDF` | `stats::exponentialQuantile`

stats::fCDF

Cumulative distribution function of Fisher's f-distribution (fratio distribution)

Syntax

```
stats::fCDF(a, b)
```

Description

`stats::fCDF(a, b)` returns a procedure representing the cumulative distribution function

$$x \to \begin{cases} \dfrac{\left(\frac{a}{b}\right)^{a/2}}{\beta\left(\frac{a}{2}, \frac{b}{2}\right)} \displaystyle\int_0^x \frac{t^{\frac{a}{2}-1}}{\left(\frac{a\,t}{b}+1\right)^{\frac{a}{2}+\frac{b}{2}}}\,\mathrm{d}\,t & \text{if } x > 0 \\ 0 & \text{if } x \le 0 \end{cases}$$

of Fisher's f-distribution with shape parameters $a > 0$, $b > 0$.

The procedure `f:=stats::fCDF(a, b)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

If `x` can be converted to a real floating point number and the shape parameters can be converted to positive floating-point numbers, then `f(x)` returns a floating point number between 0.0 and 1.0.

For all values of `a` and `b`, the call `f(x)` returns `0.0` if `x` is a nonpositive numerical value or a symbolic expression with the property $x \le 0$.

The call `f(- infinity )` returns 0.0.

The call `f( infinity )` returns 1.0.

In all other cases, `f(x)` returns the symbolic call `stats::fCDF(a, b)(x)`.

Numerical values for a and b are only accepted if they are real and positive.

Environment Interactions

The function is sensitive to the environment variable DIGITS which determines the numerical working precision. It reacts to properties of identifiers set via assume.

Examples

Example 1

We evaluate the cumulative distribution function with $a = 2$ and $b = 1$ at various points:

```
f := stats::fCDF(2, 1):
f(-infinity), f(-3), f(0.5), f(2/3), f(PI), f(infinity)
```

$$0.0,\ 0.0,\ 0.2928932188,\ 0.3453463293,\ 0.629456397,\ 1.0$$

```
delete f:
```

Example 2

If x is a symbolic object without properties, then it cannot be decided whether $x \leq 0$ holds. A symbolic function call is returned:

```
f := stats::fCDF(a, b): f(x)
```

$$\text{stats::fCDF}(a,\ b)(x)$$

With suitable properties, it can be decided whether $x \leq 0$ holds. The value 0.0 is returned:

```
assume(x <= 0): f(x)
```

$$0.0$$

MuPAD does not provide a special function to represent the cumulative distribution function for positive arguments. A symbolic call is returned:

```
assume(x > 0): f(x)
```

$$\text{stats::fCDF}(a, b)(x)$$

```
unassume(x): delete f:
```

Example 3

We use symbolic arguments:

```
f := stats::fCDF(a, b): f(x), f(2)
```

$$\text{stats::fCDF}(a, b)(x),\ \text{stats::fCDF}(a, b)(2)$$

When numerical values are assigned to `a` and `b`, the function `f` starts to produce floating-point numbers for numerical arguments:

```
a := 2: b := 1: f(2)
```

$$0.5527864045$$

```
delete f, a, b:
```

Parameters

a, b

The shape parameters: arithmetical expressions representing positive real values

Return Values

procedure.

See Also

MuPAD Functions

`stats::fPDF` | `stats::fQuantile` | `stats::fRandom`

stats::fPDF

Probability density function of Fisher's f-distribution (fratio distribution)

Syntax

```
stats::fPDF(a, b)
```

Description

`stats::fPDF(a, b)` returns a procedure representing the probability density function

$$x \to \begin{cases} \dfrac{\left(\frac{a}{b}\right)^{a/2}}{\beta\left(\frac{a}{2},\frac{b}{2}\right)} \dfrac{x^{\frac{a}{2}-1}}{\left(\frac{a\,x}{b}+1\right)^{\frac{a}{2}+\frac{b}{2}}} & \text{if } x > 0 \\ 0 & \text{if } x \le 0 \end{cases}$$

of Fisher's f-distribution with shape parameters $a > 0$, $b > 0$.

The procedure `f:=stats::fPDF(a, b)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

If $x \le 0$ can be decided, then `f(x)` returns 0 If $x > 0$ can be decided, then `f(x)` returns the value

$$\frac{\left(\frac{a}{b}\right)^{a/2}}{\beta\left(\frac{a}{2},\frac{b}{2}\right)} \frac{x^{\frac{a}{2}-1}}{\left(\frac{a\,x}{b}+1\right)^{\frac{a}{2}+\frac{b}{2}}}.$$

If x is a floating-point number and both a and b can be converted to positive floating-point numbers, then these values are returned as floating-point numbers. Otherwise, symbolic expressions are returned.

The function `f` reacts to properties of identifiers set via `assume`. If x is a symbolic expression with the property $x \le 0$ or $x \ge 0$, the corresponding values are returned.

`f(- infinity )` and `f( infinity )` return 0.

`f(x)` returns the symbolic call `stats::fPDF(a, b)(x)` if neither $x \le 0$ nor $x > 0$ can be decided.

Numerical values for `a` and `b` are only accepted if they are real and positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision. It reacts to properties of identifiers set via `assume`.

Examples

Example 1

We evaluate the probability density function with $a = 2$ and $b = 4$ at various points:

```
f := stats::fPDF(2, 4):
f(-infinity), f(-PI), f(1/2), f(0.5), f(PI), f(infinity)
```

$$0,\ 0,\ \frac{64}{125},\ 0.512,\ \frac{1}{\left(\frac{\pi}{2}+1\right)^3},\ 0$$

```
delete f:
```

Example 2

If `x` is a symbolic object without properties, then it cannot be decided whether $x \ge 0$ holds. A symbolic function call is returned:

```
f := stats::fPDF(a, b): f(x)
```

$$\text{stats::fPDF}(a, b)(x)$$

With suitable properties, it can be decided whether $x \geq 0$ holds. An explicit expression is returned:

```
assume(0 <= x): f(x)
```

$$\frac{x^{\frac{a}{2}-1}\left(\frac{a}{b}\right)^{a/2}}{\left(\frac{a\,x}{b}+1\right)^{\frac{a}{2}+\frac{b}{2}}\beta\left(\frac{a}{2}, \frac{b}{2}\right)}$$

```
unassume(x): delete f:
```

Example 3

We use symbolic arguments:

```
f := stats::fPDF(a, b): f(x)
```

$$\text{stats::fPDF}(a, b)(x)$$

When numerical values are assigned to `a` and `b`, the function `f` starts to produce numerical values:

```
a := 2: b := 1: f(3), f(3.0)
```

$$\frac{\sqrt{7}}{49},\ 0.05399492472$$

```
delete f, a, b:
```

Parameters

a, b

The shape parameters: arithmetical expressions representing positive real values

Return Values

procedure.

See Also

MuPAD Functions

`stats::fCDF` | `stats::fQuantile` | `stats::fRandom`

stats::fQuantile

Quantile function of Fisher's f-distribution (fratio distribution)

Syntax

```
stats::fQuantile(a, b)
```

Description

`stats::fQuantile(a, b)` returns a procedure representing the quantile function (inverse) of the cumulative distribution function `stats::fCDF(a, b)`. For $0 \le x \le 1$, the solution of *stats*::*fCDF*(a, b)(y) = x is given by y = *stats*::*fQuantile*(a, b)(x).

The procedure `f:=stats::fQuantile(a, b)` can be called in the form `f(x)` with arithmetical expressions `x`. The return value of `f(x)` is either a floating-point number, *infinity*, or a symbolic expression:

If `x` is a real number between 0 and 1 and `a` and `b` can be converted to positive floating-point numbers, then `f(x)` returns a positive floating-point number approximating the solution y of *stats*::*fCDF*(a, b)(y) = x.

The calls `f(0)` and `f(0.0)` produce 0.0 for all values of `a` and `b`.

The calls `f(1)` and `f(1.0)` produce *infinity* for all values of `a` and `b`.

In all other cases, `f(x)` returns the symbolic call `stats::fQuantile(a, b)(x)`.

Numerical values of x are only accepted if $0 \le x \le 1$.

Numerical values of `a` and `b` are only accepted if they are real and positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision. The procedure generated by `stats::fQuantile` is sensitive to properties of identifiers, which can be set via `assume`.

Examples

Example 1

We evaluate the quantile function with $a = \pi$ and $b = 11$ at various points:

```
f := stats::fQuantile(PI, 11):
f(0), f(1/10), f(0.5), f(1 - 10^(-10)), f(1)
```

$$0.0,\ 0.2017865341,\ 0.8492236618,\ 280.937214,\ \infty$$

The value `f(x)` satisfies $stats::fCDF(\pi, 11)(f(x)) = x$:

```
stats::fCDF(PI, 11)(f(0.987654321))
```

$$0.987654321$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::fQuantile(a, b): f(x), f(9/10)
```

$$\text{stats::fQuantile}(a, b)(x),\ \text{stats::fQuantile}(a, b)\left(\frac{9}{10}\right)$$

When positive real values are assigned to `a` and `b`, the function `f` starts to produce floating-point values:

```
a := 17: b := 6: f(0.999), f(1 - sqrt(2)/10^5)
```

$$17.35343418,\ 75.00107347$$

Numerical values for x are only accepted if $0 \le x \le 1$:

```
f(0.5)
```

```
                                1.077968248
```

```
f(2)
```

```
Error: An argument x with 0 <= x <= 1 is expected. [f]
```

```
delete f, a, b:
```

Parameters

a, b

The shape parameters: arithmetical expressions representing positive real values

Return Values

procedure.

See Also

MuPAD Functions

`stats::fCDF` | `stats::fPDF` | `stats::fRandom`

stats::fRandom

Generate a random number generator for Fisher's f-deviates (fratio deviates)

Syntax

```
stats::fRandom(a, b, <Seed = n>)
```

Description

`stats::fRandom(a, b)` returns a procedure that produces f-deviates (random numbers) with shape parameters $a > 0$, $b > 0$.

The procedure `f:=stats::fRandom(a, b)` can be called in the form `f()`. The return value of `f()` is either a floating-point number or a symbolic expression:

If `a` and `b` can be converted to positive floating-point numbers, then `f()` returns a positive floating-point number.

In all other cases, `stats::fRandom(a, b)()` is returned symbolically.

Numerical values of `a` and `b` are only accepted if they are real and positive.

The values `X = f()` are distributed randomly according to the cumulative distribution function of the f-distribution with shape parameters a and b. For $0 \le x$, the probability that $X \le x$ is given by

$$\frac{\left(\frac{a}{b}\right)^{a/2}}{\beta\left(\frac{a}{2}, \frac{b}{2}\right)} \int_0^x \frac{t^{\frac{a}{2}-1}}{\left(\frac{a\,t}{b}+1\right)^{\frac{a}{2}+\frac{b}{2}}}\,\mathrm{d}t$$

.

Without the option `Seed = n`, an initial seed is chosen internally. This initial seed is set to a default value when MuPAD is started. Thus, each time MuPAD is started or re-initialized with the `reset` function, random generators produce the same sequences of numbers.

Note: In contrast to the function `random`, the generators produced by `stats::fRandom` do not react to the environment variable `SEED`.

For efficiency, it is recommended to produce sequences of K random numbers via

```
f := stats::fRandom(a, b): f() $k = 1..K;
```

rather than by

```
stats::fRandom(a, b)() $k = 1..K;
```

The latter call produces a sequence of generators each of which is called once. Also note that

```
stats::fRandom(a, Seed = n)() $k = 1..K;
```

does not produce a random sequence, because a sequence of freshly initialized generators would be created each of them producing the same number.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We generate f-deviates with shape parameters $a = 2$ and $b = \frac{3}{4}$:

```
f := stats::fRandom(2, 3/4): f() $ k = 1..4
```

```
0.06381499229, 4.951243823, 4.433412266, 2.189546079
```

```
delete f:
```

Example 2

With symbolic parameters, no random floating-point numbers can be produced:

```
f := stats::fRandom(a, b): f()
```

$$\text{stats::fRandom}(a, b)()$$

When the shape parameters evaluate to positive real numbers, `f` starts to produce random floating-point numbers:

```
a := PI: b := 8: f()
```

$$1.641736211$$

```
delete f, a, b:
```

Example 3

We use the option `Seed = n` to reproduce a sequence of random numbers:

```
f := stats::fRandom(4, 5, Seed = 1): f() $ k = 1..4
```

$$0.002660717454,\ 1.586532187,\ 0.6498965358,\ 0.8953358537$$

```
g := stats::fRandom(4, 5, Seed = 1): g() $ k = 1..4
```

$$0.002660717454,\ 1.586532187,\ 0.6498965358,\ 0.8953358537$$

```
f() = g(), f() = g()
```

$$1.672785971 = 1.672785971,\ 0.5207718594 = 0.5207718594$$

```
delete f, g:
```

Parameters

a, b

The shape parameters: arithmetical expressions representing positive real values

Options

Seed

Option, specified as `Seed = n`

Initializes the random generator with the integer seed `n`. n can also be the option `CurrentTime`, to make the seed depend on the current time.

This option serves for generating generators that return predictable sequences of pseudo-random numbers. The generator is initialized with the seed `n` which may be an arbitrary integer. Several generators with the same initial seed produce the same sequence of numbers.

When this option is used, the shape parameters `a` and `b` must be convertible to positive floating-point numbers at the time when the random generator is generated.

Return Values

procedure.

Algorithms

The implemented algorithm uses independent gamma deviates `X` and `Y` to produce an f-deviate $\frac{b}{a}\frac{X}{Y}$. For more information see: D. Knuth, Seminumerical Algorithms (1998), Vol. 2, p. 135.

See Also

MuPAD Functions

`stats::fCDF` | `stats::fPDF` | `stats::fQuantile`

stats::finiteCDF

Cumulative distribution function of a finite sample space

Syntax

```
stats::finiteCDF([x1, x2, …], [p1, p2, …])

stats::finiteCDF([[x1, p1], [x2, p2], …])

stats::finiteCDF(s, <c1, c2>)

stats::finiteCDF(s, <[c1, c2]>)
```

Description

`stats::finiteCDF([x1, x2, …, xn], [p1, p2, …, pn])` returns a procedure representing the cumulative distribution function $x \to \sum_{i=1}^{k} p_i$ of the finite sample space consisting of the data elements `x1, …, xn` with the probabilities `p1, …, pn`. Here, $k = \left|\left\{ j \mid x_j \le x \right\}\right|$, i.e., x_k is the largest element of the data sample less or equal to x (the data elements are assumed to be ordered: $x_1 < x_2 < x_3$ etc.)

The procedure `f := stats::finiteCDF([x1, x2, …], [p1, p2, …])` can be called in the form `f(x)` with an arithmetical expression `x`.

If `x` is a numerical value and the data elements `x1, x2, …` are all numerical, then `f(x)` returns an arithmetical expression (the sum of the probabilities of all data elements smaller or equal to `x`).

The call `f(- infinity )` produces 0; the call `f( infinity )` produces 1.

Otherwise, if `x` is a symbolic expression that cannot be converted to a real floating-point number or if the data `x1, x2, …` contain elements that cannot be converted to real floating-point numbers, then `f(x)` returns the symbolic call `stats::finiteCDF([x1, x2, …], [p1, p2, …])(x)` with the data `x1, x2, …` in ascending order.

If all probability values p_1, p_2, ... are numerical, they must add up to 1. Otherwise, an error is raised.

Duplicate data elements are automatically combined to a single data element, adding up the corresponding probability values. Cf. "Example 5" on page 30-185.

The data elements x_1, x_2, ... are assumed to be in ascending order: $x_1 < x_2 < \ldots$. If all data elements are numerical, they are re-ordered automatically, if they are not ascending. If the data contain symbolic elements that cannot be converted to floating-point numbers, the ordering is assumed implicitly.

`stats::finiteCDF` generalizes `stats::empiricalCDF`, which assumes equiprobable data. For numerical data x_1, x_2, ..., the call `stats::finiteCDF([x_1, dots, x_n], [1/n, dots, 1/n])` corresponds to `stats::empiricalCDF([`x_1, ..., x_n`])`.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision. Note, however, that this function is implemented with `option remember`. After the first call it does not react to changes of `DIGITS` unless the input parameters are changed.

Examples

Example 1

We evaluate the finite distribution function of some numerical data at various points:

```
f := stats::finiteCDF([1, 0, 2.3, PI], [p1, p0, 0.2, 0.3]):
f(-infinity), f(0.1), f(2.3), f(PI), f(10), f(infinity)
```

$$0,\ p0,\ p0+p1+0.2,\ p0+p1+0.5,\ p0+p1+0.5,\ 1$$

Alternatively, the data may be passed as a list:

```
f := stats::finiteCDF([[1, p1], [0, p0], [2.3, 0.2], [PI, 0.3]]):
f(-infinity), f(0.1), f(2.3), f(PI), f(10), f(infinity)
```

$$0,\ p0,\ p0+p1+0.2,\ p0+p1+0.5,\ p0+p1+0.5,\ 1$$

```
delete f:
```

Example 2

We use symbolic arguments. In the symbolic return value, the input data appear as a sorted list:

```
stats::finiteCDF([3, 4, PI], [0.2, 0.5, 0.3])(x)
```

$$\text{stats::finiteCDF}([3,\ \pi,\ 4],\ [0.2,\ 0.3,\ 0.5])(x)$$

If the data contain symbolic elements, the return value is again a symbolic call:

```
stats::finiteCDF([3, x, PI], [0.2, 0.5, 0.3])(0.7)
```

$$\text{stats::finiteCDF}([3,\ x,\ \pi],\ [0.2,\ 0.5,\ 0.3])(0.7)$$

Example 3

We create a sample consisting of one string column and two non-string columns:

```
s := stats::sample(
  [["1996", 1242, 2/5],
   ["1997", 1353, 0.1],
   ["1998", 1142, 0.2],
   ["1999", 1201, 0.2],
   ["2001", 1201, 0.1]])
```

```
"1996"  1242  2/5
"1997"  1353  0.1
"1998"  1142  0.2
"1999"  1201  0.2
"2001"  1201  0.1
```

We compute values of the finite distributions of the data in the second and third column:

```
f := stats::finiteCDF(s, 2, 3):
```

```
f(1000), f(1200), f(1201)
```

$$0,\ 0.2,\ 0.5$$

```
delete s, f:
```

Example 4

If numerical probability values are given, they must add up to 1:

```
f := stats::finiteCDF([Head, TAIL], [0.45, 0.54]):
```

```
Error: The probabilities do not add up to one. [stats::finiteCDF]
```

Symbolic probability values are not checked for consistency:

```
f := stats::finiteCDF([Head, TAIL], [0.45, p]):
f(x)
```

$$\text{stats::finiteCDF}([\text{Head},\ \text{TAIL}],\ [0.45,\ p])(x)$$

However, when the probabilities are set to numerical values, they are checked:

```
p:= 0.7: f(x)
```

```
Error: The probabilities do not add up to one. [f]
```

```
delete f, p:
```

Example 5

Duplicate data elements are automatically combined to a single data element, adding up the corresponding probability values:

```
f:= stats::finiteCDF([x1, x2, x1, x2], [0.1, 0.2, 0.3, 0.4]):
f(3)
```

$$\text{stats::finiteCDF}([x1,\ x2],\ [0.4,\ 0.6])(3)$$

```
x1 := 1: x2 := 3: f(2)
```

```
    0.4
```

```
delete f, x1, x2:
```

Parameters

x_1, x_2, …

The statistical data: arbitrary MuPAD objects

p_1, p_2, …

Probability values: arithmetical expressions

s

A sample of domain type `stats::sample`

c_1, c_2

Column indices of the sample `s`: positive integers. Column c_1 provides the data x_1, x_2 etc. Column c_2 provides the data p_1, p_2 etc. There is no need to specify column numbers if the sample has only two columns.

Return Values

procedure.

See Also

MuPAD Functions

`stats::empiricalCDF` | `stats::empiricalPF` | `stats::empiricalQuantile` | `stats::empiricalRandom` | `stats::finitePF` | `stats::finiteQuantile` | `stats::finiteRandom`

stats::finitePF

Probability function of a finite sample space

Syntax

```
stats::finitePF([x1, x2, …], [p1, p2, …])
```

```
stats::finitePF([[x1, p1], [x2, p2], …])
```

```
stats::finitePF(n, <c1, c2>)
```

```
stats::finitePF(n, <[c1, c2]>)
```

Description

`stats::finitePF([x1, x2, …, xn], [p1, p2, …, pn])` returns a procedure representing the probability function

$$x \to \begin{cases} p_i & \text{if } x = x_i \\ 0 & \text{otherwise} \end{cases}$$

of the sample space given by the data `x1, x2, …` with the probabilities `p1, p2, ….`

The procedure `f := stats::finitePF([x1, x2, …], [p1, p2, …])` can be called in the form `f(x)` with an arithmetical expression `x` or sets of lists of such expressions.

If `x` is an expression that is contained in the data `x1, x2, ….`, then the corresponding probability value is returned.

If `x` is an expression that is not contained in the data `x1, x2, ….`, then `0` is returned.

If `x` is a set, the sum of the probability values of its elements is returned.

If `x` is a list, it is treated like a set (i.e., duplicate entries in `x` are eliminated). The sum of the probability values of the elements in `x` is returned.

If all probability values p_1, p_2, … are numerical, they must add up to 1. Otherwise, an error is raised. Cf. “Example 4” on page 30-189.

Duplicate data elements are automatically combined to a single data element, adding up the corresponding probability values. Cf. “Example 5” on page 30-190.

`stats::finitePF` generalizes `stats::empiricalPF`, which assumes equiprobable data. For numerical data x_1, x_2, …, the call `stats::finitePF([x_1, dots, x_n], [1/n, dots, 1/n])` corresponds to `stats::empiricalPF([`x_1, …, x_n`])`.

Examples

Example 1

We demonstrate the basic usage of this function:

```
f := stats::finitePF([1, x, y, PI], [1/4, px, py, 0.25]):
f(0), f(1), f(1.0), f(x), f(y), f(PI), f(float(PI)), f(10)
```

$$0,\ \frac{1}{4},\ 0,\ px,\ py,\ 0.25,\ 0,\ 0$$

Alternatively, the data may be passed as a list:

```
f := stats::finitePF([[1, 1/4], [x, px], [y, py], [PI, 0.25]]):
f(0), f(1), f(1.0), f(x), f(y), f(PI), f(float(PI)), f(10)
```

$$0,\ \frac{1}{4},\ 0,\ px,\ py,\ 0.25,\ 0,\ 0$$

```
delete f:
```

Example 2

We create a sample of type `stats::sample` consisting of one string column and two non-string columns:

```
s := stats::sample(
  [["1996", 1242, 2/5],
```

```
   ["1997", 1353, 0.1],
   ["1998", 1142, 0.2],
   ["1999", 1201, 0.2],
   ["2001", 1201, 0.1]])
```

```
"1996"  1242  2/5
"1997"  1353  0.1
"1998"  1142  0.2
"1999"  1201  0.2
"2001"  1201  0.1
```

We use the data in the first and third column:

```
f := stats::finitePF(s, 1, 3):
f("1995"), f("1998"), f("2000"), f("2001")
```

$$0,\ 0.2,\ 0,\ 0.1$$

```
delete s, f:
```

Example 3

We consider a loaded die:

```
f:= stats::finitePF([1, 2, 3, 4, 5, 6],
                    [0.1, 0.1, 0.1, 0.2, 0.2, 0.3]):
```

What is the probabiliy that tossing the die produces a score more than or equal to 4?

```
f({4, 5, 6})
```

$$0.7$$

```
delete f:
```

Example 4

The probability values must add up to 1:

```
stats::finitePF([Head, TAIL], [0.45, 0.54]):
```

```
Error: The probabilities do not add up to one. [stats::finitePF]
```

Example 5

Duplicate data elements are automatically combined to a single data element, adding up the corresponding probability values:

```
f:= stats::finitePF([x1, x2, x1, x2], [0.1, 0.2, 0.3, 0.4]):
f(x1), f(x2)
```

$$0.4,\ 0.6$$

```
delete f:
```

Parameters

x_1, x_2, …

The statistical data: arbitrary MuPAD objects

p_1, p_2, …

Probability values: arithmetical expressions

s

A sample of domain type `stats::sample`

c_1, c_2

Column indices of the sample `s`: positive integers. Column c_1 provides the data x_1, x_2 etc. Column c_2 provides the data p_1, p_2 etc. There is no need to specify column numbers if the sample has only two columns.

Return Values

procedure.

See Also

MuPAD Functions

`stats::empiricalCDF` | `stats::empiricalPF` | `stats::empiricalQuantile` | `stats::empiricalRandom` | `stats::finiteCDF` | `stats::finiteQuantile` | `stats::finiteRandom`

stats::finiteQuantile

Quantile function of a finite sample space

Syntax

```
stats::finiteQuantile([x1, x2, …], [p1, p2, …])
stats::finiteQuantile([[x1, p1], [x2, p2], …])
stats::finiteQuantile(s, <c1, c2>)
stats::finiteQuantile(s, <[c1, c2]>)
```

Description

`stats::finiteQuantile([x1, x2, …, xn], [p1, p2, …, pn])` returns a procedure representing the quantile function of the data `x1, x2` etc. with the probabilities `p1, p2` etc. It is the (discrete) inverse of the cumulative distribution function `stats::finiteCDF([x1, x2, …], [p1, p2, …])`. For $0 \le x \le 1$, the `x`-quantile y = *stats*::*finiteQuantile*([x_1, x_2, …], [p_1, p_2, …])(x) is the smallest of the data elements `x1, x2, …` satisfying

$$\text{stats::finiteCDF}([x_1, x_2, \ldots], [p_1, p_2, \ldots])(y) \ge x$$

(The data elements are assumed to be ordered: $x_1 < x_2 < x_3$ etc.)

The procedure `f := stats::finiteQuantile([x1, x2, …], [p1, p2, …])` can be called in the form `f(x)` with an arithmetical expression `x`.

If x is a real number satisfying $0 \le x \le 1$ and all probability values `p1, p2, …` are numerical, then `f(x)` returns one of the data elements `x1, x2, ….`

Otherwise, if `x` is a symbolic expression that cannot be converted to a real floating-point number or if the probabilities `p1, p2, …` contain elements that cannot be converted to real floating-point numbers, then `f(x)` returns the symbolic call `stats::finiteQuantile([x1, x2, …], [p1, p2, …])(x)` with the data `x1, x2, …` in ascending order.

Numerical values of x are only accepted if $0 \le x \le 1$.

If all probability values `p1, p2, ...` are numerical, they must add up to 1. Otherwise, an error is raised.

Duplicate data elements are automatically combined to a single data element, adding up the corresponding probability values. Cf. "Example 5" on page 30-196.

y = *stats*::*finiteQuantile*([x_1, x_2, …], [p_1, p_2, …])(x) satisfies

$$\text{stats::finiteCDF}\left([x_1, x_2, \ldots], [p_1, p_2, \ldots]\right)(x_i) < x \le \text{stats::finiteCDF}\left([x_1, x_2, \ldots], [p_1, p_2, \ldots]\right)(y)$$

for all data elements x_i in the sample satisfying $x_i < y$.

The data elements `x1, x2, ...` are assumed to be in ascending order: `x1 < x2 < ...`. If all data elements are numerical, they are re-ordered automatically, if they are not ascending. If the data contain symbolic elements that cannot be converted to floating-point numbers, the ordering is assumed implicitly.

`stats::finiteQuantile` generalizes `stats::empiricalQuantile`, which assumes equiprobable data. For numerical data `x1, x2, ...`, the call `stats::finiteQuantile([x_1, dots, x_n], [1/n, dots, 1/n])` corresponds to `stats::empiricalQuantile([x1, ..., xn])`.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision. Note, however, that this function is implemented with `option remember`. After the first call it does not react to changes of `DIGITS` unless the input parameters are changed.

Examples

Example 1

We compute various quantiles of some numerical data:

```
f := stats::finiteQuantile([1, x, y, PI],
                           [1/4, 3/8, 1/4, 1/8]):
f(0), f(0.1), f(3/10), f(0.5), f(1/sqrt(2)), f(99/100), f(1)
```

$$1, 1, x, x, y, \pi, \pi$$

Alternatively, the data may be passed as a list:

```
f := stats::finiteQuantile([[1, 1/4], [x, 3/8],
                            [y, 1/4], [PI, 1/8]]):
f(0), f(0.1), f(3/10), f(0.5), f(1/sqrt(2)), f(99/100), f(1)
```

$$1, 1, x, x, y, \pi, \pi$$

```
delete f:
```

Example 2

We use symbolic arguments. In the symbolic return value, the input data appear as a sorted list:

```
f:= stats::finiteQuantile([3, 4, PI], [0.2, 0.5, 0.3]):
f(x)
```

$$\text{stats::finiteQuantile}([3, \pi, 4], [0.2, 0.3, 0.5])(x)$$

Numerical values for x are only accepted if $0 \le x \le 1$:

```
f(0.5)
```

$$\pi$$

```
f(2)
```

```
Error: An argument x with 0 <= x <= 1 is expected. [f]
```

```
delete f:
```

Example 3

We create a sample of type `stats::sample` consisting of one string column and two non-string columns:

```
s := stats::sample(
  [["1996", 1242, 2/5],
   ["1997", 1353, 0.1],
   ["1998", 1142, 0.2],
   ["1999", 1201, 0.2],
   ["2001", 1201, 0.1]])
```

```
"1996"  1242  2/5
"1997"  1353  0.1
"1998"  1142  0.2
"1999"  1201  0.2
"2001"  1201  0.1
```

We compute quantile values of the data in the second and third column:

```
f := stats::finiteQuantile(s, 2, 3):
f(0.1), f(1/4), f(0.7), f(99/100)
```

$$1142,\ 1201,\ 1242,\ 1353$$

```
delete s, f:
```

Example 4

If numerical probability values are given, they must add up to 1:

```
f := stats::finiteQuantile([Head, TAIL], [0.45, 0.54]):
```

```
Error: The probabilities do not add up to one. [stats::finiteQuantile]
```

Symbolic probability values are not checked for consistency:

```
f := stats::finiteQuantile([Head, TAIL], [0.45, p]):
f(x)
```

$$\text{stats::finiteQuantile}([\text{Head}, \text{TAIL}], [0.45, p])(x)$$

However, when the probabilities are set to numerical values, they are checked:

```
p:= 0.7: f(x)
```

```
Error: The probabilities do not add up to one. [f]
```

```
delete f, p:
```

Example 5

Duplicate data elements are automatically combined to a single data element, adding up the corresponding probability values:

```
f:= stats::finiteQuantile([x1, x2, x1, x2], [p1, p2, 0.3, 0.4]):
f(0.5)
```

$$\text{stats::finiteQuantile}([x1, x2], [p1 + 0.3, p2 + 0.4])(0.5)$$

```
p1 := 0.1: p2 := 0.2: f(0.5)
```

$$x2$$

```
delete f, p1, p2:
```

Parameters

x_1, x_2, …

The statistical data: arbitrary MuPAD objects

p_1, p_2, …

Probability values: arithmetical expressions

s

A sample of domain type `stats::sample`

c_1, c_2

Column indices of the sample `s`: positive integers. Column c_1 provides the data x_1, x_2 etc. Column c_2 provides the data p_1, p_2 etc. There is no need to specify column numbers if the sample has only two columns.

Return Values

procedure.

See Also

MuPAD Functions

`stats::empiricalCDF` | `stats::empiricalPF` | `stats::empiricalQuantile` | `stats::empiricalRandom` | `stats::finiteCDF` | `stats::finitePF` | `stats::finiteRandom` | `stats::median`

stats::finiteRandom

Generate a random generator for elements of a finite sample space

Syntax

```
stats::finiteRandom([x1, x2, …], [p1, p2, …], <Seed = n>)

stats::finiteRandom([[x1, p1], [x2, p2], …], <Seed = n>)

stats::finiteRandom(n, <c1, c2>, <Seed = n>)

stats::finiteRandom(n, <[c1, c2]>, <Seed = n>)
```

Description

`stats::finiteRandom([x1, x2, …, xn], [p1, p2, …, pn])` returns a procedure that picks out random elements from the data `x1, x2` etc. The chances of picking out elements are given by the probabilities `p1, p2` etc.

The procedure `f := stats::finiteRandom([x1, x2, …], [p1, p2, …])` can be called in the form `f()`. The call `f()` returns one of the data elements `x1, x2, ….`

The values `X = f()` are distributed randomly according to the discrete distribution function of the sample space, i.e., the probability of $X \leq x$ is given by `stats::finiteCDF([x1, x2, …], [p1, p2, …])(x)`.

All probability values `p1, p2, …` must be convertible to floating-point numbers. They must add up to 1.

Without the option `Seed = n`, an initial seed is chosen internally. This initial seed is set to a default value when MuPAD is started. Thus, each time MuPAD is started or re-initialized with the `reset` function, random generators produce the same sequences of numbers.

Note: In contrast to the function `random`, the generators produced by `stats::finiteRandom` do not react to the environment variable `SEED`.

For efficiency, it is recommended to produce sequences of K random numbers via

```
f := stats::finiteRandom([x_1, x_2, …], [p_1, p_2, …]):

f() $k = 1..K;
```

rather than by

```
stats::finiteRandom([x_1, x_2, dots], [p_1, p_2, dots])() $k = 1..K;
```

The latter call produces a sequence of generators each of which is called once. Also note that

```
stats::finiteRandom([x_1, x_2, dots], [p_1, p_2, dots], Seed = s)()
$k = 1..K;
```

does not produce a random sequence, because a sequence of freshly initialized generators would be created each of them producing the same number.

`stats::finiteRandom` generalizes `stats::empiricalRandom`, which assumes equiprobable data. For numerical data `x_1, x_2, ...`, the call `stats::finiteRandom([x_1, dots, x_n], [1/n, dots, 1/n])` corresponds to `stats::empiricalRandom([x_1, …, x_n])`.

Examples

Example 1

We pick out random elements of some data:

```
f := stats::finiteRandom([1, x, y, PI],
                         [1/4, 3/8, 1/4, 1/8],
                                  Seed = 234):
f(), f(), f(), f(), f(), f(), f(), f(), f()
```

$$1, 1, x, y, y, x, x, y, \pi$$

Alternatively, the data may be passed as a list:

```
f := stats::finiteRandom([[1, 1/4], [x, 3/8],
```

```
                              [y, 1/4], [PI, 1/8]],
                                      Seed = 234):
f(), f(), f(), f(), f(), f(), f(), f(), f()
```

$$1, 1, x, y, y, x, x, y, \pi$$

```
delete f:
```

Example 2

We create a sample of type `stats::sample` consisting of one string column and two non-string columns:

```
s := stats::sample(
  [["1996", 1242, 2/5],
   ["1997", 1353, 0.1],
   ["1998", 1142, 0.2],
   ["1999", 1201, 0.2],
   ["2001", 1201, 0.1]])
```

```
"1996"  1242  2/5
"1997"  1353  0.1
"1998"  1142  0.2
"1999"  1201  0.2
"2001"  1201  0.1
```

We pick random values using the data in the first and third column:

```
f := stats::finiteRandom(s, 1, 3, Seed = 123):
f(), f(), f(), f(), f(), f(), f()
```

"1996", "1998", "1997", "1996", "1998", "1996", "1996"

```
delete s, f:
```

Example 3

We toss a loaded coin:

```
f:= stats::finiteRandom([Head, Tail], [0.4, 0.6], Seed = 123):
f(), f(), f(), f(), f(), f(), f(), f(), f(), f()
```

```
Head, Tail, Tail, Head, Tail, Head, Head, Tail, Head, Tail
```

We toss the coin 10000 times and count the number of Heads and Tails:

```
t := [f() $ k = 1..10^4]:
NumberOfHeads = nops(select(t, _equal, Head)),
NumberOfTails = nops(select(t, _equal, Tail))
```

```
NumberOfHeads = 3975, NumberOfTails = 6025
```

```
delete f, t:
```

Example 4

The probability values must add up to 1:

```
stats::finiteRandom([Head, TAIL], [0.45, 0.54]):
```

```
Error: The probabilities do not add up to one. [stats::finiteRandom]
```

Parameters

x_1, x_2, …

The statistical data: arbitrary MuPAD objects

p_1, p_2, …

Probability values: real numerical values

s

A sample of domain type `stats::sample`

c_1, c_2

Column indices of the sample `s`: positive integers. Column c_1 provides the data x_1, x_2 etc. Column c_2 provides the data p_1, p_2 etc. There is no need to specify column numbers if the sample has only two columns.

Options

`Seed`

Option, specified as `Seed = n`

Initializes the random generator with the integer seed `n`. n can also be the option `CurrentTime`, to make the seed depend on the current time.

This option serves for generating generators that return predictable sequences of pseudo-random values. The generator is initialized with the seed `n` which may be an arbitrary integer. Several generators with the same initial seed produce the same sequence of values.

Return Values

procedure.

Algorithms

The random values are chosen by applying the quantile function to uniformly distributed random numbers between 0 and 1.

See Also

MuPAD Functions

`stats::empiricalCDF` | `stats::empiricalPF` | `stats::empiricalQuantile` | `stats::empiricalRandom` | `stats::finiteCDF` | `stats::finitePF` | `stats::finiteQuantile` | `stats::median`

stats::frequency

Tally numerical data into classes and count frequencies

Syntax

```
stats::frequency(data, <ClassesClosed = Left | Right>)
stats::frequency(data, n, <ClassesClosed = Left | Right>)
stats::frequency(data, [n], <ClassesClosed = Left | Right>)
stats::frequency(data, [a1 .. b1, a2 .. b2, …], <ClassesClosed = Left | Right>)
stats::frequency(data, [[a1, b1], [a2, b2], …], <ClassesClosed = Left | Right>)
stats::frequency(data, Classes = n, <ClassesClosed = Left | Right>)
stats::frequency(data, Classes = [n], <ClassesClosed = Left | Right>)
stats::frequency(data, Classes = [a1 .. b1, a2 .. b2, …], <ClassesClosed = Left | Right>
stats::frequency(data, Classes = [[a1, b1], [a2, b2], …], <ClassesClosed = Left | Right>
stats::frequency(data, Cells = n, <CellsClosed = Left | Right>)
stats::frequency(data, Cells = [n], <CellsClosed = Left | Right>)
stats::frequency(data, Cells = [a1 .. b1, a2 .. b2, …], <CellsClosed = Left | Right>)
stats::frequency(data, Cells = [[a1, b1], [a2, b2], …], <CellsClosed = Left | Right>)
```

Description

`stats::frequency(data, [[a1, b1], [a2, b2], …])` tallies numerical data into different classes given by semiopen intervals $(a_i, b_i]$. It counts how many data elements fall into each class.

All data elements must be real numerical values. Exact numerical values such as π, $\sqrt{3}$ etc. are allowed if they can be converted to real floating-point numbers via `float`. An

error is raised if symbolic data are found that cannot be converted to real floating point numbers.

Note: Note that `stats::frequency` is fast if all data elements are integers, rational numbers, or floating point numbers. Exact numerical values such as π, $\sqrt{3}$ etc. are processed, but have a noticeable impact on the efficiency of `stats::frequency`.

Data given by an array, a table etc. are internally treated like a list containing all operands of the data container. In particular, all rows and columns of arrays, matrices and `stats::sample` objects are taken into account. A `stats::sample` object must not contain any text entries.

For the specification of the classes, `stats::frequency` accepts either a single positive integer (or, equivalently, a list of one positive integer), or a list of classes given as ranges or lists of two elements.

A single integer n in the specification `Classes= n` or `Classes= [n]` is interpreted as "subdivide the range from *min*(*data*) to *max*(*data*) into n classes of equal size". The left border of the first class is set to - ∞.

The classes may be specified directly as in `Classes = [[a1, b1], [a2, b2], …]` or `Classes=[a_1..b_1, a_2..b_2, dots]`.

Note: With the default setting `ClassesClosed = Right`, the i-th class is the *semi-open* interval $(a_i, b_i]$, i.e., a datum x is tallied into the i-th class if $a_i < x \le b_i$ is satisfied.

With `ClassesClosed = Left`, the i-th class is the semi-open interval $[a_i, b_i)$, i.e., a datum x is tallied into the i-th class if $a_i \le x < b_i$ is satisfied.

The class boundaries must be numerical real values satisfying $a_1 \le b_1 \le a_2 \le b_2 \le a_3 \le \ldots$. In most applications, $b_1 = a_2$, $b_2 = a_3$ etc. is appropriate.

Exact values such as π, $\sqrt{3}$ etc. are accepted and processed.

The classes need not cover the entire data range. Data are ignored if they do not fall into one of the specified classes.

If giving classes directly, the leftmost border may be - ∞ and the rightmost border may be *infinity*.

Examples

Example 1

We split the following data into 10 classes of equal size (default). The first class covers the values from - ∞ to 2:

```
data := [0, 1, 2, PI, 4, 5, 6, 7, 7.1, 20]:
T := stats::frequency(data)
```

1	[[−∞, 2], 3, [0, 1, 2]]
2	[[2, 4], 2, [π, 4]]
3	[[4, 6], 2, [5, 6]]
4	[[6, 8], 2, [7, 7.1]]
5	[[8, 10], 0, []]
6	[[10, 12], 0, []]
7	[[12, 14], 0, []]
8	[[14, 16], 0, []]
9	[[16, 18], 0, []]
10	[[18, 20], 1, [20]]

We split the information on the classes into 3 separate tables:

```
TheClasses = map(T, op, 1)
```

TheClasses =

1	$[-\infty, 2]$
2	[2, 4]
3	[4, 6]
4	[6, 8]
5	[8, 10]
6	[10, 12]
7	[12, 14]
8	[14, 16]
9	[16, 18]
10	[18, 20]

```
TheFrequencies = map(T, op, 2)
```

TheFrequencies =

1	3
2	2
3	2
4	2
5	0
6	0
7	0
8	0
9	0
10	1

```
TheValues = map(T, op, 3)
```

TheValues =

1	[0, 1, 2]
2	$[\pi, 4]$
3	[5, 6]
4	[7, 7.1]
5	[]
6	[]
7	[]
8	[]
9	[]
10	[20]

The classes are specified explicitly:

```
classes:= [[0, 5], [5, 10], [10, 20]]:
stats::frequency(data, classes)
```

1	[[0, 5], 5, [1, 2, π, 4, 5]]
2	[[5, 10], 3, [6, 7, 7.1]]
3	[[10, 20], 1, [20]]

Note that the value 0 is not tallied into any of the classes (the first class represents the semi-open interval $(0, 5]$)! In order to include all values, we use $\pm\infty$ as class boundaries:

```
classes:= [[-infinity, 5], [5, 10], [10, infinity]]:
stats::frequency(data, classes)
```

1	[[−∞, 5], 6, [0, 1, 2, π, 4, 5]]
2	[[5, 10], 3, [6, 7, 7.1]]
3	[[10, ∞], 1, [20]]

```
delete data, T, classes:
```

Example 2

We demonstrate the difference between the options `ClassesClosed = Left` and `ClassesClosed = Right`. In the first case, the value 1 is tallied into the second class:

```
stats::frequency([0, 1, 2], Classes = [-infinity..1, 1..infinity],
                            ClassesClosed = Left)
```

1	[[−∞, 1], 1, [0]]
2	[[1, ∞], 2, [1, 2]]

With `ClassesClosed = Right`, the value 1 is tallied into the first class:

```
stats::frequency([0, 1, 2], Classes = [-infinity..1, 1..infinity],
                            ClassesClosed = Right)
```

1	[[−∞, 1], 2, [0, 1]]
2	[[1, ∞], 1, [2]]

The default setting is `ClassesClosed = Right`:

```
stats::frequency([0, 1, 2], Classes = [-infinity..1, 1..infinity])
```

1	[[−∞, 1], 2, [0, 1]]
2	[[1, ∞], 1, [2]]

Example 3

We create a sample of 1000 normally distributed data points:

```
X := stats::normalRandom(0, 10):
data := [X() $ i = 1..1000]:
```

These data are tallied into 5 different classes of equal width:

```
T := stats::frequency(data, 5):
```

We determine the number of data values in each class:

```
for i from 1 to 5 do
    print(Class = T[i][1], NumberOfElements = T[i][2]);
end_for:
```

Class = [−∞, −5.982260368], NumberOfElements = 23

Class = [−5.982260368, −1.635193057], NumberOfElements = 300

Class = [−1.635193057, 2.711874254], NumberOfElements = 485

Class = [2.711874254, 7.058941565], NumberOfElements = 182

Class = [7.058941565, 11.40600888], NumberOfElements = 10

We determine the outliers of the data sample by collecting the values smaller than - 9 and the values larger than 10:

```
classes := [[-infinity, -9], [10, infinity]]:
T := stats::frequency(data, classes);
```

1	[[−∞, −9], 2, [−10.32932768, −9.707360153]]
2	[[10, ∞], 1, [11.40600888]]

```
delete X, data, T, i, classes:
```

Parameters

`data`

The statistical data: a list, a set, a table, an array, a matrix, or an object of type `stats::sample` containing numerical real data values

`n`

The number of classes (cells): a positive integer. If not specified, $n = 10$ is used.

a_1, b_1, a_2, …

The class boundaries: real numerical values satisfying

$$a_1 \le b_1 \le a_2 \le b_2 \le \ldots$$

Also $\pm\infty$ are allowed as class boundaries.

Return Values

table is returned with integer indices from 1 through the number of classes. The i-th entry of the table `T = stats::frequency(data, ...)` is the list `T[i]` = [[a_i, b_i], n_i, [v_1, v_2, …]], where [a_i, b_i] is the i-th class, n_i is the number of data falling in this class, and [v_1, v_2, …] is the sorted list of all data in this class (i.e., $a_i < v_j \le b_j$ for all j from 1 through n_i).

See Also

MuPAD Functions
`stats::mean` | `stats::stdev`

MuPAD Graphical Primitives
`plot::Histogram2d`

stats::gammaCDF

Cumulative distribution function of the gamma distribution

Syntax

```
stats::gammaCDF(a, b)
```

Description

`stats::gammaCDF(a, b)` returns a procedure representing the cumulative distribution function

$$x \to \begin{cases} \frac{1}{b^a\,\Gamma(a)} \int_0^x t^{a-1}\, e^{-\frac{t}{b}}\, dt & \text{if } x > 0 \\ 0 & \text{if } x \le 0 \end{cases}$$

of the gamma distribution with shape parameter $a > 0$ and scale parameter $b > 0$.

The procedure `f:=stats::gammaCDF(a, b)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

If $x \le 0$ can be decided, then `f(x)` returns 0. If $x \ge 0$ can be decided, then `f(x)` returns the value $1 - \frac{\Gamma\left(a, \frac{x}{b}\right)}{\Gamma(a)}$.

If x is a floating-point number and both a and b can be converted to positive floating-point numbers, then these values are returned as floating-point numbers. Otherwise, symbolic expressions are returned.

The function `f` reacts to properties of identifiers set via `assume`. If x is a symbolic expression with the property $x \le 0$ or $x \ge 0$, the corresponding values are returned.

The call `f(- infinity )` returns 0.

The call `f( infinity )` returns 1.

`f(x)` returns the symbolic call `stats::gammaCDF(a, b)(x)` if neither $x \leq 0$ nor $x \geq 0$ can be decided.

Numerical values for `a` and `b` are only accepted if they are real and positive.

Note that, for large a, exact results may be costly to compute. If floating-point values are desired, it is recommended to pass floating-point arguments `x` to `f` rather than to compute exact results `f(x)` and convert them via `float`. Cf. “Example 4” on page 30-213.

Note that $\text{stats::gammaCDF}(a, b) = \text{stats::erlangCDF}\left(a, \frac{1}{b}\right)$.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision. The procedure generated by `stats::gammaCDF` reacts to properties of identifiers set via `assume`.

Examples

Example 1

We evaluate the cumulative distribution function with $a = 2$ and $b = 1$ at various points:

```
f := stats::gammaCDF(2, 1):
f(-infinity), f(-3), f(0.5), f(2/3), f(PI), f(infinity)
```

$$0,\ 0,\ 0.09020401043,\ 1 - \frac{5\,e^{-\frac{2}{3}}}{3},\ 1 - e^{-\pi}\,(\pi + 1),\ 1$$

```
delete f:
```

Example 2

If `x` is a symbolic object without properties, then it cannot be decided whether $x \geq 0$ holds. A symbolic function call is returned:

```
f := stats::gammaCDF(a, b): f(x)
```

$$\text{stats::gammaCDF}(a, b)(x)$$

With suitable properties, it can be decided whether $x \geq 0$ holds. An explicit expression is returned:

```
assume(0 <= x): f(x)
```

$$1 - \frac{\Gamma\left(a, \frac{x}{b}\right)}{\Gamma(a)}$$

```
unassume(x): delete f:
```

Example 3

We use symbolic arguments:

```
f := stats::gammaCDF(a, b): f(x)
```

$$\text{stats::gammaCDF}(a, b)(x)$$

When numerical values are assigned to `a` and `b`, the function `f` starts to produce numerical values:

```
a := 2: b := 4: f(3), f(3.0)
```

$$1 - \frac{7\, e^{-\frac{3}{4}}}{4},\ 0.1733585327$$

```
delete f, a, b:
```

Example 4

We consider a gamma distribution with large shape parameter:

```
f := stats::gammaCDF(2000, 2):
```

For floating-point approximations, one should not compute an exact result and convert it via `float`. For large shape parameter, it is faster to pass a floating-point argument to `f`. The following call takes some time, because an exact computation of the huge integer $\Gamma(2000) = 1999!$ is involved:

```
float(f(4010))
```

```
0.5474266776
```

The following call is much faster:

```
f(float(4010))
```

```
0.5474266776
```

```
delete f:
```

Parameters

a

The shape parameter: an arithmetical expression representing a positive real value

b

The scale parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`gamma` | `igamma` | `stats::erlangCDF` | `stats::erlangPDF` | `stats::erlangQuantile` | `stats::erlangRandom` | `stats::gammaPDF` | `stats::gammaQuantile` | `stats::gammaRandom`

stats::gammaPDF

Probability density function of the gamma distribution

Syntax

```
stats::gammaPDF(a, b)
```

Description

`stats::gammaPDF(a, b)` returns a procedure representing the probability density function

$$x \to \begin{cases} \frac{1}{b^a\,\Gamma(a)}\,x^{a-1}\,e^{-\frac{x}{b}} & \text{if } x > 0 \\ 0 & \text{if } x \le 0 \end{cases}$$

of the gamma distribution with shape parameter $a > 0$ and scale parameter $b > 0$.

The procedure `f:=stats::gammaPDF(a, b)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

If $x \le 0$ can be decided, then `f(x)` returns 0. If $x > 0$ can be decided, then `f(x)` returns the value $\frac{x^{a-1}}{\exp^{x/b}}\,\frac{1}{\Gamma(a)\,b^a}$.

If x is a floating-point number and both a and b can be converted to positive floating-point numbers, then these values are returned as floating-point numbers. Otherwise, symbolic expressions are returned.

The function `f` reacts to properties of identifiers set via `assume`. If x is a symbolic expression with the property $x \le 0$ or $x \ge 0$, the corresponding values are returned.

`f(- infinity )` and `f( infinity )` return 0.

`f(x)` returns the symbolic call `stats::gammaPDF(a, b)(x)` if neither $x \leq 0$ nor $x > 0$ can be decided.

Numerical values for `a` and `b` are only accepted if they are real and positive.

Note that, for large a, exact results may be costly to compute. If floating-point values are desired, it is recommended to pass floating-point arguments `x` to `f` rather than to compute exact results `f(x)` and convert them via `float`. Cf. “Example 4” on page 30-217.

Note that $\text{stats::gammaPDF}(a, b) = \text{stats::erlangPDF}\left(a, \frac{1}{b}\right)$.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision. The procedure generated by `stats::gammaPDF` reacts to properties of identifiers set via `assume`.

Examples

Example 1

We evaluate the probability density function with $a = 2$ and $b = 1$ at various points:

```
f := stats::gammaPDF(2, 1):
f(-infinity), f(-PI), f(1/2), f(0.5), f(PI), f(infinity)
```

$$0,\ 0,\ \frac{e^{-\frac{1}{2}}}{2},\ 0.3032653299,\ \pi\, e^{-\pi},\ 0$$

```
delete f:
```

Example 2

If `x` is a symbolic object without properties, then it cannot be decided whether $x \geq 0$ holds. A symbolic function call is returned:

```
f := stats::gammaPDF(a, b): f(x)
```

$$\text{stats::gammaPDF}(a, b)(x)$$

With suitable properties, it can be decided whether $x \geq 0$ holds. An explicit expression is returned:

```
assume(0 < x): f(x)
```

$$\frac{x^{a-1}\, e^{-\frac{x}{b}}}{b^a\, \Gamma(a)}$$

```
unassume(x): delete f:
```

Example 3

We use symbolic arguments:

```
f := stats::gammaPDF(a, b): f(x), f(3)
```

$$\text{stats::gammaPDF}(a, b)(x),\ \frac{3^{a-1}\, e^{-\frac{3}{b}}}{b^a\, \Gamma(a)}$$

When numerical values are assigned to `a` and `b`, the function `f` starts to produce numerical results:

```
a := 2: b := 4: f(3), f(3.0)
```

$$\frac{3\, e^{-\frac{3}{4}}}{16},\ 0.08856872864$$

```
delete a, b, f:
```

Example 4

We consider a gamma distribution with large shape parameter:

```
f := stats::gammaPDF(2000, 2):
```

For floating-point approximations, one should not compute an exact result and convert it via `float`. For large shape parameter, it is faster to pass a floating-point argument to `f`. The following call takes some time, because an exact computation of the huge integer $\Gamma(2000) = 1999!$ is involved:

```
float(f(4050))
```

```
0.00377271215
```

The following call is much faster:

```
f(float(4050))
```

```
0.00377271215
```

```
delete f:
```

Parameters

a

The shape parameter: an arithmetical expression representing a positive real value

b

The scale parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`gamma` | `igamma` | `stats::erlangCDF` | `stats::erlangPDF` | `stats::erlangQuantile` | `stats::erlangRandom` | `stats::gammaCDF` | `stats::gammaQuantile` | `stats::gammaRandom`

stats::gammaQuantile

Quantile function of the gamma distribution

Syntax

```
stats::gammaQuantile(a, b)
```

Description

`stats::gammaQuantile(a, b)` returns a procedure representing the quantile function (inverse) of the cumulative distribution function `stats::gammaCDF(a, b)`. For $0 \le x \le 1$, the solution of *stats*::*gammaCDF*(*a*, *b*)(*y*) = *x* is given by *y* = *stats*::*gammaQuantile*(*a*, *b*)(*x*).

The procedure `f:=stats::gammaQuantile(a, b)` can be called in the form `f(x)` with arithmetical expressions `x`. The return value of `f(x)` is either a floating-point number, *infinity*, or a symbolic expression:

If `x` is a real number between 0 and 1 and `a` and `b` can be converted to positive floating-point numbers, then `f(x)` returns a positive floating-point number approximating the solution *y* of *stats*::*gammaCDF*(*a*, *b*)(*y*) = *x*.

The calls `f(0)` and `f(0.0)` produce 0.0 for all values of `a` and `b`.

The calls `f(1)` and `f(1.0)` produce *infinity* for all values of `m`.

In all other cases, `f(x)` returns the symbolic call `stats::gammaQuantile(a, b)(x)`.

Numerical values of *x* are only accepted if $0 \le x \le 1$.

Numerical values of `a` and `b` are only accepted if they are real and positive.

Note that $\text{stats::gammaQuantile}(a, b) = \text{stats::erlangQuantile}\left(a, \frac{1}{b}\right)$.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the quantile function with $a = \pi$ and $b = 11$ at various points:

```
f := stats::gammaQuantile(PI, 11):
f(0), f(1/10), f(0.5), f(1 - 10^(-10)), f(1)
```

$$0.0,\ 13.08489993,\ 30.96813726,\ 324.7230043,\ \infty$$

The value `f(x)` satisfies $stats::gammaCDF(\pi, 11)(f(x)) = x$:

```
stats::gammaCDF(PI, 11)(f(0.987654))
```

$$0.987654$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::gammaQuantile(a, b): f(x), f(9/10)
```

$$\text{stats::gammaQuantile}(a, b)(x),\ \text{stats::gammaQuantile}(a, b)\left(\frac{9}{10}\right)$$

When positive real values are assigned to `a` and `b`, the function `f` starts to produce floating-point values:

```
a := 17: b := 6: f(0.999), f(1 - sqrt(2)/10^5)
```

```
195.7416524, 240.0294477
```

Numerical values for x are only accepted if $0 \le x \le 1$:

```
f(0.5)
```

```
100.0071221
```

```
f(2)
```

```
Error: An argument x with 0 <= x <= 1 is expected. [f]
```

```
delete f, a, b:
```

Parameters

a

The shape parameter: an arithmetical expression representing a positive real value

b

The scale parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`stats::erlangCDF` | `stats::erlangPDF` | `stats::erlangQuantile` | `stats::erlangRandom` | `stats::gammaCDF` | `stats::gammaPDF` | `stats::gammaRandom`

stats::gammaRandom

Generate a random number generator for gamma deviates

Syntax

```
stats::gammaRandom(a, b, <Seed = n>)
```

Description

`stats::gammaRandom(a, b)` returns a procedure that produces gamma deviates (random numbers) with shape parameter $a > 0$ and scale parameter $b > 0$.

The procedure `f:=stats::gammaRandom(a, b)` can be called in the form `f()`. The return value of `f()` is either a floating-point number or a symbolic expression:

If `a` and `b` can be converted to positive floating-point numbers, then `f()` returns a nonnegative floating-point number.

In all other cases, `stats::gammaRandom(a, b)()` is returned symbolically.

Numerical values of `a` and `b` are only accepted if they are real and positive.

The values `X = f()` are distributed randomly according to the cumulative distribution function of the gamma distribution with parameters a and b. For any $0 \le x$, the probability that $X \le x$ is given by

$$\frac{1}{b^a\,\Gamma(a)} \int_0^x t^{a-1}\, e^{-b\,t}\, dt$$

.

Without the option `Seed = n`, an initial seed is chosen internally. This initial seed is set to a default value when MuPAD is started. Thus, each time MuPAD is started or re-initialized with the `reset` function, random generators produce the same sequences of numbers.

Note: In contrast to the function `random`, the generators produced by `stats::gammaRandom` do not react to the environment variable `SEED`.

For efficiency, it is recommended to produce sequences of K random numbers via

```
f := stats::gammaRandom(a, b): f() $k = 1..K;
```

rather than by

```
stats::gammaRandom(a, b)() $k = 1..K;
```

The latter call produces a sequence of generators each of which is called once. Also note that

```
stats::gammaRandom(a, b, Seed = n)() $k = 1..K;
```

does not produce a random sequence, because a sequence of freshly initialized generators would be created each of them producing the same number.

Note that $\text{stats::gammaRandom}(a, b) = \text{stats::erlangRandom}\left(a, \frac{1}{b}\right)$.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We generate gamma deviates with parameters $a = 2$ and $b = \frac{4}{3}$:

```
f := stats::gammaRandom(2, 4/3): f() $ k = 1..4
```

```
     3.958784095, 3.891811185, 6.046842446, 3.142485711
```

```
delete f:
```

Example 2

With symbolic parameters, no random floating-point numbers can be produced:

```
f := stats::gammaRandom(a, b): f()
```

$$\text{stats::gammaRandom}(a, b)()$$

When a and b evaluate to positive real numbers, the result is evaluated to a real floating point number:

```
a := PI: b := 8: f() $ k = 1..4
```

$$19.74371462,\ 12.37357049,\ 13.40137346,\ 29.97534861$$

```
delete f, a, b:
```

Example 3

We use the option `Seed = n` to reproduce a sequence of random numbers:

```
f := stats::gammaRandom(PI, 1/3, Seed = 10^3): f() $ k = 1..4
```

$$0.3631090007,\ 0.8803177461,\ 0.9712460319,\ 1.740056499$$

```
g := stats::gammaRandom(PI, 1/3, Seed = 10^3): g() $ k = 1..4
```

$$0.3631090007,\ 0.8803177461,\ 0.9712460319,\ 1.740056499$$

```
f() = g(), f() = g()
```

$$1.561212345 = 1.561212345,\ 0.4650866732 = 0.4650866732$$

```
delete f:
```

Parameters

a

The shape parameter: an arithmetical expression representing a positive real value

b

The scale parameter: an arithmetical expression representing a positive real value

Options

Seed

Option, specified as `Seed = n`

Initializes the random generator with the integer seed `n`. n can also be the option `CurrentTime`, to make the seed depend on the current time.

This option serves for generating generators that return predictable sequences of pseudo-random numbers. The generator is initialized with the seed `n` which may be an arbitrary integer. Several generators with the same initial seed produce the same sequence of numbers.

When this option is used, the parameters `a` and `b` must be convertible to positive floating-point numbers at the time when the random generator is generated.

Return Values

procedure.

Algorithms

The implemented algorithm for the computation of gamma deviates uses a rejection method applied to uniform random numbers. For more information see: D. Knuth, Seminumerical Algorithms (1998), Vol. 2, pp. 133.

See Also

MuPAD Functions

`stats::erlangCDF` | `stats::erlangPDF` | `stats::erlangQuantile` | `stats::erlangRandom` | `stats::gammaCDF` | `stats::gammaPDF` | `stats::gammaQuantile`

stats::geometricMean

Geometric mean of a data sample

Syntax

```
stats::geometricMean(x1, x2, …)
```

```
stats::geometricMean([x1, x2, …])
```

```
stats::geometricMean(s, <c>)
```

Description

`stats::geometricMean(x1, x2, …, xn)` returns the geometric mean $(x_1\, x_2\, \ldots\, x_n)^{1/n}$ of the data x_i.

The column index `c` is optional, if the data are given by a `stats::sample` object containing only one non-string column. Cf. “Example 3” on page 30-227.

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

Examples

Example 1

We calculate the geometric mean of three values:

```
stats::geometricMean(a, b, c)
```

$$(a\, b\, c)^{1/3}$$

Alternatively, the data may be passed as a list:

```
stats::geometricMean([2, 3, 5])
```

$$30^{1/3}$$

Example 2

We create a sample:

```
stats::sample([[a1, b1, c1], [a2, b2, c2]])
```

```
a1  b1  c1
a2  b2  c2
```

The geometric mean of the second column is:

```
stats::geometricMean(%, 2)
```

$$\sqrt{b1\ b2}$$

Example 3

We create a sample consisting of one string column and one non-string column:

```
stats::sample([["1996", 1242], ["1997", 1353], ["1998", 1142]])
```

```
"1996"  1242
"1997"  1353
"1998"  1142
```

We compute the geometric mean of the second column. In this case this column does not have to be specified, since it is the only non-string column in the sample:

```
float(stats::geometricMean(%))
```

$$1242.68722$$

Parameters

x_1, x_2, …

The statistical data: arithmetical expressions.

s

A sample of domain type `stats::sample`.

c

An integer representing a column index of the sample `s`. This column provides the data `x1`, `x2` etc.

Return Values

arithmetical expression.

See Also

MuPAD Functions

`stats::harmonicMean` | `stats::mean` | `stats::median` | `stats::modal` | `stats::quadraticMean` | `stats::stdev` | `stats::variance`

stats::geometricCDF

The (discrete) cumulative distribution function of the geometric distribution

Syntax

```
stats::geometricCDF(p)
```

Description

`stats::geometricCDF(p)` returns a procedure representing the (discrete) cumulative distribution function

$$x \to \begin{cases} 0 & \text{if } x < 1 \\ 1-(1-p)^{\lfloor x \rfloor} & \text{if } x \geq 1 \end{cases}$$

of the geometric distribution with 'probability parameter' `p`.

The procedure `f:=stats::geometricCDF(p)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a number or a symbolic expression:

If $x < 1$ can be decided, then `f(x)` returns 0. If $x \geq 1$ can be decided, then `f(x)` returns the value $1-(1-p)^{\lfloor x \rfloor}$.

If x is a floating-point number and p can be converted to a floating-point number, then these values are returned as floating-point numbers. Otherwise, symbolic expressions are returned.

The function `f` reacts to properties of identifiers set via `assume`. If x is a symbolic expression with the property $x < 1$ or $x \geq 1$, the corresponding values are returned.

`f(x)` returns the symbolic call `stats::geometricCDF(p)(x)` if neither $x < 1$ nor $x \geq 1$ can be decided.

If $p = 0$ or $p = 0.0$, then $f(x)$ returns 0 or 0.0, respectively, for any value of x.

Numerical values for p are only accepted if they satisfy $0 \le p \le 1$.

If x is a real floating-point number, $f(x)$ produces a floating number provided p is a numerical value. If x is an exact numerical value, no internal floating-point conversion of the parameter p is attempted.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the distribution function with $p = \frac{1}{3}$ at various points:

```
f := stats::geometricCDF(1/3):
f(-PI) = f(float(-PI)), f(1) = f(1.0), f(103/10) = f(10.3)
```

$$0 = 0.0,\ \frac{1}{3} = 0.3333333333,\ \frac{58025}{59049} = 0.9826584701$$

```
delete f:
```

Example 2

If x is a symbolic object without properties, then it cannot be decided whether $x \ge 1$ holds. A symbolic function call is returned:

```
f := stats::geometricCDF(p): f(x)
```

$$\text{stats::geometricCDF}(p)(x)$$

With suitable properties, it can be decided whether $x \ge 1$ holds. An explicit expression is returned:

```
assume(1 <= x): f(x)
```

$$1-(1-p)^{\lfloor x \rfloor}$$

```
unassume(x): delete f:
```

Example 3

We use symbolic arguments:

```
f := stats::geometricCDF(p): f(x)
```

$$\text{stats::geometricCDF}(p)(x)$$

If x is a numerical value, symbolic expressions in p are returned:

```
f(-1), f(1), f(5/2), f(PI)
```

$$0,\ p,\ 1-(p-1)^2,\ (p-1)^3+1$$

When numerical values are assigned to p, the function f starts to produce numbers if the argument is numerical:

```
p := 1/3: f(-1), f(1), f(5/2), f(PI)
```

$$0,\ \frac{1}{3},\ \frac{5}{9},\ \frac{19}{27}$$

```
delete f, p:
```

Parameters

p

The 'probability parameter': an arithmetical expression representing a real number $0 \le p \le 1$.

Return Values

procedure.

Algorithms

The geometric distribution describes the number of Bernoulli trials with success probability p up to and including the first success.

See Also

MuPAD Functions

`stats::geometricPF` | `stats::geometricQuantile` | `stats::geometricRandom`

stats::geometricPF

Probability function of the geometric distribution

Syntax

```
stats::geometricPF(p)
```

Description

`stats::geometricPF(p)` returns a procedure representing the probability function

$$x \to p\,(1-p)^{x-1}$$

for $x = 1, 2, 3, \ldots$ of the geometric distribution with 'probability parameter' `p`.

The procedure `f:=stats::geometricPF(p)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a number or a symbolic expression:

If `x` is a non-integer numerical value, $f(x)$ returns 0 or 0.0, respectively.

If `x` is a positive integer or the floating point equivalent of such an integer, then an explicit value is returned.

In all other cases, `f(x)` returns the symbolic call `stats::geometricPF(n,p)(x)`.

Numerical values for `p` are only accepted if they satisfy $0 \le p \le 1$.

If `x` is a floating-point number, the result is a floating-point number provided p can be converted to a floating-point number between 0.0 and 1.0. If `x` is an exact numerical value, no internal floating point conversion of the parameter p is attempted.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We calculate the geometric probability with $p = \frac{1}{8}$ at various points:

```
f := stats::geometricPF(1/8):
f(-1), f(0.5),  f(1), f(3/2), f(3) = f(float(3))
```

$$0,\ 0.0,\ \frac{1}{8},\ 0,\ \frac{49}{512} = 0.095703125$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::geometricPF(p): f(x)
```

$$\text{stats::geometricPF}(p)(x)$$

If x is a numerical value, symbolic expressions in p are returned:

```
f(17/2), f(8), f(9.0), f(9.2)
```

$$0,\ -p\,(p-1)^7,\ p\,(1.0-1.0\,p)^{8.0},\ 0.0$$

When numerical values are assigned to p, the function f starts to produce numbers if the argument is numerical:

```
p := 1/3: f(17/2), f(8), f(9.0), f(9.2)
```

$$0,\ \frac{128}{6561},\ 0.01300614744,\ 0.0$$

```
delete f, p:
```

Parameters

p

The `probability parameter': an arithmetical expression representing a real number $0 \leq p \leq 1$.

Return Values

procedure.

See Also

MuPAD Functions

`stats::geometricCDF` | `stats::geometricQuantile` | `stats::geometricRandom`

stats::geometricQuantile

Quantile function of the geometric distribution

Syntax

```
stats::geometricQuantile(p)
```

Description

`stats::geometricQuantile(p)` returns a procedure representing the quantile function (discrete inverse) of the cumulative distribution function `stats::geometricCDF(p)`. For $0 \le x \le 1$, $k = stats::geometricQuantile(p)(x)$ is the smallest positive integer satisfying

$$\text{stats::geometricCDF}(p)(k) = 1 - (1-p)^k \ge x$$

.

The procedure `f:=stats::geometricQuantile(p)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of the call `f(x)` is either a positive integer, *infinity*, or a symbolic expression:

If p is a real number satisfying $0 < p \le 1$ and x is a real number satisfying $0 \le x < 1$, then $f(x)$ returns a positive integer.

If $p = 0$, then $f(x)$ returns *infinity* for any x.

If $p = 1$, then $f(x)$ returns 1 for any x.

If $p \ne 0$, then $f(0)$ and $f(0.0)$ return 1.

If $p \ne 1$, then $f(1)$ and $f(1.0)$ return *infinity*.

In all other cases, `f(x)` returns the symbolic call `stats::geometricQuantile(p)(x)`.

Numerical values for `p` are only accepted if they satisfy $0 \le p \le 1$.

If floating-point arguments are passed to the quantile function f, the result is computed with floating-point arithmetic. This is faster than using exact arithmetic, but the result

is subject to internal round-off errors. In particular, round-off may be significant for arguments x close to 1. Cf. “Example 3” on page 30-238.

Finite quantile values k = *stats*::*geometricQuantile*(p)(x) satisfy

$$\text{stats::geometricCDF}(p)(k-1) < x \le \text{stats::geometricCDF}(p)(k)$$

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the quantile function with $p = \frac{1}{\pi}$ at various points:

```
f := stats::geometricQuantile(1/PI):
f(0), f(1/20), f(PI/6), f(0.7), f(1-1/10^10), f(1)
```

$$1,\ 1,\ 2,\ 4,\ 61,\ \infty$$

The value `f(x)` satisfies

$$\text{stats::geometricCDF}(p)(f(x)-1) < x \le \text{stats::geometricCDF}(p)(f(x))$$

```
x := 0.98: k := f(x)
```

$$11$$

```
float(stats::geometricCDF(1/PI)(k - 1)), x,
float(stats::geometricCDF(1/PI)(k))
```

$$0.9783294488,\ 0.98,\ 0.9852273995$$

```
delete f, x, k:
```

Example 2

We use symbolic arguments:

```
f := stats::geometricQuantile(p): f(x), f(9/10)
```

$$\text{stats::geometricQuantile}(p)(x),\ \text{stats::geometricQuantile}(p)\left(\frac{9}{10}\right)$$

When `p` evaluates to a suitable real number, the function `f` starts to produce quantile values:

```
p := 1/sqrt(2):
f(1/2),  f(999/1000), f(1 - 1/10^10), f(1 - 1/10^80)
```

$$1,\ 6,\ 19,\ 151$$

```
delete f, p:
```

Example 3

If floating-point arguments are passed to the quantile function, the result is computed with floating-point arithmetic. This is faster than using exact arithmetic, but the result is subject to internal round-off errors:

```
f := stats::geometricQuantile(1/123):
f(1 - 1/10^19) <> f(float(1 - 1/10^19))
```

$$5360 \neq \infty$$

```
delete f:
```

Parameters

p

The “probability parameter”: an arithmetical expression representing a real number $0 \leq p \leq 1$.

Return Values

procedure.

See Also

MuPAD Functions

`stats::geometricCDF` | `stats::geometricPF` | `stats::geometricRandom`

stats::geometricRandom

Generate a random number generator for geometric deviates

Syntax

```
stats::geometricRandom(p, <Seed = s>)
```

Description

`stats::geometricRandom(p)` returns a procedure that produces geometric deviates (random numbers) with `probability parameter' `p`.

The procedure `f:=stats::geometricRandom(p)` can be called in the form `f()`. The return value of `f()` is a positive integer if `p` is a real number satisfying $0 \le p \le 1$.

Otherwise, `stats::geometricRandom(p)()` is returned symbolically.

Numerical values for `p` are only accepted if they satisfy $0 \le p \le 1$.

The values $X = f()$ are distributed randomly according to the discrete distribution function of the geometric distribution with parameter p, i.e., for $1 \le x$, the probability of $X \le x$ is given by $1 - (1 - p)^{\lfloor x \rfloor}$.

Without the option `Seed = s`, an initial seed is chosen internally. This initial seed is set to a default value when MuPAD is started. Thus, each time MuPAD is started or re-initialized with the `reset` function, random generators produce the same sequences of numbers.

Note: With this option, the parameter `p` must evaluate to a numerical value at the time, when the generator is created.

Note: In contrast to the function `random`, the generators produced by `stats::geometricRandom` do not react to the environment variable `SEED`.

For efficiency, it is recommended to produce sequences of K random numbers via

```
f := stats::geometricRandom(p): f() $k = 1..K;
```

rather than by

```
stats::geometricRandom(p)() $k = 1..K;
```

The latter call produces a sequence of generators each of which is called once. Also note that

```
stats::geometricRandom(p, Seed = s)() $k = 1..K;
```

does not produce a random sequence, because a sequence of freshly initialized generators would be created each of them producing the same number.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We generate geometric deviates with $p = \frac{1}{3}$:

```
f := stats::geometricRandom(1/3): f() $ k = 1..10
```

```
4, 1, 5, 1, 4, 5, 2, 1, 3, 1
```

```
delete f:
```

Example 2

With symbolic parameters, no random floating-point numbers can be produced:

```
f := stats::geometricRandom(p): f()
```

```
stats::geometricRandom(p)()
```

When p evaluates to a real number between 0 and 1, the generator starts to produce random numbers:

```
p := 1/sqrt(70): f(), f(), f()
```

```
3, 6, 2
```

```
delete f, p:
```

Example 3

We use the option `Seed = s` to reproduce a sequence of random numbers:

```
f := stats::geometricRandom(1/10, Seed = 1): f() $ k = 1..10
```

```
6, 3, 21, 17, 2, 55, 7, 16, 26, 11
```

```
g := stats::geometricRandom(1/10, Seed = 1): g() $ k = 1..10
```

```
6, 3, 21, 17, 2, 55, 7, 16, 26, 11
```

```
f() = g(), f() = g()
```

```
9 = 9, 8 = 8
```

```
delete f, g:
```

Parameters

p

The “probability parameter”: an arithmetical expression representing a real number $0 \le p \le 1$.

Options

Seed

Option, specified as `Seed = s`

Initializes the random generator with the integer seed `s`. s can also be the option `CurrentTime`, to make the seed depend on the current time.

This option serves for generating generators that return predictable sequences of pseudo-random numbers. The generator is initialized with the seed `s` which may be an arbitrary integer. Several generators with the same initial seed produce the same sequence of numbers.

When this option is used, the parameter `p` must be convertible to a floating-point number between 0.0 and 1.0 at the time when the random generator is generated.

Return Values

procedure.

See Also

MuPAD Functions

`stats::geometricCDF` | `stats::geometricPF` | `stats::geometricQuantile`

stats::harmonicMean

Harmonic mean of a data sample

Syntax

```
stats::harmonicMean(x1, x2, …)
```

```
stats::harmonicMean([x1, x2, …])
```

```
stats::harmonicMean(s, <c>)
```

Description

`stats::harmonicMean(x1, x2, …, xn)` returns the harmonic mean $\frac{1}{\left(\frac{1}{n}\left(\sum_i \frac{1}{x_i}\right)\right)}$ of the data x_i.

The column index `c` is optional, if the data are given by a `stats::sample` object containing only one non-string column. Cf. “Example 3” on page 30-245.

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

Examples

Example 1

We calculate the harmonic mean of three values:

```
stats::harmonicMean(a, b, c)
```

$$\frac{3}{\frac{1}{a}+\frac{1}{b}+\frac{1}{c}}$$

Alternatively, data may be passed as a list:

```
stats::harmonicMean([2, 3, 5])
```

$$\frac{90}{31}$$

Example 2

We create a sample:

```
stats::sample([[a1, b1, c1], [a2, b2, c2]])
```

```
a1  b1  c1
a2  b2  c2
```

The harmonic mean of the second column is:

```
stats::harmonicMean(%, 2)
```

$$\frac{2}{\frac{1}{b1}+\frac{1}{b2}}$$

Example 3

We create a sample consisting of one string column and one non-string column:

```
stats::sample([["1996", 1242], ["1997", 1353], ["1998", 1142]])
```

```
"1996"  1242
"1997"  1353
"1998"  1142
```

We compute the harmonic mean of the second column. In this case this column does not have to be specified, since it is the only non-string column:

```
float(stats::harmonicMean(%))
```

$$1239.71654$$

Parameters

x_1, x_2, …

The statistical data: arithmetical expressions.

s

A sample of domain type `stats::sample`.

c

An integer representing a column index of the sample *s*. This column provides the data x_1, x_2 etc.

Return Values

arithmetical expression. `FAIL` is returned, if one of the data values is zero (the harmonic mean does not exist).

See Also

MuPAD Functions

`stats::geometricMean` | `stats::mean` | `stats::median` | `stats::modal` | `stats::quadraticMean` | `stats::stdev` | `stats::variance`

stats::hodrickPrescottFilter

The Hodrick-Prescott filter

Syntax

```
stats::hodrickPrescottFilter([x1, x2, …], p)
```

```
stats::hodrickPrescottFilter(s, <c>, p)
```

Description

`stats::hodrickPrescottFilter([x1, x2, …], p)` returns a list of data from which cyclic variations of the time series given by the input data x_1, x_2 etc. are eliminated using the Hodrick-Prescott filter process.

The Hodrick-Prescott filter scheme tries to split a time series consisting of the data x_1, x_2 etc. into a "trend" that is approximately linear in time plus a "cyclic" contribution. The data returned by `stats::hodrickPrescottFilter` describe the trend. The cyclic part c may be computed by

```
x := [x1, x2, ...]:

y := stats::HodrickPrescottFilter(x, p):

c := zip(x, y, _subtract):
```

Thus, $x_i = y_i + c_i$.

Large values of the penalty parameter p lead to smooth straight trend curves. Cf. "Example 5" on page 30-252.

If the data are provided by a `stats::sample` object containing only one non-string column, the column index `c` is optional. Cf. "Example 3" on page 30-249.

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

Examples

Example 1

We apply the Hodrick-Prescott filter to some data. The result shows an obvious trend towards increasing data values:

```
stats::hodrickPrescottFilter([1, 2, 3, 2, 3, 4, 3, 4, 5], 10)
```

```
[1.356447588, 1.819582682, 2.247073017, 2.621315566, 3.0, 3.378684434, 3.752926983,
  4.180417318, 4.643552412]
```

Example 2

We create a sample:

```
s := stats::sample([[i + frandom() - 0.5, -i + frandom() - 0.5]
                    $ i = 1..10])
```

```
0.7703581656   -0.6689628213
 1.653156516    -1.505187219
 2.766272902    -3.319835772
 3.952083055    -3.821218044
 4.854984926    -4.818141187
 6.221918655    -6.026170226
 7.288981492    -7.288474164
 8.355687175    -8.455102606
 9.379160127    -8.580615152
 10.23505742    -9.712454973
```

The Hodrick-Prescott filter process applied to the data in the first column yields:

```
p := 10:
stats::hodrickPrescottFilter(s, 1, p)
```

```
[0.7163345093, 1.721715792, 2.781120731, 3.880013706, 4.989011271, 6.150799324,
  7.274037423, 8.338504453, 9.338923372, 10.28719986]
```

```
stats::hodrickPrescottFilter(s, 2, p)
```

$$[-0.6351055566, -1.760720308, -2.903263691, -3.951897795, -5.004070747, -6.091890803, -7.154501435, -8.098185826, -8.896213527, -9.700312476]$$

```
delete s, p:
```

Example 3

We create a sample consisting of one string column and one non-string column:

```
s := stats::sample([["1996", 1242],
                    ["1997", 1353],
                    ["1998", 1142],
                    ["1999", 1255],
                    ["2000", 1417],
                    ["2001", 1312],
                    ["2002", 1440],
                    ["2003", 1422],
                    ["2004", 1470]
                   ])
```

```
"1996"  1242
"1997"  1353
"1998"  1142
"1999"  1255
"2000"  1417
"2001"  1312
"2002"  1440
"2003"  1422
"2004"  1470
```

We apply the Hodrick-Prescott filter to the second column. In this case, this column needs not be specified, since it is the only non-string column:

```
y := stats::hodrickPrescottFilter(s, 10)
```

$$[1239.848378, 1255.015604, 1270.397993, 1296.009146, 1329.022865, 1362.512038, 1398.347268, 1433.347951, 1468.498758]$$

We convert this list to a sample object:

```
y := stats::sample(y)
```

```
1239.848378
1255.015604
1270.397993
1296.009146
1329.022865
1362.512038
1398.347268
1433.347951
1468.498758
```

We create a new sample consisting of the filtered data:

```
stats::concatCol(stats::col(s, 1), y)
```

```
"1996"  1239.848378
"1997"  1255.015604
"1998"  1270.397993
"1999"  1296.009146
"2000"  1329.022865
"2001"  1362.512038
"2002"  1398.347268
"2003"  1433.347951
"2004"  1468.498758
```

```
delete s, y:
```

Example 4

We model monthly data with a decaying trend of $\frac{1}{(1+0.01\,i)}$, where i is the index of the month. These trend data are obscured by cyclic contributions and random noise:

```
monthlyData:= i ->
    (   1/(1 + 0.01*i)              // the trend
      + 0.7*cos(i * 1.12*2*float(PI))  // cycle
      + 0.3*sin(i * 2.04*4*float(PI))  // cycle
      + 0.2*cos(i * 1.01*6*float(PI))  // cycle
      + 2.3*frandom()            // random noise
```

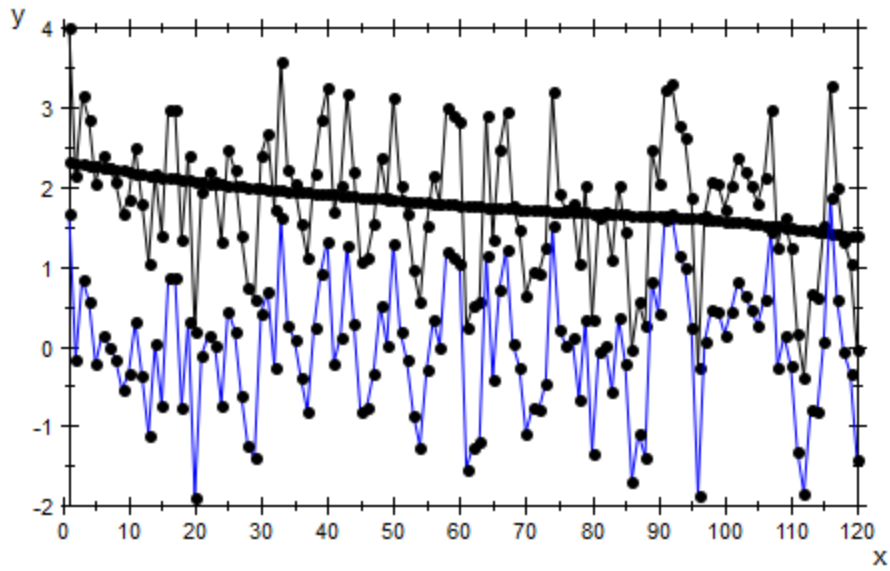
```
    ):
```

We provide monthly data for 10 years, i.e., 120 months. The cyclic contributions and the noise are eliminated from the time series by the Hodrick-Prescott filter process:

```
n := 120:
x := [monthlyData(i) $ i = 1..n]:
trend := stats::hodrickPrescottFilter(x, 10^5):
cycle := zip(x, trend, _subtract):
```

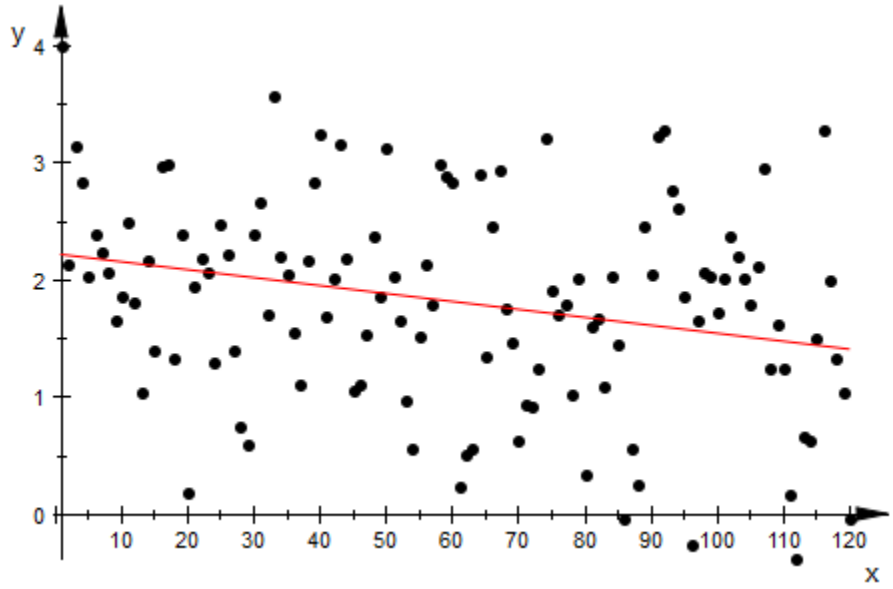
We visualize the splitting of the time series (black) into the approximately linear trend contribution (red) plus the cyclic part (blue):

```
plot(
  plot::Listplot([[i, x[i]] $ i = 1..n], Color = RGB::Black),
  plot::Listplot([[i, trend[i]] $ i = 1..n], Color = RGB::Red),
  plot::Listplot([[i, cycle[i]] $ i = 1..n], Color = RGB::Blue)
)
```



We use a scatterplot to visualize a linear regression of the unfiltered data. The regression line is in good accordance with the trend line above:

```
plot(plot::Scatterplot([[i, x[i]] $ i = 1..n]))
```

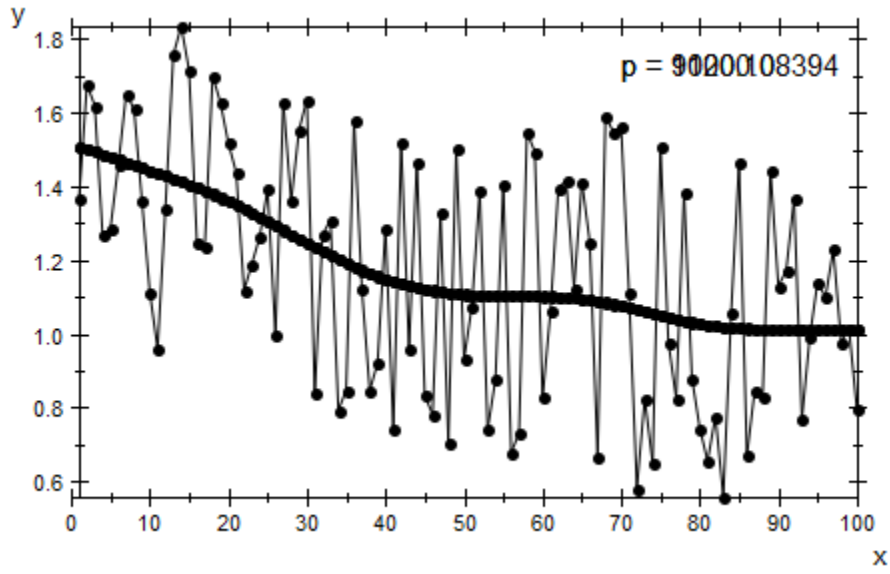



```
delete monthlyData, n, x, trend, cycle:
```

Example 5

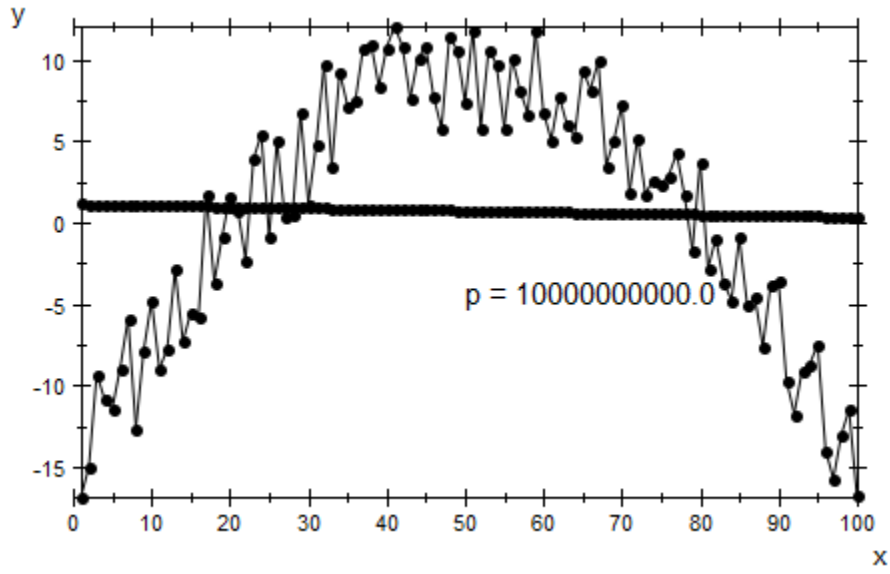
We demonstrate the effect of the penalty parameter p by an animated plot:

```
delete p:
n := 100:
data := [1/(1 + 0.01*i) + frandom() $ i = 1..n]:
for i from 0 to 30 step 1/5 do
  trend := stats::hodrickPrescottFilter(data, 10^(0.2*i));
  L[i] := plot::Listplot([[i, trend[i]] $ i = 1..n],
              Color = RGB::Red, VisibleFromTo = i .. i + 0.2);
  T[i] := plot::Text2d(expr2text(p = 10^(0.2*i)), [70, 1.7],
                       VisibleFromTo = i .. i + 0.2);
end_for:
plot(plot::Listplot([[i, data[i]] $ i= 1..n], Color=RGB::Black),
     L[i] $ i = 0..30 step 1/5, T[i] $ i = 0..30 step 1/5)
```



Large penalty parameters *p* result in trend curves that are close to a straight line. This is not always the desired information. The following animation features a time series with a parabolic trend curve obscured by random noise. Too large values of *p* produce a trend curve that just displays the mean of the data:

```
data := [8*frandom() + 5 - (i - 50)^2/100 $ i = 1..n]:
for i from 0 to 50 do
  trend := stats::hodrickPrescottFilter(data, 10^(0.2*i));
  L[i] := plot::Listplot([[i, trend[i]] $ i = 1..n],
              Color = RGB::Red,
                    VisibleFromTo = i/5 .. (i + 1)/5);
  T[i] := plot::Text2d(expr2text(p = 10^(0.2*i)), [50, -5],
                    VisibleFromTo = i/5 .. (i + 1)/5);
end_for:
plot(plot::Listplot([[i, data[i]] $ i= 1..n], Color=RGB::Black),
     L[i] $ i = 0..50, T[i] $ i = 0..50)
```




```
delete n, data, i, trend, L, T:
```

Parameters

x_1, x_2, …

The statistical data (time series): arithmetical expressions.

s

A sample of domain type `stats::sample`.

c

An integer representing a column index of the sample *s*. This column provides the data x_1, x_2 etc.

p

The penalty parameter of the Hodrick-Prescott scheme: a real positive numerical value.

If the data x_1, x_2 etc. represent monthly measurements, the literature recommends values of p between 100 0 and 140 0.

If the data represent quaterly measurements, values of p around 1600 are recommended.

If the data represent yearly measurements, values of p between 6 and 14 are recommended.

Return Values

List of floating-point data.

References

Robert Hodrick and Edwad C. Prescott, “Postwar U.S. Business Cycles: An Empirical Investigation.” Journal of Money, Credit and Banking, 1997.

Maravall, Agustin and Ana del Rio, “Time Aggregation and the Hodrick-Prescott Filter”, Banco de Espana, 2001.

See Also

MuPAD Functions
`stats::linReg` | `stats::reg`

MuPAD Graphical Primitives
`plot::Listplot` | `plot::Scatterplot`

stats::hypergeometricCDF

The (discrete) cumulative probability function of the hypergeometric distribution

Syntax

```
stats::hypergeometricCDF(N, X, n)
```

Description

`stats::hypergeometricCDF(N, X, n)` returns a procedure representing the probability function

$$x \to \begin{cases} 0 & \text{if } x < \max(0,\, n+X-N) \\ \displaystyle\sum_{i=\max(0,\,n+X-N)}^{\lfloor x \rfloor} \frac{\binom{X}{i}\binom{N-X}{n-i}}{\binom{N}{n}} & \text{if } \max(0,\, n+X-N) \le x \le \min(n,\, X) \\ 1 & \text{if } x > \min(n,\, X) \end{cases}$$

of the hypergeometric distribution with “population size” `N`, “success population size” `X` and “sample size” `n`.

The procedure `f:=stats::hypergeometricCDF(N, X, n)` can be called in the form `f(x)` with arithmetical expressions `x`. The return value of `f(x)` is either a floating-point number, an exact numerical value, or a symbolic expression:

If `x` is an integer, a rational or a floating point number, while `N` is a positive integer and both `X` and `n` are nonnegative integers, then an explicit numerical value is returned.

The function `f` reacts to properties of identifiers set via `assume`.

If any of the parameters is symbolic with properties as follows, then 0, 1 or a symbolic result is returned:

If $x < max(0, n + X - N)$, then $f(x) = 0$.

If $x \geq min(n, X)$, then $f(x) = 1$.

If $X = N$, then $f(x) = 0$ for $x < n$ and $f(x) = 1$ for $x \geq n$.

If $n = N$, then $f(x) = 0$ for $x < X$ and $f(x) = 1$ for $x \geq X$.

If $X = N - 1$, then $f(x) = 0$ for $x < n - 1$, $f(x) = \frac{n}{N}$ for $n - 1 \leq x < n$ and $f(x) = 1$ for $x \geq n$.

If $n = N - 1$, then $f(x) = 0$ for $x < X - 1$, $f(x) = \frac{X}{N}$ for $X - 1 \leq x < X$ and $f(x) = 1$ for $x \geq X$.

If $X = 1$, then $f(x) = \frac{N-n}{N}$ for $0 \leq x < 1$ and $f(x) = 1$ for $x \geq 1$.

If $n = 1$, then $f(x) = \frac{N-X}{N}$ for $0 \leq x < 1$ and $f(x) = 1$ for $x \geq 1$.

If $X = 0$ or $n = 0$, then $f(x) = 1$ for $x \geq 0$.

If `x` and all parameters but `N` are numerical and the assumption on N is `assume(N > X)`, then symbolic values are returned.

`f(x)` returns the symbolic call `stats::hypergeometricCDF(N, X, n)(x)` in all other cases.

Numerical values for `N` are only accepted if they are positive integers.

Numerical values for `X` are only accepted if they are nonnegative integers.

Numerical values for `n` are only accepted if they are nonnegative integers.

Note: If `x` is a floating-point number, the result is a floating number provided N, X and n are numerical values. If `x` is an exact value, the result is a rational number.

Note that for large numbers, floating-point results are computed much faster than exact results. If floating-point approximations are desired, pass a floating-point number `x` to `stats::hypergeometricCDF`.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We compute the distribution function with $N = 20$, $X = 4$ and $n = 3$ at various points:

```
f := stats::hypergeometricCDF(20, 4, 3):
f(-1), f(0), f(1/2), f(1), f(2), f(PI), f(5)
```

$$0,\ \frac{28}{57},\ \frac{28}{57},\ \frac{52}{57},\ \frac{284}{285},\ 1,\ 1$$

```
f(-infinity), f(infinity)
```

$$0,\ 1$$

```
f(-0.2), f(0.0), f(0.7), f(1.0), f(float(PI)), f(4.0)
```

$$0.0,\ 0.4912280702,\ 0.4912280702,\ 0.9122807018,\ 1.0,\ 1.0$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::hypergeometricCDF(N, X, n): f(x), f(8), f(8.0)
```

$$\text{stats::hypergeometricCDF}(N, X, n)(x),\ \text{stats::hypergeometricCDF}(N, X, n)(8),\ \text{stats::hypergeometricCDF}(N, X, n)(8.0)$$

When real numbers are assigned to N, X and n, the function f starts to produce explicit results if the argument is numerical:

```
N := 15: X := 6: n := 5:
f(0), f(1), f(2.0), f(3.5), f(4)
```

$$\frac{6}{143},\ \frac{42}{143},\ 0.7132867133,\ 0.953046953,\ \frac{999}{1001}$$

```
delete f, N, X, n:
```

Example 3

If one or more parameters are symbolic, usually a symbolic call is returned. Some combinations of symbolic and numeric values for N, X, n and x, however, may yield symbolic or numeric results:

```
f := stats::hypergeometricCDF(N, X, n):
X := 1:
f(-1), f(0), f(1/2), f(0.5), f(3/2), f(2.0)
```

$$0,\ \frac{N-n}{N},\ \frac{N-n}{N},\ \frac{N-1.0\,n}{N},\ 1,\ 1.0$$

```
X := N - 1:
f(1), f(n-1), f(n)
```

$$\text{stats::hypergeometricCDF}(N,\ N-1,\ n)(1),\ \frac{n}{N},\ 1$$

```
delete X:
```

Example 4

If x and all parameters but N are numerical and N is assumed to be greater than X, a symbolic expression is returned:

```
X := 6:
assume(N > X):
f := stats::hypergeometricCDF(N, X, 5):
f(1), f(2), f(3)
```

$$1-\frac{15\,N+20\binom{N-6}{2}+15\binom{N-6}{3}-84}{\binom{N}{5}},\ 1-\frac{15\,N+20\binom{N-6}{2}-84}{\binom{N}{5}},\ 1-\frac{15\,N-84}{\binom{N}{5}}$$

```
delete f, N, X:
```

Parameters

N

The “population size”: an arithmetical expression representing a positive integer

X

The “success population size”: an arithmetical expression representing a nonnegative integer

n

The “sample size”: an arithmetical expression representing a nonnegative integer

Return Values

procedure.

See Also

MuPAD Functions

`stats::hypergeometricPF` | `stats::hypergeometricQuantile` | `stats::hypergeometricRandom`

stats::hypergeometricPF

Probability function of the hypergeometric distribution

Syntax

```
stats::hypergeometricPF(N, X, n)
```

Description

`stats::hypergeometricPF(N, X, n)` returns a procedure representing the probability function

$$x \to \frac{\binom{X}{x}\binom{N-X}{n-x}}{\binom{N}{n}}$$

for $x \in \mathbb{N} \cap [\max(0, n+X-N), \min(n, X)]$ of the hypergeometric distribution with "population size" `N`, "success population size" `X` and "sample size" `n`.

The procedure `f:=stats::hypergeometricPF(N, X, n)` can be called in the form `f(x)` with arithmetical expressions `x`. The return value of `f(x)` is either a floating-point number, an exact numerical value, or a symbolic expression:

If `x` is a noninteger numerical value, `f(x)` returns 0 or 0.0, respectively.

If `x` is an integer or the floating-point equivalent of an integer, while `N` is a positive integer and both `X` and `n` are nonnegative integers, then an explicit numerical value is returned.

The function `f` reacts to properties of identifiers set via `assume`.

If any of the parameters is symbolic with properties as follows, then 0, 1 or a symbolic result is returned:

If $X = N$, then $f(x) = 1$ for $x = n$ and $f(x) = 0$ for $x \neq n$. If $n = N$, then $f(x) = 1$ for $x = X$ and $f(x) = 0$ for $x \neq X$.

If $X = N - 1$, then $f(x) = \frac{N-n}{N}$ for $x = n$, $f(x) = \frac{n}{N}$ for $x = n - 1$ and $f(x) = 0$ for $x \neq n, n - 1$.

If $n = N - 1$, then $f(x) = \frac{N-X}{N}$ for $x = X$, $f(x) = \frac{X}{N}$ for $x = X - 1$ and $f(x) = 0$ for $x \neq X, X - 1$.

If $X = 1$, then $f(x) = \frac{N-n}{N}$ for $x = 0$, $f(x) = \frac{n}{N}$ for $x = 1$ and $f(x) = 0$ for $x \neq 0, 1$.

If $n = 1$, then $f(x) = \frac{N-X}{N}$ for $x = 0$, $f(x) = \frac{X}{N}$ for $x = 1$ and $f(x) = 0$ for $x \neq 0, 1$.

If $X = 0$ or $n = 0$, then $f(x) = 1$ for $x = 0$ and $f(x) = 0$ for $x \neq 0$.

If `x` and all parameters but `N` are numerical and the assumption on `N` is `assume(N > X)`, then symbolic values are returned.

`f(x)` returns the symbolic call `stats::hypergeometricPF(N, X, n)(x)` in all other cases.

Numerical values for `N` are only accepted if they are positive integers.

Numerical values for `X` are only accepted if they are nonnegative integers.

Numerical values for `n` are only accepted if they are nonnegative integers.

Note: If `x` is a floating-point number, the result is a floating number provided `N`, `X` and `n` are numerical values. If `x` is an exact value, the result is a rational number.

Note that for large numbers, floating-point results are computed much faster than exact results. If floating-point approximations are desired, pass a floating-point number `x` to `stats::hypergeometricPF`.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We compute the probability function with $N = 20$, $X = 4$ and $n = 3$ at various points:

```
f := stats::hypergeometricPF(20, 4, 3):
f(-infinity), f(0), f(1/2), f(1), f(2), f(4), f(infinity)
```

$$0,\ \frac{28}{57},\ 0,\ \frac{8}{19},\ \frac{8}{95},\ 0,\ 0$$

```
f(-0.2), f(0.0), f(0.7), f(1.0), f(2.0), f(2.7), f(3.0), f(4.0)
```

$$0.0,\ 0.4912280702,\ 0.0,\ 0.4210526316,\ 0.08421052632,\ 0.0,\ 0.00350877193,\ 0.0$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::hypergeometricPF(N, X, n): f(x), f(8), f(8.0)
```

$$\text{stats::hypergeometricPF}(N, X, n)(x),\ \text{stats::hypergeometricPF}(N, X, n)(8),\ \text{stats::hypergeometricPF}(N, X, n)(8.0)$$

When real numbers are assigned to N, X and n, the function f starts to produce explicit results if the argument is numerical:

```
N := 15: X := 6: n := 5:
f(0), f(1), f(2.0), f(3.5), f(4)
```

$$\frac{6}{143},\ \frac{36}{143},\ 0.4195804196,\ 0.0,\ \frac{45}{1001}$$

```
delete f, N, X, n, x:
```

Example 3

If one or more parameters are symbolic, usually a symbolic call is returned. Some combinations of symbolic and numeric values for N, X, n and x, however, may yield symbolic or numeric results:

```
f := stats::hypergeometricPF(N, X, n):
X := 1:
f(-1), f(0), f(1), f(3/2), f(2), f(3)
```

$$0,\ \frac{N-n}{N},\ \frac{n}{N},\ 0,\ 0,\ 0$$

```
X := N:
f(-1), f(n), f(n + 1)
```

$$0,\ 1,\ 0$$

```
delete f, X:
```

Example 4

If x and all parameters but N are numerical and N is assumed to be greater than X, a symbolic expression is returned:

```
X := 6:
assume(N > X):
f := stats::hypergeometricPF(N, X, 5):
f(2), f(4), f(5.0)
```

$$\frac{15\binom{N-6}{3}}{\binom{N}{5}},\ \frac{15\,N-90}{\binom{N}{5}},\ \frac{6.0}{\binom{N}{5.0}}$$

```
delete f, N, X:
```

Parameters

N

The “population size”: an arithmetical expression representing a positive integer

X

The “success population size”: an arithmetical expression representing a nonnegative integer

n

The “sample size”: an arithmetical expression representing a nonnegative integer

Return Values

procedure.

See Also

MuPAD Functions

`stats::hypergeometricCDF` | `stats::hypergeometricQuantile` | `stats::hypergeometricRandom`

stats::hypergeometricQuantile

Quantile function of the hypergeometric distribution

Syntax

```
stats::hypergeometricQuantile(N, X, n)
```

Description

`stats::hypergeometricQuantile(N, X, n)` returns a procedure representing the quantile function (discrete inverse) of the cumulative distribution function `stats::hypergeometricCDF(N, X, n)`. For $0 \le x \le 1$, k = *stats*::*hypergeometricQuantile*(N, X, n)(x) is the smallest nonnegative integer satisfying

$$\text{stats::hypergeometricCDF}(N, X, n)(k) = \sum_{i=0}^{k} \frac{\binom{X}{i}\binom{N-X}{n-i}}{\binom{N}{n}} \ge x$$

.

The procedure `f:=stats::hypergeometricQuantile(N, X, n)` can be called in the form `f(x)` with arithmetical expressions `x`. The return value of `f(x)` is either a natural number between 0 and $min(X, n)$, or a symbolic expression:

If x is a real number satisfying $0 \le x \le 1$, while `N` is a positive integer and both `X` and `n` are nonnegative integers, then an explicit numerical value is returned.

The function `f` reacts to properties of identifiers set via `assume`.

If any of the parameters is symbolic, then in some cases a symbolic result will be returned:

0 will be returned if either any of x, n or X is zero or if $n = 1$ and $x \le \frac{N-X}{N}$ or if $X = 1$ and $x \le \frac{N-n}{N}$.

1 will be returned if $n = 1$ and $x > \frac{N-X}{N}$ or if $X = 1$ and $x > \frac{N-n}{N}$.

n will be returned if $X = N - 1$ and $x > \frac{n}{N}$ or if $X = N$ and $x > 0$.

X will be returned if $n = N - 1$ and $x > \frac{X}{N}$ or if $n = N$ and $x > 0$.

$n - 1$ will be returned if $X = N - 1$ and $x = \frac{n}{N}$ provided that n is symbolic, whereas $X - 1$ will be returned if $n = N - 1$ and $x = \frac{X}{N}$ provided that X is symbolic.

Finally $min(X, n)$ will be returned if $x = 1$.

The symbolic call `stats::hypergeometricQuantile(N, X, n)(x)` is returned by `f(x)` in all other cases.

Numerical values for `N` are only accepted if they are positive integers.

Numerical values for `X` are only accepted if they are nonnegative integers.

Numerical values for `n` are only accepted if they are nonnegative integers.

If `x` is a floating-point number, the result is a floating number provided N, X and n are numerical values. If `x` is an exact value, the result is a rational number.

Note: Note that if floating-point arguments are passed to the quantile function f, the result is computed with floating-point arithmetic. This is faster than using exact arithmetic, but the result is subject to internal round-off errors. In particular, round-off may be significant for arguments x close to 1. Cf. "Example 4" on page 30-269.

The quantile value $k = stats::hypergeometricQuantile(N, X, n)(x)$ satisfies

$$\mathrm{cdf}(k-1) < x \le \mathrm{cdf}(k)$$

where $cdf = stats::hypergeometricCDF(N, X, n)$.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the quantile function with $N = 50$, $X = 30$ and $n = 10$ at some points:

```
f := stats::hypergeometricQuantile(50, 30, 10):
f(0), f((2/3)^30), f(PI/10), f(0.5), f(1 - 1/10^10)
```

$$0,\ 0,\ 5,\ 6.0,\ 10$$

With *cdf* = *stats*::*hypergeometricCDF*(*N*, *X*, *n*), the quantile value *f*(*x*) satisfies the inequalities $cdf(f(x) - 1) < x \le cdf(f(x))$:

```
x := 0.7: f(x)
```

$$7.0$$

```
stats::hypergeometricCDF(50, 30, 10)(float(f(x) - 1)), x,
stats::hypergeometricCDF(50, 30, 10)(float(f(x)))
```

$$0.6350317132,\ 0.7,\ 0.8609613426$$

```
delete f, x:
```

Example 2

We use symbolic arguments:

```
f := stats::hypergeometricQuantile(N, X, n): f(x), f(9/10)
```

$$\text{stats::hypergeometricQuantile}(N, X, n)(x),\ \text{stats::hypergeometricQuantile}(N, X, n)\left(\frac{9}{10}\right)$$

When `N`, `X` and `n` evaluate to suitable numbers, the function `f` starts to produce quantile values:

```
N := 500: X := 80: n := 18:
f(1/2),  f(999/1000), f(1 - 1/10^10), f(1 - 1/10^80)
```

$$3, 8, 15, 18$$

```
delete f, N, X, n:
```

Example 3

If one or more parameters are symbolic, usually a symbolic call is returned. Some combinations of symbolic and numeric values for N, X, n and x, however, may yield symbolic or numeric results:

```
f := stats::hypergeometricQuantile(N, X, n):
f(0), f(1)
```

$$0, \min(X, n)$$

```
X := N - 1:
f(n/N), f(7/10)
```

$$n - 1, \text{stats::hypergeometricQuantile}(N, N - 1, n)\left(\frac{7}{10}\right)$$

```
assume(x > n/N):
f(0.5), f(x)
```

$$\text{stats::hypergeometricQuantile}(N, N - 1, n)(0.5), n$$

```
delete f, X, x:
```

Example 4

If floating-point arguments are passed to the quantile function, the result is computed with floating-point arithmetic. This is faster than using exact arithmetic, but the result is subject to internal round-off errors:

```
f := stats::hypergeometricQuantile(10000, 2000, 30):
f(1 - 1/10^18) <> f(float(1 - 1/10^18))
```

$$28 \neq 27.0$$

```
delete f:
```

Parameters

N

The “population size”: an arithmetical expression representing a positive integer

X

The “success population size”: an arithmetical expression representing a nonnegative integer

n

The “sample size”: an arithmetical expression representing a nonnegative integer

Return Values

procedure.

See Also

MuPAD Functions

`stats::hypergeometricCDF` | `stats::hypergeometricPF` | `stats::hypergeometricRandom`

stats::hypergeometricRandom

Generate a random number generator for hypergeometric deviates

Syntax

```
stats::hypergeometricRandom(N, X, n, <Seed = s>)
```

Description

`stats::hypergeometricRandom(N, X, n)` returns a procedure that produces hypergeometric-deviates (random numbers) with population size `N`, success population size `X` and sample size `n`.

The procedure `f:=stats::hypergeometricRandom(N, X, n)` can be called in the form `f()`.

The return value of `f(x)` is either an integer between $max(0, X + n - N)$ and $min(X, n)$ or a symbolic expression:

If `N` is a positive integer and both `X` and `n` are nonnegative integers, then an explicit numerical value is returned.

If any of the parameters is symbolic, then in some cases numerical or symbolic result will be returned:

0 will be returned if either n or X is zero, n will be returned if $N = X$ and X will be returned if $N = n$.

The symbolic call `stats::hypergeometricRandom(N, X, n)()` is returned in all other cases.

Numerical values for `N` are only accepted if they are positive integers.

Numerical values for `X` and `n` are only accepted if they are integers that satisfy $0 \le X, n \le N$.

The values $R = f()$ are distributed randomly according to the hypergeometric distribution with poupulation size `N`, success population size `X` and sample size `n`.

For any $max(0, X + n - N) \le x \le min(X, n)$, the probability of $R \le x$ is given by

$$\sum_{i=\max(0,\,n+X-N)}^{\lfloor x \rfloor} \frac{\binom{X}{i}\binom{N-X}{n-i}}{\binom{N}{n}}$$

.

Without the option `Seed = s`, an initial seed is chosen internally. This initial seed is set to a default value when MuPAD is started. Thus, each time MuPAD is started or re-initialized with the `reset` function, random generators produce the same sequences of numbers.

Note: With this option, the parameters `N`, `X` and `n` must evaluate to suitable numerical values at the time, when the generator is created.

Note: In contrast to the function `random`, the generators produced by `stats::hypergeometricRandom` do not react to the environment variable `SEED`.

For efficiency, it is recommended to produce sequences of K random numbers via

```
f := stats::hypergeometricRandom(N, X, n): f() $k = 1..K;
```

rather than by

```
stats::hypergeometricRandom(N, X, n)() $k = 1..K;
```

The latter call produces a sequence of generators each of which is called once. Also note that

```
stats::hypergeometricRandom(N, X, n, Seed = s)() $k = 1..K;
```

does not produce a random sequence, because a sequence of freshly initialized generators would be created each of them producing the same number.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We generate hypergeometric deviates with parameters $N = 100$, $X = 30$, and $n = 7$:

```
f := stats::hypergeometricRandom(100, 30, 7): f() $ k = 1..10
```

$$2, 2, 6, 2, 2, 3, 0, 3, 4, 4$$

```
delete f:
```

Example 2

With symbolic parameters, no random numbers can be produced:

```
f := stats::hypergeometricRandom(N, X, n): f()
```

$$\text{stats::hypergeometricRandom}(N, X, n)()$$

When N, X and n evaluate to suitable numbers, the generator starts to produce random numbers:

```
N := 200: X := 80: n := 20: f() $ k= 1..10
```

$$6, 4, 12, 5, 10, 5, 11, 5, 9, 7$$

```
delete f, N, X, n:
```

Example 3

We use the option `Seed = s` to reproduce a sequence of random numbers:

```
f := stats::hypergeometricRandom(500, 100, 50, Seed = 1):
f() $ k = 1..10
```

$$6, 10, 4, 13, 8, 5, 12, 12, 11, 11$$

```
g := stats::hypergeometricRandom(500, 100, 50, Seed = 1):
g() $ k = 1..10
```

```
     6, 10, 4, 13, 8, 5, 12, 12, 11, 11
```

```
f() = g(), f() = g()
```

```
     5 = 5, 10 = 10
```

```
delete f, g:
```

Parameters

N

The “population size”: an arithmetical expression representing a positive integer

X

The “success population size”: an arithmetical expression representing a nonnegative integer

n

The “sample size”: an arithmetical expression representing a nonnegative integer

Options

Seed

Option, specified as `Seed = s`

Initializes the random generator with the integer seed `s`. s can also be the option `CurrentTime`, to make the seed depend on the current time.

This option serves for generating generators that return predictable sequences of pseudo-random numbers. The generator is initialized with the seed `s` which may be an arbitrary integer. Several generators with the same initial seed produce the same sequence of numbers.

When this option is used, the parameters `N`, `X` and `n` must be numerical values at the time when the random generator is generated.

Return Values

procedure.

See Also

MuPAD Functions

`stats::hypergeometricCDF` | `stats::hypergeometricPF` | `stats::hypergeometricQuantile`

stats::ksGOFT

The Kolmogorov-Smirnov goodness-of-fit test

Syntax

```
stats::ksGOFT(x1, x2, …, CDF = f)

stats::ksGOFT([x1, x2, …], CDF = f)

stats::ksGOFT(s, <c>, CDF = f)
```

Description

`stats::ksGOFT`([x_1, x_2, …], *CDF* = *f*) applies the Kolmogorov-Smirnov goodness-of-fit test for the null hypothesis: "x_1, x_2, … is an `f`-distributed sample".

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

An error is raised if any of the data cannot be converted to a real floating-point number.

Let y_1, …, y_n be the input data x_1, …, x_n arranged in ascending order. `stats::ksGOFT` returns the list

$$[\text{PValue1} = \text{p1},\ \text{StatValue1} = \text{K1},\ \text{PValue2} = \text{p2},\ \text{StatValue2} = \text{K2}]$$

containing the following information:

1. `K1` is the Kolmogorov-Smirnov statistic $\sqrt{n}\max\left(\left\{\frac{j}{n} - f\left(y_j\right) \;\middle|\; 1 \le j \le n\right\}\right)$.
2. `p1` is the observed significance level $e^{-2\,\mathrm{K1}^2}\left(1 - \frac{2\,\mathrm{K1}}{3\sqrt{n}} + \frac{2\,\mathrm{K1}^2}{3n} - \frac{4\,\mathrm{K1}^4}{9n}\right)$ of the statistic `K1`.
3. `K2` is the Kolmogorov-Smirnov statistic $\sqrt{n}\max\left(\left\{f\left(y_j\right) - \frac{j-1}{n} \;\middle|\; 1 \le j \le n\right\}\right)$.

4 p2 is the observed significance level $e^{-2\,K2^2}\left(1-\frac{2\,K2}{3\sqrt{n}}+\frac{2\,K2^2}{3\,n}-\frac{4\,K2^4}{9\,n}\right)$ of the statistic K2.

For the Kolmogorov-Smirnov statistic K corresponding to *K1* or *K2*, respectively, the observed significance levels p1, p2 are computed by an asymptotic approximation of the exact probability

$$\Pr(K>k)=\frac{k}{n^{n-\frac{1}{2}}}\left(\sum_{i=1}^{\lfloor\sqrt{n}\,k\rfloor}\binom{n}{i}\left((i-\sqrt{n}\,k)^i\,(\sqrt{n}\,k+n-i)^{n-i-1}\right)\right).$$

For large n, these probabilities are approximated by

$$e^{-2\,k^2}\left(1-\frac{2\,k}{3\sqrt{n}}+\frac{2\,k^2}{3\,n}-\frac{4\,k^4}{9\,n}\right).$$

Thus, the observed significance levels returned by `stats::ksGOFT` approximate the exact probabilities for large n. Roughly speaking, for n = 10, the 3 leading digits of p1, p2 correspond to the exact probabilities. For n = 100, the 4 leading digits of p1, p2 correspond to the exact probabilities. For n = 1000, the 6 leading digits of p1, p2 correspond to the exact probabilities.

The observed significance level `PValue1 = p1` returned by `stats::ksGOFT` has to be interpreted in the following way:

Under the null hypothesis, the probability $p1 = Pr(K > K1)$ should not be small. Specifically, $p1 = Pr(K > K1) \geq \alpha$ should hold for a given significance level $0<\alpha<<1$. If this condition is violated, the hypothesis may be rejected at level α.

Thus, if the observed significance level p1 = $Pr(K > K1)$ satisfies $p1 < \alpha$, the sample leading to the value K1 of the statistic K represents an unlikely event and the null hypotheses may be rejected at level α.

The corresponding interpretation holds for `PValue2 = p2`: if `p2 = Pr(K > K2)` satisfies $p2 < \alpha$, the null hypotheses may be rejected at level α.

Note that *both* observed significance levels p1, p2 must be sufficiently large to make the data pass the test. The null hypothesis may be rejected at level α if any of the two values is smaller than α.

If *p1* and *p2* are both close to 1, this should raise suspicion about the randomness of the data: they indicate a fit that is *too good*.

Distributions that are not provided by the `stats`-package can be implemented easily by the user. A user defined procedure *f* can implement any cumulative distribution function; `stats::ksGOFT` calls *f*(*x*) with real floating-point arguments from the data sample. The function *f* must return a numerical real value between 0 and 1. Cf. "Example 3" on page 30-280.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We create a sample of 1000 normally distributed random numbers:

```
r := stats::normalRandom(0, 1, Seed = 123):
data := [r() $ i = 1 .. 1000]:
```

We test whether these data are indeed normally distributed with mean 0 und variance 1. We pass the corresponding cumulative distribution function `stats::normalCDF(0, 1)` to `stats::ksGOFT`:

```
stats::ksGOFT(data, CDF = stats::normalCDF(0, 1))
```

```
[PValue1 = 0.722852785, StatValue1 = 0.397634617, PValue2 = 0.2073651285,
  StatValue2 = 0.8816932145]
```

The result shows that the data can be accepted as a sample of normally distributed numbers: both observed significance levels 0.722... and 0.207... are not small.

Next, we inject some further data into the sample:

```
data := data . [frandom() $ i = 1..100]:
stats::ksGOFT(data, CDF = stats::normalCDF(0, 1))
```

```
[PValue1 = 0.7520868926, StatValue1 = 0.3724682725, PValue2 = 0.00006540335794,
  StatValue2 = 2.188978212]
```

Now, the data should not be accepted as a sample of normal deviates with mean 0 and variance 1, because the second observed significance level `PValue2 = 0.000065..` is very small.

```
delete r, data:
```

Example 2

We create a sample consisting of one string column and two non-string columns:

```
s := stats::sample(
  [["1996", 1242, PI - 1/2], ["1997", 1353, PI + 0.3],
   ["1998", 1142, PI + 0.5], ["1999", 1201, PI - 1],
   ["2001", 1201, PI]])
```

```
"1996"  1242  PI - 1/2
"1997"  1353  PI + 0.3
"1998"  1142  PI + 0.5
"1999"  1201    PI - 1
"2001"  1201        PI
```

We consider the data in the third column. The mean and the variance of these data are computed:

```
[m, v] := [stats::mean(s, 3), stats::variance(s, 3)]
```

$$[\pi - 0.14,\ 0.373]$$

We check whether the data of the 3rd column are normally distributed with the mean and variance computed above:

```
stats::ksGOFT(s, 3, CDF = stats::normalCDF(m, v))
```

```
[PValue1 = 0.7366062033, StatValue1 = 0.3294603834, PValue2 = 0.6216810571,
  StatValue2 = 0.4263183582]
```

Both observed significance levels $0.736\ldots$ and $0.621\ldots$ returned by the test are not small. There is no reason to reject the null hypothesis that the data are normally distributed.

```
delete s, m, v:
```

Example 3

We demonstrate how user-defined distribution functions can be used. The following function represents the cumulative distribution function $Pr(X \le x) = x^2$ of a variable X supported on the interval [0, 1]. It will be called with floating-point arguments x and must return numerical values between 0 and 1:

```
f := proc(x)
     begin
       if x <= 0 then return(0)
       elif x < 1 then return(x^2)
       else return(1)
       end_if
     end_proc:
```

We test the hypothesis that the following data are f-distributed:

```
data := [sqrt(frandom()) $ k = 1..10^2]:
stats::ksGOFT(data, CDF = f)
```

```
[PValue1 = 0.366526126, StatValue1 = 0.692353403, PValue2 = 0.8844815733,
  StatValue2 = 0.2318684139]
```

At a given significance level of 0.1, say, the hypothesis should not be rejected: both observed significance levels `p1` = `0.366...` and `p2` = `0.884...` exceed 0.1.

```
delete f, data:
```

Parameters

x_1, x_2, …

The statistical data: real numerical values

f

A procedure representing a cumulative distribution function. Typically, one of the distribution functions of the `stats`-package such as `stats::normalCDF`(n, v) etc.

s

A sample of domain type `stats::sample`

c

An integer representing a column index of the sample `s`. This column provides the data `x1`, `x2` etc. There is no need to specify a column number `c` if the sample has only one column.

Return Values

List with four equations `[PValue1 = p1, StatValue1 = K1, PValue2 = p2, StatValue2 = K2]`, with floating-point values `p1`, `K1`, `p2`, `K2`. See the "Details" section below for the interpretation of these values.

References

D. E. Knuth, The Art of Computer Programming, Vol 2: Seminumerical Algorithms, pp. 48. Addison-Wesley (1998).

See Also

MuPAD Functions

`stats::csGOFT` | `stats::sample` | `stats::swGOFT` | `stats::tTest`

stats::kurtosis

Kurtosis (excess) of a data sample

Syntax

```
stats::kurtosis(x₁, x₂, …)
```

```
stats::kurtosis([x₁, x₂, …])
```

```
stats::kurtosis(s, <c>)
```

Description

`stats::kurtosis`(x_1, x_2, …, x_n) returns the kurtosis (the coefficient of excess)

$$\frac{\frac{1}{n}\left(\sum_{i=1}^{n}(x_i-\overline{x})^4\right)}{\left(\frac{1}{n}\left(\sum_{i=1}^{n}(x_i-\overline{x})^2\right)\right)^2}-3,$$

where $\overline{x}$ is the mean of the data x_i.

The kurtosis measures whether a distribution is “flat” or “peaked”. For normally distributed data, the kurtosis is zero. If the distribution function of the data has a flatter top than the normal distribution, then the kurtosis is negative. The kurtosis is positive, if the distribution function has a high peak compared to the normal distribution.

The column index `c` is optional, if the data are given by a `stats::sample` object containing only one non-string column. Cf. “Example 3” on page 30-283.

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

Examples

Example 1

We calculate the kurtosis of some values:

```
stats::kurtosis(0, 7, 7, 6, 6, 6, 5, 5, 4, 1)
```

$$-\frac{74146}{271441}$$

Alternatively, data may be passed as a list:

```
stats::kurtosis([2, 2, 4, 6, 8, 10, 10])
```

$$-\frac{85}{54}$$

Example 2

We create a sample:

```
stats::sample([[a, 5, 8], [b, 3, 7], [c, d, 0]])
```

```
a  5  8
b  3  7
c  d  0
```

The kurtosis of the second column is:

```
stats::kurtosis(%, 2)
```

$$\frac{3\left(\frac{d}{3}-\frac{1}{3}\right)^4+3\left(\frac{d}{3}-\frac{7}{3}\right)^4+3\left(\frac{2\,d}{3}-\frac{8}{3}\right)^4}{\left(\left(\frac{d}{3}-\frac{1}{3}\right)^2+\left(\frac{d}{3}-\frac{7}{3}\right)^2+\left(\frac{2\,d}{3}-\frac{8}{3}\right)^2\right)^2}-3$$

Example 3

We create a sample consisting of one string column and one non-string column:

```
stats::sample([["1996", 1242], ["1997", 1353], ["1998", 1142]])
```

```
"1996"  1242
"1997"  1353
"1998"  1142
```

We compute the kurtosis of the second column. In this case this column does not have to be specified, since it is the only non-string column:

```
stats::kurtosis(%)
```

$$-\frac{3}{2}$$

Parameters

x_1, x_2, …

The statistical data: arithmetical expressions.

s

A sample of domain type `stats::sample`.

c

An integer representing a column index of the sample `s`. This column provides the data `x1`, `x2` etc.

Return Values

Arithmetical expression. `FAIL` is returned, if the kurtosis does not exist.

See Also

MuPAD Functions

`stats::obliquity`

stats::linReg

Linear regression (least squares fit)

Syntax

```
stats::linReg([x₁, x₂, …], [y₁, y₂, …], <[w₁, w₂, …]>, <CovarianceMatrix>)

stats::linReg([[x₁, y₁, <w₁>], [x₂, y₂, <w₂>], …], <CovarianceMatrix>)

stats::linReg(s, <cx, cy, <cw>>, <CovarianceMatrix>)

stats::linReg(s, <[cx, cy, <cw>]>, <CovarianceMatrix>)
```

Description

`stats::linReg([x₁, x₂, …], [y₁, y₂, …], [w₁, w₂, …])` computes the least squares estimators a, b of a linear relation $y_i = a + b\,x_i$ between the data pairs (x_i, y_i) by minimizing

$$\chi^2 = \sum_i w_i \left| y_i - a - b\,x_i \right|^2 .$$

A linear relation $y_i = a + b\,x_i + e_i$ between the data pairs (x_i, y_i) is assumed.

The column indices `cx`, `cy` are optional if the data are given by a `stats::sample` object containing only two non-string columns. Cf. “Example 2” on page 30-287.

Multivariate linear regression and non-linear regression is provided by `stats::reg`.

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

Examples

Example 1

We calculate the least square estimators of four pairs of values given in two lists. Note that there is a linear relation $y = 1 + 2\,x$ between the entries of the lists. The minimized quadratic deviation is 0 indicating a perfect fit:

```
stats::linReg([0, 1, 2, 3], [1, 3, 5, 7])
```

$$[[1, 2], 0]$$

Alternatively, data may be specified by a list of pairs:

```
stats::linReg([[1, 1.0], [2, 1.2], [3, 1.3], [4, 1.5]])
```

$$[[0.85, 0.16], 0.002]$$

We assume that the variable y in the previous example is Poissonian, i.e. that the measurements $(y_i) = (1.0, 1.2, 1.3, 1.5)$ have errors given by the standard deviation $\sigma(y_i) = \sqrt{y_i}$. We provide corresponding weights $w_i = \frac{1}{\sigma(y_i)^2} = \frac{1}{y_i}$ and estimate confidence intervals for the least squares estimators by using the option `CovarianceMatrix`:

```
stats::linReg([[1, 1.0, 1/1.0], [2, 1.2, 1/1.2],
               [3, 1.3, 1/1.3], [4, 1.5, 1/1.5]], CovarianceMatrix)
```

$$\left[[0.8491476359, 0.1601801222], 0.001608234159, \begin{pmatrix} 1.650048247 & -0.5751045352 \\ -0.5751045352 & 0.2460598263 \end{pmatrix}\right]$$

The square roots of the diagonal elements of the covariance matrix provide standard deviations for the estimated parameters:

```
sqrt(%[3][1,1]), sqrt(%[3][2,2])
```

$$1.284542038, 0.4960441778$$

Thus, we obtain the estimates $a \pm \sigma(a) = 0.849 \pm 1.28$, $b \pm \sigma(b) = 0.16 \pm 0.496$.

Example 2

We create a sample consisting of one string column and two non-string columns:

```
stats::sample([["1", 0, 0], ["2", 10, 15], ["3", 20, 30]])
```

```
"1"    0    0
"2"   10   15
"3"   20   30
```

The least square estimators are calculated using the data columns 2 and 3. In this example there are only two non-string columns, so the column indices do not have to be specified:

```
stats::linReg(%)
```

$$\left[\left[0, \frac{3}{2}\right], 0\right]$$

Example 3

We create a sample consisting of three data columns:

```
stats::sample([[1, 0, 0], [2, 10, 15], [3, 20, 30]])
```

```
1    0    0
2   10   15
3   20   30
```

We compute the least square estimators for the data pairs given by the first and the second column:

```
stats::linReg(%, 1, 2)
```

$$[[-10, 10], 0]$$

Example 4

We create a sample of three columns containing symbolic data:

```
stats::sample([[x, y, 0], [2, 4, 15], [3, 20, 30]])
```

```
x   y   0
2   4  15
3  20  30
```

We compute the symbolic least square estimators for the data pairs given by the first and the second column. Here we specify these columns by a list of column indices:

```
map(stats::linReg(%, [1, 2], CovarianceMatrix), normal)
```

$$\left[\left[-\frac{68\,x-13\,y+5\,x\,y-24\,x^2+28}{\sigma_2},\ -\frac{24\,x+5\,y-2\,x\,y-84}{\sigma_2}\right],\ \frac{256\,x^2-32\,x\,y-896\,x+y^2+56\,y+784}{\sigma_2},\ \begin{pmatrix}\frac{x^2+13}{\sigma_2} & \sigma_1\\ \sigma_1 & \frac{3}{\sigma_2}\end{pmatrix}\right]$$

where

$$\sigma_1 = -\frac{x+5}{\sigma_2}$$

$$\sigma_2 = 2\,(x^2-5\,x+7)$$

Example 5

We create data (x_i, y_i) with a randomized relation $y_i = a + b\,x_i$:

```
DIGITS := 5:
r := stats::normalRandom(0, 5):
X := [i $ i = 0..100]:
Y := [12 + 17*x + r() $ x in X]:
```

By construction, the variances $\sigma(y_i)^2$ for the data y_i in the list Y is 5. We use the weights $w_i = \frac{1}{\sigma(y_i)^2} = \frac{1}{5}$ for all data:

```
W := [1/5 $ i = 0..100]:
[ab, chisquared, C]:= stats::linReg(X, Y, W, CovarianceMatrix)
```

$$\left[[12.065,\ 16.998],\ 87.997,\ \begin{pmatrix} \frac{335}{1717} & -\frac{5}{1717} \\ -\frac{5}{1717} & \frac{1}{17170} \end{pmatrix}\right]$$

The standard deviations of the estimators a, b are the square roots of the diagonal elements of C:

```
sqrt(float(C[1,1])), sqrt(float(C[2,2]))
```

$$0.44171,\ 0.0076316$$

Thus, the estimate for a is $a \pm \sigma(a) \approx 11.93 \pm 0.44$, the estimate for b is $b \pm \sigma(b) \approx 17.003 \pm 0.0076$.

```
delete r, X, Y, W, ab, chisquared, C:
```

Parameters

x_1, x_2, …

Statistical data: arithmetical expressions

y_1, y_2, …

Statistical data: arithmetical expressions

w_1, w_2, …

Weights: arithmetical expressions. If no weights are provided, $w_1 = w_2 = \ldots = 1$ is used.

s

A sample of domain type `stats::sample`.

cx, cy, cw

Integers representing column indices of the sample `s`. Column `cx` provides the data x_1, x_2, ..., column `cy` provides the data y_1, y_2, ... Column `cw`, if present, provides the weights w_1, w_2, ... If no index for the weights is provided, $w_1 = w_2 = \ldots = 1$ is used.

Options

CovarianceMatrix

Changes the return value from `[[a, b], chisquared]` to `[[a, b], chisquared, C]`, where C is the covariance matrix

$$\begin{pmatrix} \sigma(a)^2 & \mathrm{cov}(a, b) \\ \mathrm{cov}(a, b) & \sigma(b)^2 \end{pmatrix}$$

Of the estimators a, b.

With this option, information on confidence intervals for the least squares estimators are provided. In particular, the return value includes the covariance matrix

$$\begin{pmatrix} \sigma(a)^2 & \mathrm{cov}(a, b) \\ \mathrm{cov}(a, b) & \sigma(b)^2 \end{pmatrix}$$

Of type `Dom::Matrix()`. Assuming that the data (y_i) are randomly perturbed with stochastic variations $\sigma(y_i)^2$, the quadratic error to be minimized is

$$\chi^2 = \sum_i w_i \left| y_i - a - b\, x_i \right|^2$$

With

$$w_i = \frac{1}{\sigma(y_i)^2}$$

.

The covariance matrix of the least squares estimators is given by

$$\sigma(a)^2 = \frac{\sum_i w_i x_i^2}{\left(\sum_i w_i\right)\left(\sum_i w_i x_i^2\right) - \left(\sum_i w_i x_i\right)^2},$$

$$\sigma(b)^2 = \frac{\sum_i w_i}{\left(\sum_i w_i\right)\left(\sum_i w_i x_i^2\right) - \left(\sum_i w_i x_i\right)^2},$$

$$\mathrm{cov}(a, b) = \frac{\sum_i w_i x_i}{\left(\sum_i w_i\right)\left(\sum_i w_i x_i^2\right) - \left(\sum_i w_i x_i\right)^2}.$$

Return Values

Without the option `CovarianceMatrix`, a list `[[a, b], chisquared]` is returned. The arithmetical expressions a and b are estimators of the the offset and the slope of the linear relation. The arithmetical expression `chisquared` is the quadratic deviation

$$\chi^2 = \sum_i w_i \left| y_i - a - b\, x_i \right|^2,$$

where a, b are the optimized estimators.

With the option `CovarianceMatrix`, a list `[[a, b], chisquared, C]` is returned. The matrix `C` is the covariance matrix of the optimized estimators a and b.

`FAIL` is returned if the estimators a and b do not exist.

References

P.R. Bevington and D.K. Robinson, “Data Reduction and Error Analysis for The Physical Sciences”, McGraw-Hill, New York, 1992.

See Also

MuPAD Functions
`stats::reg` | `stats::sample`

MuPAD Graphical Primitives
`plot::Scatterplot`

stats::logisticCDF

Cumulative distribution function of the logistic distribution

Syntax

```
stats::logisticCDF(m, s)
```

Description

`stats::logisticCDF(m, s)` returns a procedure representing the cumulative distribution function

$$x \to \frac{1}{2}\left(1 + \tanh\left(\frac{\pi\,(x-m)}{2\sqrt{3}\,s}\right)\right)$$

of the logistic distribution with mean `m` and standard deviation `s` > 0 as a procedure.

The procedure `f := stats::logisticCDF(m, s)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

If `x` is a floating-point number and `m` and `s` can be converted to floating-point numbers, then `f(x)` returns a floating-point number between 0.0 and 1.0.

The call `f(- infinity )` returns `0`; the call `f( infinity )` returns `1`.

In all other cases, the expression `1/2*(1 + tanh(PI*(x - m)/(2*sqrt(3)*s)))` is returned symbolically.

Numerical values for `m` and `s` are only accepted if they are real and `s` is positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the cumulative distribution function with `m` = 0 and `s` = 1 at various points:

```
f := stats::logisticCDF(0, 1):
f(-infinity), f(-3), f(0.5), f(2/3), f(PI), f(infinity)
```

$$0,\ \frac{1}{2}-\frac{\tanh\left(\frac{\pi\sqrt{3}}{2}\right)}{2},\ 0.7123653231,\ \frac{\tanh\left(\frac{\pi\sqrt{3}}{9}\right)}{2}+\frac{1}{2},\ \frac{\tanh\left(\frac{\sqrt{3}\,\pi^2}{6}\right)}{2}+\frac{1}{2},\ 1$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::logisticCDF(m, s): f(x)
```

$$\frac{1}{2}-\frac{\tanh\left(\frac{\pi\sqrt{3}\,(m-x)}{6\,s}\right)}{2}$$

When numerical values are assigned to `m` and `s`, the function `f` starts to produce numerical values:

```
m := 0: s := 1: f(3), f(3.0)
```

$$\frac{\tanh\left(\frac{\pi\sqrt{3}}{2}\right)}{2}+\frac{1}{2},\ 0.995685277$$

```
delete f, m, s:
```

Parameters

`m`

The mean: an arithmetical expression representing a real value

s

The standard deviation: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`stats::logisticPDF` | `stats::logisticQuantile` | `stats::logisticRandom`

stats::logisticPDF

Probability density function of the logistic distribution

Syntax

```
stats::logisticPDF(m, s)
```

Description

`stats::logisticPDF(m, s)` returns the probability density function

$$x \to \frac{\pi}{4\,s\,\sqrt{3}}\,\mathrm{sech}\left(\frac{\pi\,(x-m)}{4\,\sqrt{3}\,s}\right)^2$$

of the logistic distribution with mean `m` and standard deviation `s` > 0.

The procedure `f := stats::logisticPDF(m, s)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

If `x` is a floating-point number and `m` and `s` can be converted to floating-point numbers, then `f(x)` returns a positive floating-point number.

`f( infinity )` and `f(- infinity )` return `0`.

In all other cases, the expression $\frac{\pi}{4\,s\,\sqrt{3}}\,\mathrm{sech}\left(\frac{\pi\,(x-m)}{4\,\sqrt{3}\,s}\right)^2$ is returned symbolically.

Numerical values for `m` and `s` are only accepted if they are real and `s` is positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the probability density function with `m` = 0 and `s` = 1 at various points:

```
f := stats::logisticPDF(0, 1): f(1/2), f(0.5), f(x)
```

$$\frac{\pi\sqrt{3}}{12\cosh\left(\frac{\pi\sqrt{3}}{12}\right)^2},\ 0.3716492483,\ \frac{\pi\sqrt{3}}{12\cosh\left(\frac{\pi\sqrt{3}\,x}{6}\right)^2}$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::logisticPDF(m, s): f(x), f(3)
```

$$\frac{\pi\sqrt{3}}{12\,s\cosh\left(\frac{\pi\sqrt{3}\,(m-x)}{6\,s}\right)^2},\ \frac{\pi\sqrt{3}}{12\,s\cosh\left(\frac{\pi\sqrt{3}\,(m-3)}{6\,s}\right)^2}$$

When numerical values are assigned to `m` and `s`, the function `f` starts to produce numerical values:

```
m := 0: s := 1: f(3), f(3.0)
```

$$\frac{\pi\sqrt{3}}{12\cosh\left(\frac{\pi\sqrt{3}}{2}\right)^2},\ 0.007792274633$$

```
delete f, m, s:
```

Parameters

`m`

The mean: an arithmetical expression representing a real value

s

The standard deviation: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`stats::logisticCDF` | `stats::logisticQuantile` | `stats::logisticRandom`

stats::logisticQuantile

Quantile function of the logistic distribution

Syntax

```
stats::logisticQuantile(m, s)
```

Description

`stats::logisticQuantile(m, s)` returns a procedure representing the quantile function (inverse)

$$x \to m + \frac{\sqrt{3}\,s}{\pi} \ln\left(\frac{x}{1-x}\right)$$

of the cumulative distribution function `stats::logisticCDF(m, s)`. For $0 \le x \le 1$, the solution of `stats::logisticCDF`$(m, s)(y) = x$ is given by

$$y = \text{stats::logisticQuantile}(m, s)(x)$$

.

The procedure `f := stats::logisticQuantile(m, s)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number, ±`infinity`, or a symbolic expression:

The call `f(x)` returns a real floating-point number if `x` is a floating-point number between `0.0` and `1.0`, `m` can be converted to a real floating-point number, and `s` can be converted to a positive real floating-point number.

The calls `f(0)` and `f(0.0)` produce `- infinity`; the calls `f(1)` and `f(1.0)` produce `infinity`.

In all other cases, the symbolic expression `m + sqrt(3)*s*ln(x/(1-x))/PI` is returned.

Numerical values of x are only accepted if $0 \le x \le 1$.

Numerical values of m and s are only accepted if they are real and s is positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the quantile function with mean `m` = 0 and standard deviation `s` = 1 at various points:

```
f := stats::logisticQuantile(0, 1):
f(0), f(1/10), f(0.7), f(0.999999999), f(1)
```

$$-\infty,\ -\frac{\sqrt{3}\,\ln(9)}{\pi},\ 0.4671397935,\ 11.42533526,\ \infty$$

The value `f(x)` satisfies `stats::logisticCDF(0, 1)(f(x))` = `x`:

```
stats::logisticCDF(0, 1)(f(0.987654321))
```

$$0.987654321$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::logisticQuantile(m, s): f(x), f(1/3), f(0.4)
```

$$m+\frac{\sqrt{3}\,s\,\ln\left(-\frac{x}{x-1}\right)}{\pi},\ m-\frac{\sqrt{3}\,s\,\ln(2)}{\pi},\ m-0.2235446302\,s$$

When suitable numerical values are assigned to `a` and `b`, the function `f` starts to produce numerical values:

```
m := 0: s := 1: f(0.999), f(999/1000)
```

$$3.807893483, \frac{\sqrt{3}\,\ln(999)}{\pi}$$

Numerical values for x are only accepted if $0 \le x \le 1$:

```
f(0.5)
```

$$0.0$$

```
f(2)
```

```
Error: An argument x with 0 <= x <= 1 is expected. [f]
```

```
delete f, m, s:
```

Parameters

`m`

The mean: an arithmetical expression representing a real value

`s`

The standard deviation: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`stats::logisticCDF` | `stats::logisticPDF` | `stats::logisticRandom`

stats::logisticRandom

Generate a random number generator for logistic deviates

Syntax

```
stats::logisticRandom(m, s, <Seed = s>)
```

Description

`stats::logisticRandom(m, s)` returns a procedure that produces logistic deviates (random numbers) with mean `m` and standard deviation `b` > 0.

The procedure `f := stats::logisticRandom(m, s)` can be called in the form `f()`. The return value of `f()` is either a floating-point number or a symbolic expression:

If `m` can be converted to a floating-point number and `s` can be converted to a positive floating point number, then `f(x)` returns a real floating point number.

Otherwise, `stats::logisticRandom(m, s)()` is returned symbolically.

Numerical values of `m` and `s` are only accepted if they are real and `s` is positive.

The values `X = f()` are distributed randomly according to the logistic distribution with mean `m` and standard deviation `s`. For any real `x`, the probability that $X \le x$ is given by

$$\frac{1+\tanh\left(\frac{\pi\,(x-m)}{2\sqrt{3}\,s}\right)}{2}.$$

Without the option `Seed = s`, an initial seed is chosen internally. This initial seed is set to a default value when MuPAD is started. Thus, each time MuPAD is started or re-initialized with the `reset` function, random generators produce the same sequences of numbers.

Note: In contrast to the function `random`, the generators produced by `stats::logisticRandom` do not react to the environment variable `SEED`.

For efficiency, it is recommended to produce sequences of K random numbers via

```
f := stats::logisticRandom(m, s): f() $k = 1..K;
```

rather than by

```
stats::logisticRandom(m, s)() $k = 1..K;
```

The latter call produces a sequence of generators each of which is called once. Also note that

```
stats::logisticRandom(m, s, Seed = n)() $k = 1..K;
```

does not produce a random sequence, because a sequence of freshly initialized generators would be created each of them producing the same number.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We generate logistic deviates with mean `m` = 0 and standard deviation `s` = 1:

```
f := stats::logisticRandom(0, 1): f() $ k = 1..4
```

$$-0.5473627217,\ 0.8782646344,\ -0.9428030153,\ 2.897981396$$

```
delete f:
```

Example 2

With symbolic parameters, no random floating-point numbers can be produced:

```
f := stats::logisticRandom(m, s): f()
```

```
stats::logisticRandom(m, s)()
```

When numbers are assigned to `m` and `s`, the function `f` starts to produce random floating point numbers:

```
m := PI: s := 1/8: f() $ k = 1..4
```

```
3.071738219, 3.037168533, 3.128342993, 3.193154296
```

```
delete f, m, s:
```

Example 3

We use the option `Seed = s` to reproduce a sequence of random numbers:

```
f := stats::logisticRandom(PI, 3, Seed = 1): f() $ k = 1..4
```

```
3.590026186, 5.349173335, -0.228813398, 0.5644575096
```

```
g := stats::logisticRandom(PI, 3, Seed = 1): g() $ k = 1..4
```

```
3.590026186, 5.349173335, -0.228813398, 0.5644575096
```

```
f() = g(), f() = g()
```

```
5.811492258 = 5.811492258, -6.396713726 = -6.396713726
```

```
delete f, g:
```

Parameters

`m`

The mean: an arithmetical expression representing a real value

`s`

The standard deviation: an arithmetical expression representing a positive real value

Options

Seed

Option, specified as `Seed = s`

Initializes the random generator with the integer seed `s`. s can also be the option `CurrentTime`, to make the seed depend on the current time.

This option serves for generating generators that return predictable sequences of pseudo-random numbers. The generator is initialized with the seed `s` which may be an arbitrary integer. Several generators with the same initial seed produce the same sequence of numbers.

When this option is used, the parameters `m` and `s` must be convertible to floating-point numbers at the time when the random generator is generated.

Return Values

procedure.

Algorithms

The implemented algorithm for the computation of the logistic deviates uses the quantile function of the logistic distribution applied to unformly distributed random numbers on the interval $[0, 1)$.

See Also

MuPAD Functions

`stats::logisticCDF` | `stats::logisticPDF` | `stats::logisticQuantile`

stats::lognormalCDF

Cumulative distribution function of the log-normal distribution

Syntax

```
stats::lognormalCDF(m, v)
```

Description

`stats::lognormalCDF(m, v)` returns a procedure representing the cumulative distribution function

$$x \to \frac{1}{\sqrt{2\,\pi\,v}} \int_{-\infty}^{x} e^{-\frac{\ln(t-m)^2}{2\,v}}\,dt$$

of the log-normal distribution with location parameter `m` and shape parameter `v`.

A random variable X is log-normally distributed if $ln(X)$ is a normally distributed variable. The "location parameter" m of X is the mean of $ln(X)$ and the "shape parameter" v is the variance of $ln(X)$.

The procedure `f := stats::lognormalCDF(m, v)` can be called in the form `f(x)` with an arithmetical expression `x`. The value $\frac{1}{2} + \frac{\mathrm{erf}\left(\frac{\ln(x)-m}{\sqrt{2\,v^3}}\right)}{2}$ is returned.

If x is a floating-point number and both m and v can be converted to floating-point numbers, this value is returned as a floating-point number. Otherwise, a symbolic expression is returned.

Numerical values for m and v are only accepted if they are real and v is positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the CDF of a lognormal distribution for some arbitrary parameter values:

```
f := stats::lognormalCDF(1/2, 3/4):
f(0.1), f(10.3)
```

$$0.0006057758986,\ 0.9828096232$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::lognormalCDF(m, v):
f(3), f(x)
```

$$\frac{1}{2}-\frac{\mathrm{erf}\left(\frac{\sqrt{2}\,(m-\ln(3))}{2\sqrt{v}}\right)}{2},\ \frac{1}{2}-\frac{\mathrm{erf}\left(\frac{\sqrt{2}\,(m-\ln(x))}{2\sqrt{v}}\right)}{2}$$

When numerical values are assigned to m and v, the function f starts to produce numerical values:

```
m := 4: v := PI:
f(3), f(3.0)
```

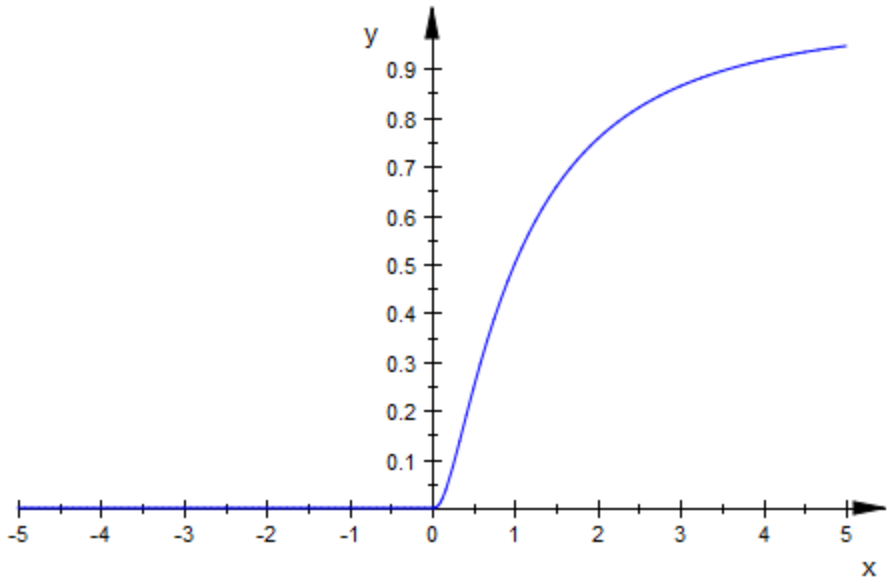
$$\frac{\mathrm{erf}\left(\frac{\sqrt{2}\,(\ln(3)-4)}{2\sqrt{\pi}}\right)}{2}+\frac{1}{2},\ 0.05082226366$$

```
delete f, m, v:
```

Example 3

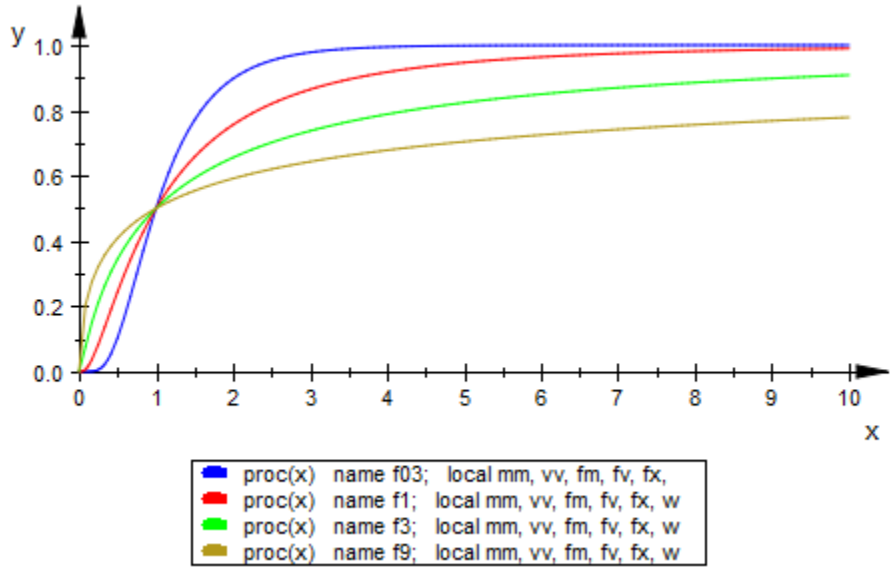
From the definition of “lognormal” above it is clear that the probability of $X < 0$ is zero for X lognormally distributed:

```
plotfunc2d(stats::lognormalCDF(0,1))
```



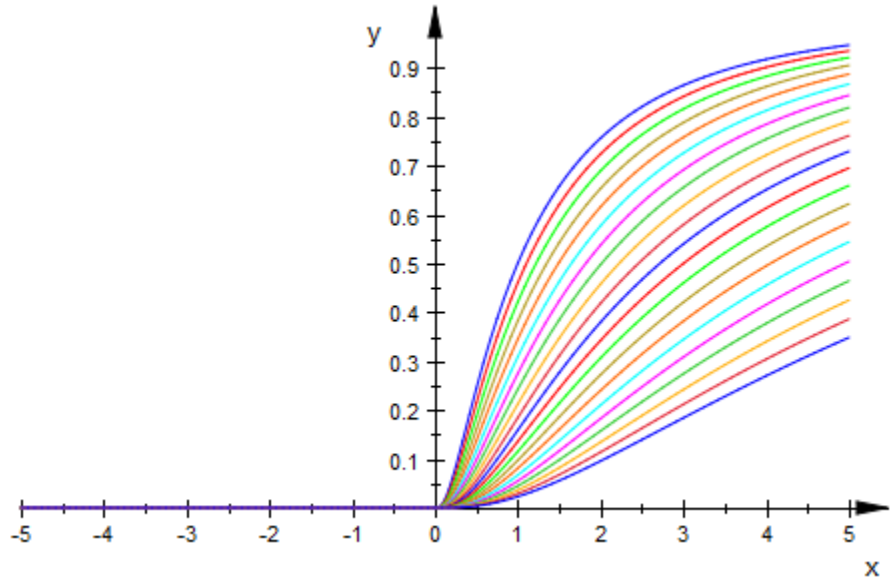
The following plot shows the influence of the shape parameter on the shape of the lognormal distribution:

```
f03 := stats::lognormalCDF(0, 0.3):
f1  := stats::lognormalCDF(0, 1):
f3  := stats::lognormalCDF(0, 3):
f9  := stats::lognormalCDF(0, 9):
plotfunc2d(f03, f1, f3, f9,  x = 0..10)
```



As for the normal distribution, a larger value of the shape parameter stretches the lognormal distribution, also changing its shape in the process:

```
plotfunc2d(stats::lognormalCDF(m, 1)$ m = 0..2 step .1,
           LegendVisible = FALSE)
```



Parameters

m

The location parameter: an arithmetical expression representing a real value

v

The shape parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`erf` | `stats::lognormalPDF` | `stats::lognormalQuantile` | `stats::lognormalRandom` | `stats::normalCDF`

stats::lognormalPDF

Probability density function of the log-normal distribution

Syntax

```
stats::lognormalPDF(m, v)
```

Description

`stats::lognormalPDF(m, v)` returns a procedure representing the probability density function

$$x \to \frac{e^{-\frac{(\ln(x)-m)^2}{2v}}}{\sqrt{2\pi v x}}$$

of the lognormal distribution with location parameter `m` and shape parameter `v`.

A random variable X is log-normally distributed if $ln(X)$ is a normally distributed variable. The "location parameter" m of X is the mean of $ln(X)$ and the "shape parameter" v is the variance of $ln(X)$.

The procedure `f := stats::lognormalPDF(m, v)` can be called in the form `f(x)` with an arithmetical expression `x`. The value $\frac{e^{-\frac{(\ln(x)-m)^2}{2v}}}{\sqrt{2\pi v x}}$ is returned.

If `x` is a floating-point number and both `m` and `v` can be converted to floating-point numbers, this value is returned as a floating-point number. Otherwise, a symbolic expression is returned.

Numerical values for m and v are only accepted if they are real and v is positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We compute the probability density with location parameter $m = 2$ and shape parameter $v = 4$ at various points:

```
f := stats::lognormalPDF(2, 4):
f(-infinity), f(-3), f(2.0), f(PI), f(infinity)
```

$$0,\ 0,\ 0.08056298881,\ \frac{\sqrt{2}\ e^{-\frac{(\ln(\pi)-2)^2}{8}}}{4\ \pi^{3/2}},\ 0$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::lognormalPDF(m, v):
f(x), f(0.4)
```

$$\frac{\sqrt{2}\ e^{-\frac{(m-\ln(x))^2}{2\,v}}}{2\ \sqrt{\pi}\ \sqrt{v}\ x},\ \frac{1.25\ \sqrt{2}\ e^{-\frac{(m+0.9162907319)^2}{2\,v}}}{\sqrt{\pi}\ \sqrt{v}}$$

When numerical values are assigned to m and v, the function f starts to produce numerical values:

```
m := PI: v := 2:
f(3), f(3.0)
```

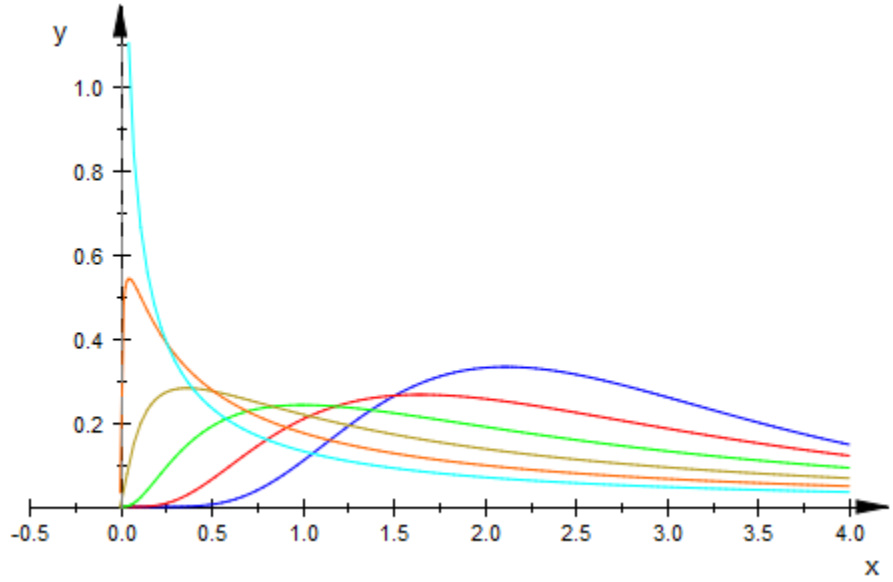
$$\frac{e^{-\frac{(\pi-\ln(3))^2}{4}}}{6\sqrt{\pi}},\ 0.03312170057$$

```
delete f, m, v:
```

Example 3

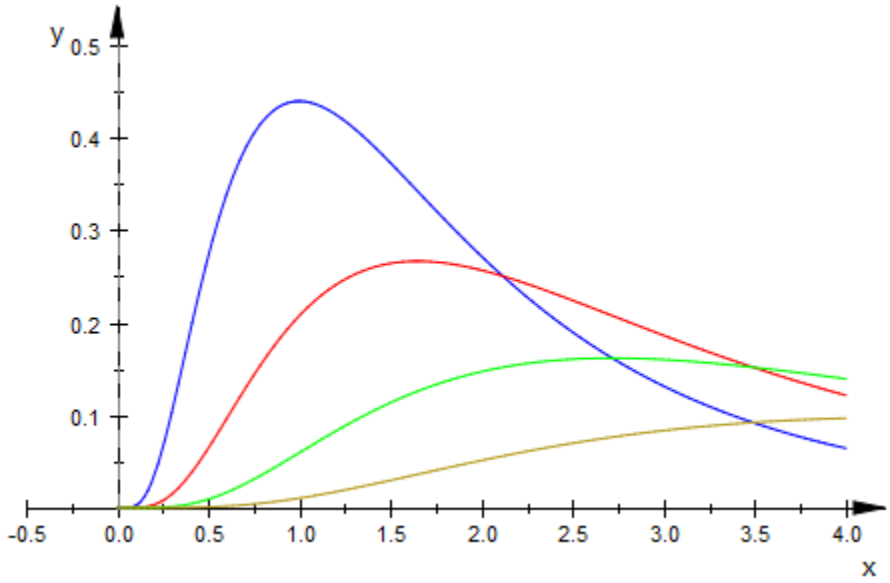
The following plot shows the influence of the shape parameter on the log-normal distribution:

```
plotfunc2d(stats::lognormalPDF(1, 0.25)(x),
           stats::lognormalPDF(1, 0.5)(x),
           stats::lognormalPDF(1, 1)(x),
           stats::lognormalPDF(1, 2)(x),
           stats::lognormalPDF(1, 4)(x),
           stats::lognormalPDF(1, 8)(x),
           x = -0.5 .. 4, ViewingBoxYRange = 0 .. 1.1,
           LegendVisible = FALSE)
```



Due to its logarithmic influence, the location parameter changes the shape of the distribution, too:

```
plotfunc2d(stats::lognormalPDF(m, 0.5)(x) $ m = 0.5..2 step 0.5,
           x = -0.5 ..4, ViewingBoxYRange = 0 .. 0.5,
           LegendVisible = FALSE)
```



Parameters

m

The location parameter: an arithmetical expression representing a real value

v

The shape parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`exp` | `stats::lognormalCDF` | `stats::lognormalQuantile` | `stats::lognormalRandom` | `stats::normalPDF`

stats::lognormalQuantile

Quantile function of the log-normal distribution

Syntax

```
stats::lognormalQuantile(m, v)
```

Description

`stats::normalQuantile(m, v)` returns a procedure representing the quantile function (inverse) of the cumulative distribution function `stats::lognormalCDF(m, v)` of the log-normal distribution with location parameter m and shape parameter $v > 0$: For $0 \le x \le 1$, the solution of *stats*::*lognormalCDF*(m, v)(y) = x is given by y = *stats*::*lognormalQuantile*(m, v)(x).

The procedure `f := stats::lognormalQuantile(m, v)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number, 0, `infinity`, or a symbolic expression:

If `x` is a real number between 0 and 1 and both `m` and `v` can be converted to floating-point numbers, then `f(x)` returns a real floating-point number approximating the solution `y` of `stats::lognormalCDF(m, v)(y) = x`.

The call `f(0)` returns `0`.

The calls `f(1)` and `f(1.0)` produce `infinity` for all values of `m` and `v`.

In all other cases, `f(x)` returns the symbolic call `stats::lognormalQuantile(m, v)(x)`.

Numerical values for m and v are only accepted if they are real and v is positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the quantile function with location parameter $m = \pi$ and shape parameter $v = 11$ at various points:

```
f := stats::lognormalQuantile(PI, 11):
f(0), f(1/10), f(0.5), f(1 - 10^(-10)), f(1)
```

$$0,\ 0.3299437681,\ 23.14069263,\ 33666650688.0,\ \infty$$

The value `f(x)` satisfies `stats::lognormalCDF(PI, 11)(f(x)) = x`:

```
stats::lognormalCDF(PI, 11)(f(0.987654))
```

$$0.987654$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::lognormalQuantile(m, v):
f(x), f(9/10)
```

$$\text{stats::lognormalQuantile}(m, v)(x),\ \text{stats::lognormalQuantile}(m, v)\left(\frac{9}{10}\right)$$

When numerical values are assigned to `m` and `v`, the function `f` starts to produce floating-point values:

```
m := 17: v := 6:
f(9/10), f(0.999)
```

$$557597210.5,\ 46816069055.0$$

Numerical values for x are only accepted if $0 \le x \le 1$:

```
f(0.5)
```

```
     24154952.75
```

```
f(2)
```

```
Error: An argument x with 0 <= x <= 1 is expected. [f]
```

```
delete f, m, v:
```

Parameters

m

The location parameter: an arithmetical expression representing a real value

v

The shape parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`stats::lognormalCDF` | `stats::lognormalPDF` | `stats::lognormalRandom`

stats::lognormalRandom

Generate a random number generator for log-normal deviates

Syntax

```
stats::lognormalRandom(m, v, <Seed = s>)
```

Description

`stats::normalRandom(m, v)` returns a procedure that produces lognormal deviates (random numbers) with location parameter m and shape parameter $v > 0$.

A random variable X is log-normally distributed if $ln(X)$ is a normally distributed variable. The "location parameter" m of X is the mean of $ln(X)$ and the "shape parameter" v is the variance of $ln(X)$.

The procedure `f := stats::lognormalRandom(m, v)` can be called in the form `f()`. The return value of `f()` is either a floating-point number or a symbolic expression:

If `m` and `v` can be converted to floating-point numbers, `f()` returns a real floating point number. Otherwise, the symbolic call `stats::lognormalRandom(m, v)()` is returned.

Numerical values of `m` and `v` are only accepted if they are real and v is positive.

The values `X = f()` are distributed randomly according to the cumulative distribution function of the log-normal distribution with parameters `m` and `v`. For any real x, the probability that $X \le x$ is given by

$$\frac{1}{2} + \frac{\operatorname{erf}\left(\frac{\ln(x) - m}{\sqrt{2\,v^3}}\right)}{2}.$$

Without the option `Seed = s`, an initial seed is chosen internally. This initial seed is set to a default value when MuPAD is started. Thus, each time MuPAD is started or re-initialized with the `reset` function, random generators produce the same sequences of numbers.

Note: In contrast to the function `random`, the generators produced by `stats::normalRandom` do not react to the environment variable `SEED`.

For efficiency, it is recommended to produce sequences of K random numbers via `f := stats::lognormalRandom(m, v): f() $k = 1..K` rather than by `stats::lognormalRandom(m, v)() $k = 1..K`. The latter call produces a sequence of generators each of which is called once. Also note that `stats::lognormalRandom(m, v, Seed = n)() $k = 1..K` does not produce a random sequence, because a sequence of freshly initialized generators would be created each of them producing the same number.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We generate log-normal deviates with location parameter 2 and shape parameter $\frac{3}{4}$:

```
f := stats::normalRandom(2, 3/4):
f() $ k = 1..4
```

```
1.541231663, 1.506864857, 1.553005641, 1.055326957
```

```
delete f:
```

Example 2

With symbolic parameters, no random floating-point numbers can be produced:

```
f := stats::lognormalRandom(m, v):
f()
```

stats::lognormalRandom(*m*, *v*)()

When *m* and *v* evaluate to real numbers, `f` starts to produce random floating point numbers:

```
m := PI/10: v := 1/8:
f() $ k = 1..4
```

2.57120706, 1.714950915, 1.747513452, 1.543972889

```
delete f, m, v:
```

Example 3

We use the option `Seed = s` to reproduce a sequence of random numbers:

```
f := stats::lognormalRandom(1, 3, Seed = 1):
f() $ k = 1..4
```

2.745783212, 0.566509847, 0.8352884053, 6.981716268

```
g := stats::lognormalRandom(1, 3, Seed = 1):
g() $ k = 1..4
```

2.745783212, 0.566509847, 0.8352884053, 6.981716268

```
f() = g(), f() = g()
```

2.891620861 = 2.891620861, 0.547178586 = 0.547178586

```
delete f, g:
```

Parameters

`m`

The location parameter: an arithmetical expression representing a real value

v

The shape parameter: an arithmetical expression representing a positive real value

Options

Seed

Option, specified as `Seed = s`

Initializes the random generator with the integer seed `s`. s can also be the option `CurrentTime`, to make the seed depend on the current time.

This option serves for generating generators that return predictable sequences of pseudo-random numbers. The generator is initialized with the seed `s` which may be an arbitrary integer. Several generators with the same initial seed produce the same sequence of numbers.

When this option is used, the parameters `m` and `v` must be convertible to suitable floating-point numbers at the time when the random generator is generated.

Return Values

procedure.

Algorithms

The implementation uses `stats::normalRandom`.

See Also

MuPAD Functions

`stats::lognormalCDF` | `stats::lognormalPDF` | `stats::lognormalQuantile`

stats::mean

Arithmetic mean of a data sample

Syntax

```
stats::mean(x1, x2, …)
```

```
stats::mean([x1, x2, …])
```

```
stats::mean(s, <c>)
```

Description

`stats::mean`(x_1, x_2, …, x_n) returns the arithmetic mean $\frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)$ of the data x_i.

The column index `c` is optional if the data are given by a `stats::sample` object containing only one non-string column. Cf. “Example 3” on page 30-324.

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

Examples

Example 1

We calculate the arithmetic mean of three values:

```
stats::mean(a, b, c)
```

$$\frac{a}{3}+\frac{b}{3}+\frac{c}{3}$$

Alternatively, data may be passed as a list:

```
stats::mean([2, 3, 5])
```

$$\frac{10}{3}$$

Example 2

We create a sample:

```
stats::sample([[a1, b1, c1], [a2, b2, c2]])
```

```
a1  b1  c1
a2  b2  c2
```

The arithmetic mean of the second column is:

```
stats::mean(%, 2)
```

$$\frac{b1}{2} + \frac{b2}{2}$$

Example 3

We create a sample consisting of one string column and one non-string column:

```
stats::sample([["1996", 1242], ["1997", 1353], ["1998", 1142]])
```

```
"1996"  1242
"1997"  1353
"1998"  1142
```

We compute the arithmetic mean of the second column. In this case, this column does not have to be specified, since it is the only non-string column:

```
float(stats::mean(%))
```

$$1245.666667$$

Parameters

x_1, x_2, …

The statistical data: arithmetical expressions.

s

A sample of domain type `stats::sample`.

c

An integer representing a column index of the sample `s`. This column provides the data `x1`, `x2` etc.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`stats::geometricMean` | `stats::harmonicMean` | `stats::median` | `stats::modal` | `stats::quadraticMean` | `stats::stdev` | `stats::variance`

stats::meandev

Mean deviation of a data sample

Syntax

```
stats::meandev(x1, x2, …)
```

```
stats::meandev([x1, x2, …])
```

```
stats::meandev(s, <c>)
```

Description

`stats::meandev( x1, x2, …, xn)` returns the mean deviation

$$\frac{1}{n}\left(\sum_{i=1}^{n}\left|x_i-\overline{x}\right|\right),$$

where $\overline{x}$ is the mean of the data x_i.

If all data are floating-point numbers, a float is returned. For symbolic data, the mean is returned as a symbolic expression.

The column index `c` is optional if the data are given by a `stats::sample` object containing only one non-string column.

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

Examples

Example 1

We calculate the mean deviation of some data:

```
stats::meandev(2, 33/7, PI)
```

$$\frac{104}{63} - \frac{2\,\pi}{9}$$

Alternatively, the data may be passed as a list:

```
data:=[2, 33/7, PI]: stats::meandev(data)
```

$$\frac{104}{63} - \frac{2\,\pi}{9}$$

If all data are floating-point numbers, the result is a float:

```
stats::meandev(float(data))
```

$$0.95266195$$

```
delete data:
```

Example 2

We create a sample of type `stats::sample`:

```
s := stats::sample([[22, 4, 1], [9, 8/3, 1], [2.0, 3, x]])
```

```
 22    4  1
  9  8/3  1
2.0    3  x
```

The mean deviations of the columns are computed:

```
stats::meandev(s, 1), stats::meandev(s, 2), stats::meandev(s, 3)
```

$$7.333333333,\ \frac{14}{27},\ \frac{2\left|\frac{x}{3} - \frac{1}{3}\right|}{3} + \frac{\left|\frac{2\,x}{3} - \frac{2}{3}\right|}{3}$$

```
delete s:
```

Example 3

With symbolic arguments, the mean deviation is returned as a symbolic expression:

```
stats::meandev(x1, x2, x3)
```

$$\frac{\left|\frac{x1}{3}+\frac{x2}{3}-\frac{2\,x3}{3}\right|}{3}+\frac{\left|\frac{x1}{3}-\frac{2\,x2}{3}+\frac{x3}{3}\right|}{3}+\frac{\left|\frac{x2}{3}-\frac{2\,x1}{3}+\frac{x3}{3}\right|}{3}$$

Parameters

x_1, x_2, …

The statistical data: arithmetical expressions

s

A sample of domain type stats::sample

c

A column index of the sample `s`: a positive integer. This column provides the data `x1`, `x2` etc.

Return Values

arithmetical expression.

See Also

MuPAD Functions

`stats::moment` | `stats::stdev` | `stats::variance`

stats::median

Median value of a data sample

Syntax

```
stats::median(x1, x2, …, <Averaged>)
```

```
stats::median([x1, x2, …], <Averaged>)
```

```
stats::median(s, <c>, <Averaged>)
```

Description

`stats::median`(x_1, x_2, …) returns the median of the data x_i.

The median of n sorted values $x_1 \le \ldots \le x_n$ is $x_{\lceil \frac{n}{2} \rceil}$.

The averaged median of n sorted values $x_1 \le \ldots \le x_n$ is $\frac{\left(x_{\lceil \frac{n}{2} \rceil} + x_{\lceil \frac{n+1}{2} \rceil}\right)}{2}$.

For odd n, both the median and the averaged median coincide with the element $x_{\frac{(n+1)}{2}}$ of the sorted data list. For even n, the median is $x_{\frac{n}{2}}$, whilst the averaged median is $\frac{\left(x_{\frac{n}{2}} + x_{\frac{(n+2)}{2}}\right)}{2}$.

The median coincides with the $\frac{1}{2}$-quantile of the data: the calls `stats::median(data <Averaged>)` and `stats::empiricalQuantile(data)(1/2 <Averaged>)` are equivalent. See the help page of `stats::empiricalQuantile` for details on the parameters specifying the data.

The column index `c` is optional if the data are given by a `stats::sample` object containing only one non-string column. Cf. “Example 3” on page 30-331.

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

Examples

Example 1

We calculate the median of a sequence of five values:

```
stats::median(3, 8, 5, 9/2, 11)
```

$$5$$

Alternatively, data may be passed as a list:

```
stats::median([2, 7, 3, 9/2, 11, 12]),
stats::median([2, 7, 3, 9/2, 11, 12], Averaged)
```

$$\frac{9}{2}, \frac{23}{4}$$

Example 2

We create a sample:

```
stats::sample([[4, 7, 5], [3, 6, 17], [8, 2, 2]])
```

```
4  7   5
3  6  17
8  2   2
```

The median of the second column is 6:

```
stats::median(%, 2)
```

$$6$$

Example 3

We create a sample consisting of one string column and one non-string column:

```
stats::sample([["1996", 1242], ["1997", 1353], ["1998", 1142]])
```

```
"1996"  1242
"1997"  1353
"1998"  1142
```

The median of the second column is calculated. In this case, there is no need to specify the index of the column, since it is the only non-string data column in the sample:

```
stats::median(%)
```

```
1242
```

Parameters

x_1, x_2, …

The statistical data: real numerical values.

s

A sample of domain type `stats::sample`

c

A column index of the sample `s`: a positive integer. This column provides the data `x1`, `x2` etc.

Options

Averaged

Return the averaged median value

Return Values

arithmetical expression. FAIL is returned if the data sample is empty.

See Also

MuPAD Functions

stats::empiricalQuantile | stats::geometricMean | stats::harmonicMean | stats::mean | stats::modal | stats::quadraticMean | stats::stdev | stats::variance

stats::modal

Modal (most frequent) value(s) in a data sample

Syntax

```
stats::modal(x1, x2, …)
```

```
stats::modal([x1, x2, …])
```

```
stats::modal(s, <c>)
```

Description

`stats::modal`(x_1, x_2, …) returns the most frequent value(s) of the data x_i.

The column index `c` is optional, if the data are given by a `stats::sample` object containing only one non-string column. Cf. “Example 3” on page 30-334.

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

Examples

Example 1

We calculate the modal value of a data sequence:

```
stats::modal(2, a, b, c, b, 10, 12, 2, b)
```

$$[b],\ 3$$

Alternatively, data may be passed as a list:

```
stats::modal([a, a, a, b, c, b, 10, 12, 2, b])
```

$[a, b], 3$

Example 2

We create a sample containing “age” and “gender”:

```
stats::sample([[32, "f"], [25, "m"], [40, "f"], [23, "f"]])
```

```
32  "f"
25  "m"
40  "f"
23  "f"
```

The modal value of the second column (the most frequent “gender”) is calculated:

```
stats::modal(%, 2)
```

$["f"], 3$

Example 3

We create a sample consisting of only one column:

```
stats::sample([4, 6, 2, 6, 8, 3, 2, 1, 7, 9, 3, 6, 5, 1, 6, 8]):
```

The modal value of these data is calculated. In this case, the column does not have to be specified, since there is only one column:

```
stats::modal(%)
```

$[6], 4$

Parameters

x_1, x_2, …

The statistical data: arithmetical expressions.

s

A sample of domain type `stats::sample`.

c

An integer representing a column index of the sample `s`. This column provides the data `x1`, `x2` etc.

Return Values

Sequence consisting of a list and an integer. The list contains the most frequent element(s) in the data, the integer specifies the number of occurrences. E.g., the result `[x5, x10], 21` means that `x5` and `x10` are the most frequent data items, each occurring 21 times.

See Also

MuPAD Functions

`stats::empiricalQuantile` | `stats::geometricMean` | `stats::harmonicMean` | `stats::mean` | `stats::median` | `stats::quadraticMean` | `stats::stdev` | `stats::variance`

stats::moment

The K-th moment of a data sample

Syntax

```
stats::moment(k, X, x₁, x₂, …)
```

```
stats::moment(k, X, [x₁, x₂, …])
```

```
stats::moment(k, X, s, <c>)
```

Description

`stats::moment`(k, X, [x_1, x_2, …, x_n]) returns the k-th moment

$$\frac{1}{n}\left(\sum_{i=1}^{n}(x_i - X)^k\right)$$

of the data x_i centered around `X`.

If k is an integer, rational or float, and all data `X`, x_1, x_2, … are floating-point numbers, then the moment is returned as a floating-point number. For symbolic data, a symbolic expression is returned.

The column index `c` is optional if the data are given by a `stats::sample` object containing only one non-string column.

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We calculate the third moment of some data centered around 0:

```
stats::moment(3, 0, 2, 33/7, 21/9, PI)
```

$$\frac{\pi^3}{4} + \frac{290509}{9261}$$

Alternatively, the data may be passed as a list:

```
data := [2, 33/7, 21/9, PI]: stats::moment(3, 0, data)
```

$$\frac{\pi^3}{4} + \frac{290509}{9261}$$

If all data are floating-point numbers, the result is a float:

```
data := float(data): stats::moment(3, 0, data)
```

$$39.12064378$$

We use `stats::moment` to compute the variance of the data:

```
m := stats::mean(data):
stats::moment(2, m, data) = stats::variance(data, Population)
```

$$1.098579542 = 1.098579542$$

```
delete data, m:
```

Example 2

We create a sample of type `stats::sample`:

```
s := stats::sample([[22, 4, 1], [9, 8/3, 1], [0.1, 2, 3]])
```

```
 22    4  1
  9  8/3  1
0.1    2  3
```

The fourth moment around a symbolic center X is computed for all columns in the sample:

```
stats::moment(4, X, s, i) $ i = 1..3
```

$$\frac{(X-9)^4}{3}+\frac{(X-22)^4}{3}+\frac{(X-0.1)^4}{3},\ \frac{(X-2)^4}{3}+\frac{(X-4)^4}{3}+\frac{\left(X-\frac{8}{3}\right)^4}{3},\ \frac{2\,(X-1)^4}{3}+\frac{(X-3)^4}{3}$$

```
delete s:
```

Example 3

For symbolic arguments, the moment is returned as a symbolic expression:

```
stats::moment(k, X, [x1, x2, x3, x4])
```

$$\frac{(\mathrm{x1}-X)^k}{4}+\frac{(\mathrm{x2}-X)^k}{4}+\frac{(\mathrm{x3}-X)^k}{4}+\frac{(\mathrm{x4}-X)^k}{4}$$

Parameters

k

An arithmetical expression

X

The center: an arithmetical expression

x_1, x_2, …

The statistical data: arithmetical expressions

s

A sample of domain type stats::sample

c

A column index of the sample `s`: a positive integer. This column provides the data x_1, x_2, ….

Return Values

arithmetical expression.

See Also

MuPAD Functions

`stats::quadraticMean` | `stats::variance`

stats::normalCDF

Cumulative distribution function of the normal distribution

Syntax

```
stats::normalCDF(m, v)
```

Description

`stats::normalCDF(m, v)` returns a procedure representing the cumulative distribution function

$$x \to \frac{1}{\sqrt{2\,\pi\,v}} \int_{-\infty}^{x} e^{-\frac{(t-m)^2}{2\,v}}\,dt$$

of the normal distribution with mean `m` and variance `v`.

The procedure `f := stats::normalCDF(m, v)` can be called in the form `f(x)` with an arithmetical expression `x`. The value `1/2 + 1/2* erf((x - m)/sqrt(2*v))` is returned.

If x is a floating-point number and both m and v can be converted to floating-point numbers, this value is returned as a floating-point number. Otherwise, a symbolic expression is returned.

Numerical values for m and v are only accepted if they are real and v is positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the cumulative distribution function with mean $m = 2$ and variance $v = \frac{3}{4}$ at various points:

```
f := stats::normalCDF(2, 3/4):
f(-infinity), f(-3), f(PI), f(infinity)
```

$$0,\ \frac{1}{2}-\frac{\mathrm{erf}\left(\frac{5\sqrt{2}\sqrt{3}}{3}\right)}{2},\ \frac{\mathrm{erf}\left(\frac{\sqrt{2}\sqrt{3}(\pi-2)}{3}\right)}{2}+\frac{1}{2},\ 1$$

```
f(-100.0), f(-3.0), f(float(PI)), f(10.0), f(100.0)
```

$$1.833507721\ 10^{-3015},\ 0.000000003882018269,\ 0.9062812543,\ 1.0,\ 1.0$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::normalCDF(m, v): f(3), f(x)
```

$$\frac{1}{2}-\frac{\mathrm{erf}\left(\frac{\sqrt{2}(m-3)}{2\sqrt{v}}\right)}{2},\ \frac{1}{2}-\frac{\mathrm{erf}\left(\frac{\sqrt{2}(m-x)}{2\sqrt{v}}\right)}{2}$$

When numerical values are assigned to m and v, the function f starts to produce numerical values:

```
m := 4: v := PI: f(3), f(3.0)
```

$$\frac{1}{2}-\frac{\mathrm{erf}\left(\frac{\sqrt{2}}{2\sqrt{\pi}}\right)}{2},\ 0.286312558$$

```
delete f, m, v:
```

Parameters

m

The mean: an arithmetical expression representing a real value

v

The variance: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`erf` | `stats::normalPDF` | `stats::normalQuantile` | `stats::normalRandom`

stats::normalPDF

Probability density function of the normal distribution

Syntax

```
stats::normalPDF(m, v)
```

Description

`stats::normalPDF(m, v)` returns a procedure representing the probability density function

$$x \to \frac{e^{\frac{1}{\frac{(x-m)^2}{2v}}}}{\sqrt{2\pi v}}$$

of the normal distribution with mean `m` and variance `v`.

The procedure `f := stats::normalPDF(m, v)` can be called in the form `f(x)` with an arithmetical expression `x`. The value `exp(-(x - m)^2/(2*v))/sqrt(2*PI*v)` is returned.

If `x` is a floating-point number and both `m` and `v` can be converted to floating-point numbers, this value is returned as a floating-point number. Otherwise, a symbolic expression is returned.

Numerical values for m and v are only accepted if they are real and v is positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We compute the probability density with mean $m = 2$ and variance $v = 4$ at various points:

```
f := stats::normalPDF(2, 4):
f(-infinity), f(-3), f(2.0), f(PI), f(infinity)
```

$$0,\ \frac{\sqrt{2}\,e^{-\frac{25}{8}}}{4\sqrt{\pi}},\ 0.1994711402,\ \frac{\sqrt{2}\,e^{-\frac{(\pi-2)^2}{8}}}{4\sqrt{\pi}},\ 0$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::normalPDF(m, v): f(x), f(0.4)
```

$$\frac{\sqrt{2}\,e^{-\frac{(m-x)^2}{2v}}}{2\sqrt{\pi}\sqrt{v}},\ \frac{\sqrt{2}\,e^{-\frac{(m-0.4)^2}{2v}}}{2\sqrt{\pi}\sqrt{v}}$$

When numerical values are assigned to m and v, the function f starts to produce numerical values:

```
m := PI: v := 2: f(3), f(3.0)
```

$$\frac{e^{-\frac{(\pi-3)^2}{4}}}{2\sqrt{\pi}},\ 0.2806844362$$

```
delete f, m, v:
```

Parameters

m

The mean: an arithmetical expression representing a real value

v

The variance: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`exp` | `stats::normalCDF` | `stats::normalQuantile` | `stats::normalRandom`

stats::normalQuantile

Quantile function of the normal distribution

Syntax

```
stats::normalQuantile(m, v)
```

Description

`stats::normalQuantile(m, v)` returns a procedure representing the quantile function (inverse) of the cumulative distribution function `stats::normalCDF(m, v)` of the normal distribution with mean m and variance $v > 0$: For $0 \le x \le 1$, the solution of *stats*::*normalCDF*(m, v)(y) = x is given by y = *stats*::*normalQuantile*(m, v)(x).

The procedure `f := stats::normalQuantile(m, v)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number, ±`infinity`, or a symbolic expression:

If `x` is a real number between `0` and `1` and both `m` and `v` can be converted to floating-point numbers, then `f(x)` returns a real floating-point number approximating the solution `y` of `stats::normalCDF(m, v)(y) = x`.

The calls `f(0)` and `f(0.0)` produce `-infinity` for all values of `m` and `v`.

The calls `f(1)` and `f(1.0)` produce `infinity` for all values of `m` and `v`.

In all other cases, `f(x)` returns the symbolic call `stats::normalQuantile(m, v)(x)`.

Numerical values for m and v are only accepted if they are real and v is positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the quantile function with mean $m = \pi$ and variance $v = 11$ at various points:

```
f := stats::normalQuantile(PI, 11):
f(0), f(1/10), f(0.5), f(1 - 10^(-10)), f(1)
```

$$-\infty,\ -1.108833039,\ 3.141592654,\ 24.23977359,\ \infty$$

The value `f(x)` satisfies `stats::normalCDF(PI, 11)(f(x)) = x`:

```
stats::normalCDF(PI, 11)(f(0.987654))
```

$$0.987654$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::normalQuantile(m, v): f(x), f(9/10)
```

$$\text{stats::normalQuantile}(m, v)(x),\ \text{stats::normalQuantile}(m, v)\left(\frac{9}{10}\right)$$

When numerical values are assigned to `m` and `v`, the function `f` starts to produce floating-point values:

```
m := 17: v := 6: f(9/10), f(0.999)
```

$$20.13914741,\ 24.56949234$$

Numerical values for x are only accepted if $0 \le x \le 1$:

```
f(0.5)
```

```
     17.0

f(2)

Error: An argument x with 0 <= x <= 1 is expected. [f]

delete f, m, v:
```

Parameters

m

The mean: an arithmetical expression representing a real value

v

The variance: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`stats::normalCDF` | `stats::normalPDF` | `stats::normalRandom`

stats::normalRandom

Generate a random number generator for normal deviates

Syntax

```
stats::normalRandom(m, v, <Seed = s>)
```

Description

`stats::normalRandom(m, v)` returns a procedure that produces normal deviates (random numbers) with mean m and variance $v > 0$.

The procedure `f := stats::normalRandom(m, v)` can be called in the form `f()`. The return value of `f()` is either a floating-point number or a symbolic expression:

If `m` and `v` can be converted to floating-point numbers, `f()` returns a real floating point number. Otherwise, the symbolic call `stats::normalRandom(m, v)()` is returned.

Numerical values of `m` and `v` are only accepted if they are real and v is positive.

The values `X = f()` are distributed randomly according to the cumulative distribution function of the normal distribution with parameters `m` and `v`. For any real x, the probability that $X \le x$ is given by

$$\frac{1}{\sqrt{2\,\pi\,v}} \int_{-\infty}^{x} e^{-\frac{(t-m)^2}{2\,v}}\, dt$$

.

Without the option `Seed = s`, an initial seed is chosen internally. This initial seed is set to a default value when MuPAD is started. Thus, each time MuPAD is started or re-initialized with the `reset` function, random generators produce the same sequences of numbers.

Note: In contrast to the function `random`, the generators produced by `stats::normalRandom` do not react to the environment variable `SEED`.

For efficiency, it is recommended to produce sequences of K random numbers via

```
f := stats::normalRandom(m, v): f() $k = 1..K;
```

rather than by

```
stats::normalRandom(m, v)() $k = 1..K;
```

The latter call produces a sequence of generators each of which is called once. Also note that

```
stats::normalRandom(m, v, Seed = n)() $k = 1..K;
```

does not produce a random sequence, because a sequence of freshly initialized generators would be created each of them producing the same number.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We generate normal deviates with mean 2 and variance $\frac{3}{4}$:

```
f := stats::normalRandom(2, 3/4): f() $ k = 1..4
```

$$1.541231663,\ 1.506864857,\ 1.553005641,\ 1.055326957$$

```
delete f:
```

Example 2

With symbolic parameters, no random floating-point numbers can be produced:

```
f := stats::normalRandom(m, v): f()
```

$$\text{stats::normalRandom}(m, v)()$$

When m and v evaluate to real numbers, `f` starts to produce random floating point numbers:

```
m := PI: v := 1/8: f() $ k = 1..4
```

$$3.77180885,\ 3.366817847,\ 3.385627281,\ 3.261792281$$

```
delete f, m, v:
```

Example 3

We use the option `Seed = s` to reproduce a sequence of random numbers:

```
f := stats::normalRandom(PI, 3, Seed = 1): f() $ k = 1..4
```

$$3.151659011,\ 1.573331837,\ 1.961614435,\ 4.084887424$$

```
g := stats::normalRandom(PI, 3, Seed = 1): g() $ k = 1..4
```

$$3.151659011,\ 1.573331837,\ 1.961614435,\ 4.084887424$$

```
f() = g(), f() = g()
```

$$3.20340985 = 3.20340985,\ 1.538612606 = 1.538612606$$

```
delete f, g:
```

Parameters

`m`

The mean: an arithmetical expression representing a real value

v

The variance: an arithmetical expression representing a positive real value

Options

Seed

Option, specified as `Seed = s`

Initializes the random generator with the integer seed `s`. s can also be the option `CurrentTime`, to make the seed depend on the current time.

This option serves for generating generators that return predictable sequences of pseudo-random numbers. The generator is initialized with the seed `s` which may be an arbitrary integer. Several generators with the same initial seed produce the same sequence of numbers.

When this option is used, the parameters `m` and `v` must be convertible to suitable floating-point numbers at the time when the random generator is generated.

Return Values

procedure.

Algorithms

The implemented algorithm for the computation of the normal deviates uses the Box-Mueller method. For more information see: D. Knuth, Seminumerical Algorithms (1998), Vol. 2, pp. 122.

See Also

MuPAD Functions

`stats::normalCDF` | `stats::normalPDF` | `stats::normalQuantile`

stats::obliquity

Obliquity (skewness) of a data sample

Syntax

```
stats::obliquity(x₁, x₂, …)
```

```
stats::obliquity([x₁, x₂, …])
```

```
stats::obliquity(s, <c>)
```

Description

stats::*obliquity*(x_1, x_2, …, x_n) returns the obliquity (skewness)

$$\frac{\frac{1}{n}\left(\sum_{i=1}^{n}(x_i-\overline{x})^3\right)}{\left(\frac{1}{n}\left(\sum_{i=1}^{n}(x_i-\overline{x})^2\right)\right)^{3/2}},$$

where $\overline{x}$ is the mean of the data x_i.

The obliquity is a measure for the symmetry of a distribution. It is zero, if the distribution of the data is symmetric around the mean. Positive values indicate that the distribution function has a “longer tail” to the right of the mean than to the left. Negative values indicate a “longer tail” to the left.

The column index `c` is optional, if the data are given by a `stats::sample` object containing only one non-string column. Cf. “Example 3” on page 30-354.

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

Examples

Example 1

We calculate the obliquity of a data sequence:

```
float(stats::obliquity(0, 7, 7, 6, 6, 6, 5, 5, 4, 1))
```

$$-1.041368312$$

Alternatively, data may be passed as a list:

```
stats::obliquity([2, 2, 4, 6, 8, 10, 10])
```

$$0$$

Example 2

We create a sample:

```
stats::sample([[a, 5, 8], [b, 3, 7], [c, d, 0]])
```

a	5	8
b	3	7
c	d	0

The obliquity of the second column is:

```
stats::obliquity(%, 2)
```

$$-\frac{\sqrt{3}\left(\left(\frac{d}{3}-\frac{1}{3}\right)^3+\left(\frac{d}{3}-\frac{7}{3}\right)^3-\left(\frac{2\,d}{3}-\frac{8}{3}\right)^3\right)}{\left(\left(\frac{d}{3}-\frac{1}{3}\right)^2+\left(\frac{d}{3}-\frac{7}{3}\right)^2+\left(\frac{2\,d}{3}-\frac{8}{3}\right)^2\right)^{3/2}}$$

Example 3

We create a sample consisting of one string column and one non-string column:

```
stats::sample([["1996", 1242], ["1997", 1353], ["1998", 1142]])
```

```
"1996"  1242
"1997"  1353
"1998"  1142
```

We compute the obliquity of the second column. In this case this column does not have to be specified, since it is the only non-string column:

```
float(stats::obliquity(%))
```

$$0.06374333648$$

Parameters

x_1, x_2, …

The statistical data: arithmetical expressions.

s

A sample of domain type `stats::sample`.

c

An integer representing a column index of the sample `s`. This column provides the data `x1`, `x2` etc.

Return Values

Arithmetical expression. `FAIL` is returned, if the obliquity does not exist.

See Also

MuPAD Functions

`stats::kurtosis`

stats::poissonCDF

The (discrete) cumulative distribution function of the Poisson distribution

Syntax

```
stats::poissonCDF(m)
```

Description

`stats::poissonCDF(m)` returns a procedure representing the (discrete) cumulative distribution function

$$x \to \begin{cases} 0 & \text{if } x < 0 \\ \sum_{i=0}^{\lfloor x \rfloor} \frac{m^i e^{-m}}{i!} & \text{if } x \geq 0 \end{cases}$$

of the Poisson distribution with mean `m`.

The procedure `f := stats::poissonCDF(m)` can be called in the form `f(x)` with arithmetical expressions `x`. The return value of `f(x)` is either a floating-point number, an exact numerical value, or a symbolic expression:

If `x` is a numerical real value, then an explicit value is returned. It is a floating-point number if `x` is a floating-point number and `m` can be converted to a positive real float. Otherwise, an exact expression is returned.

If `x` is a numerical value `< 0`, then `0`, respectively `0.0`, is returned for any value of `m`.

For symbolic values of `x`, `f(x)` returns the symbolic call `stats::poissonCDF(m)(x)`.

Numerical values for `m` are only accepted if they are nonnegative.

If `x` is a real floating-point number, the result is a floating number provided *m* is a nonnegative numerical value. If both `x` and `m` are exact numerical values, the result is an exact number.

Note: Note that for large m, floating-point results are computed much faster than exact results. If floating-point approximations are desired, pass a floating-point number `x` to `stats::poissonCDF`!

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the distribution function with `m` = $\frac{1}{2}$ at various points:

```
f := stats::poissonCDF(1/2):
f(-PI) = f(float(-PI)), f(0) = f(0.0), f(4) = f(4.0)
```

$$0 = 0.0,\ e^{-\frac{1}{2}} = 0.6065306597,\ \frac{211\ e^{-\frac{1}{2}}}{128} = 0.9998278844$$

```
delete f:
```

Example 2

We use symbolic arguments. If `x` is symbolic, a symbolic call is returned:

```
f := stats::poissonCDF(m): f(x)
```

$$\text{stats::poissonCDF}(m)(x)$$

If `x` is a numerical value, symbolic expressions in `m` are returned:

```
f(-1), f(0), f(5/2), f(PI)
```

$$0,\ e^{-m},\ e^{-m}\left(\frac{m^2}{2}+m+1\right),\ e^{-m}\left(\frac{m^3}{6}+\frac{m^2}{2}+m+1\right)$$

When numerical values are assigned to m, the function f starts to produce explicit results if the argument is numerical:

```
m := 3: f(-1), f(0), f(5/2), f(PI)
```

$$0,\ e^{-3},\ \frac{17\,e^{-3}}{2},\ 13\,e^{-3}$$

```
delete f, m:
```

Parameters

m

The mean: an arithmetical expression representing a nonnegative real number

Return Values

procedure.

See Also

MuPAD Functions

`stats::poissonPF` | `stats::poissonQuantile` | `stats::poissonRandom`

stats::poissonPF

Probability function of the Poisson distribution

Syntax

```
stats::poissonPF(m)
```

Description

`stats::poissonPF(m)` returns a procedure representing the probability function

$$x \to \frac{\frac{m^x}{e^m}}{x!}$$

of the Poisson distribution with mean `m`.

The procedure `f := stats::poissonPF(m)` can be called in the form `f(x)` with arithmetical expressions `x`. The return value of `f(x)` is either a floating-point number, an exact numerical value, or a symbolic expression:

If `x` is a non-integer numerical value, *f*(*x*) returns `0` or `0.0`, respectively.

If `x` is an integer or the floating-point equivalent of an integer, then an explicit value is returned.

In all other cases, `f(x)` returns the symbolic call `stats::poissonPF(n,p)(x)`.

Numerical values for `m` are only accepted if they are nonnegative.

If `x` is a floating-point number, the result is a floating-point number provided *m* is a nonnegative numerical value. If both `x` and `m` are exact values then the result is an exact number.

Note that for large *m*, floating-point results are computed much faster than exact results. If floating-point approximations are desired, pass a floating-point number `x` to the procedure generated by `stats::poissonPF`.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We calculate the Poisson probability with $m = 8$ at various points:

```
f := stats::poissonPF(8):
f(-1), f(-1.0), f(0), f(1/2),  f(1), f(3/2), f(3) = f(float(3))
```

$$0,\ 0.0,\ e^{-8},\ 0,\ 8\,e^{-8},\ 0,\ \frac{256\,e^{-8}}{3} = 0.02862614425$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::poissonPF(m): f(x)
```

$$\text{stats::poissonPF}(m)(x)$$

If `x` is a numerical value, symbolic expressions in `m` are returned:

```
f(8), f(17/2), f(9.0), f(9.2)
```

$$\frac{m^8\,e^{-m}}{40320},\ 0,\ 0.000002755731922\,m^{9.0}\,e^{-1.0\,m},\ 0.0$$

When numerical values are assigned to m, the function f starts to produce numbers if the argument is numerical:

```
m := 3: f(8), f(17/2), f(9.0), f(9.2)
```

$$\frac{729\,e^{-3}}{4480},\ 0,\ 0.002700503932,\ 0.0$$

```
delete f, m:
```

Parameters

m

The mean: an arithmetical expression representing a nonnegative real number

Return Values

procedure.

See Also

MuPAD Functions

`stats::poissonCDF` | `stats::poissonQuantile` | `stats::poissonRandom`

stats::poissonQuantile

Quantile function of the Poisson distribution

Syntax

```
stats::poissonQuantile(m)
```

Description

`stats::poissonQuantile(m)` returns a procedure representing the quantile function (discrete inverse) of the cumulative distribution function `stats::poissonCDF(m)`. For $0 \le x \le 1$, k = `stats::poissonQuantile(m)(x)` is the smallest nonnegative integer satisfying

$$\text{stats::poissonCDF}(m)(k) = \sum_{i=0}^{k} \frac{\frac{m^i}{e^m}}{i!} \ge x$$

.

The procedure `f := stats::poissonQuantile(m)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of the call `f(x)` is either a nonnegative integer, `infinity`, or a symbolic expression:

If m is a nonnegative real number and x a real number satisfying $0 \le x < 1$, then $f(x)$ returns a nonnegative integer.

If $m = 0$, then $f(x)$ returns `0` for any x.

If $m \neq 0$, then $f(1)$ and $f(1.0)$ return `infinity`.

In all other cases, `f(x)` returns the symbolic call `stats::poissonQuantile(m)(x)`.

Numerical values for `m` are only accepted if they are positive.

If floating-point arguments are passed to the quantile function f, the result is computed with floating-point arithmetic. This is faster than using exact arithmetic, but the result is subject to internal round-off errors. In particular, round-off may be significant for arguments x close to 1. Cf. “Example 3” on page 30-364.

Finite quantile values k = `stats::poissonQuantile(m)(x)` satisfy

$$\text{stats::poissonCDF}(m)(k-1) < x \le \text{stats::poissonCDF}(m)(k)$$

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the quantile function with $m = \pi$ at various points:

```
f := stats::poissonQuantile(PI):
f(0), f(1/20), f(0.3), f(PI/6), f(0.7), f(1-1/10^10), f(1)
```

$$0,\ 1,\ 2,\ 3,\ 4,\ 20,\ \infty$$

The value `f(x)` satisfies

$$\text{stats::poissonCDF}(\pi)(f(x)-1) < x \le \text{stats::poissonCDF}(\pi)(f(x))$$

```
x := 0.98: k := f(x)
```

$$7$$

```
float(stats::poissonCDF(PI)(k - 1)), x,
float(stats::poissonCDF(PI)(k))
```

$$0.9588410737,\ 0.98,\ 0.9847376421$$

```
delete f, x, k:
```

Example 2

We use symbolic arguments:

```
f := stats::poissonQuantile(m): f(x), f(9/10)
```

$$\text{stats::poissonQuantile}(m)(x),\ \text{stats::poissonQuantile}(m)\left(\frac{9}{10}\right)$$

When `m` evaluates to a positive real number, the function `f` starts to produce quantile values:

```
m := 17:
f(1/2),  f(999/1000), f(1 - 1/10^10), f(1 - 1/10^80)
```

$$17,\ 31,\ 49,\ 144$$

```
delete f, m:
```

Example 3

If floating-point arguments are passed to the quantile function, the result is computed with floating-point arithmetic. This is faster than using exact arithmetic, but the result is subject to internal round-off errors:

```
f := stats::poissonQuantile(123):
f(1 - 1/10^19) <> f(float(1 - 1/10^19))
```

$$236 \neq \infty$$

```
delete f:
```

Parameters

`m`

The mean: a arithmetical expression representing a nonnegative real number

Return Values

procedure.

See Also

MuPAD Functions

`stats::poissonCDF` | `stats::poissonPF` | `stats::poissonRandom`

stats::poissonRandom

Generate a random number generator for Poisson deviates

Syntax

```
stats::poissonRandom(m, <Seed = s>)
```

Description

`stats::poissonRandom(m)` returns a procedure that produces poisson-deviates (random numbers) with mean m.

The procedure `f := stats::poissonRandom(m)` can be called in the form `f()`. The return value of `f()` is a nonnegative integer if `m` is a nonnegative numerical value.

Otherwise, `stats::poissonRandom(m)()` is returned symbolically.

Numerical values for `m` are only accepted if they are nonnegative.

The values `X = f()` are distributed randomly according to the discrete distribution function of the Poisson distribution with mean `m`, i.e., for $0 \le x$, the probability of $X \le x$ is given by

$$\sum_{i=0}^{\lfloor x \rfloor} \frac{\frac{m^i}{e^m}}{i!}$$

.

Without the option `Seed = s`, an initial seed is chosen internally. This initial seed is set to a default value when MuPAD is started. Thus, each time MuPAD is started or re-initialized with the `reset` function, random generators produce the same sequences of numbers.

Note: With this option, the mean `m` must evaluate to a nonnegative numerical value at the time, when the generator is created.

Note: In contrast to the function `random`, the generators produced by `stats::poissonRandom` do not react to the environment variable `SEED`.

For efficiency, it is recommended to produce sequences of K random numbers via

```
f := stats::poissonRandom(m): f() $k = 1..K;
```

rather than by

```
stats::poissonRandom(m)() $k = 1..K;
```

The latter call produces a sequence of generators each of which is called once. Also note that

```
stats::poissonRandom(m, Seed = s)() $k = 1..K;
```

does not produce a random sequence, because a sequence of freshly initialized generators would be created each of them producing the same number.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We generate Poisson deviates with mean $m = 80$:

```
f := stats::poissonRandom(80): f() $ k = 1..10
```

```
69, 105, 77, 81, 71, 79, 86, 77, 87, 97
```

```
delete f:
```

Example 2

With symbolic parameters, no random floating-point numbers can be produced:

```
f := stats::poissonRandom(m): f()
```

$$\text{stats::poissonRandom}(m)()$$

When m evaluates to a positive real number, the generator starts to produce random numbers:

```
m := 80: f(), f(), f()
```

$$89, 76, 82$$

```
delete f, m:
```

Example 3

We use the option `Seed = s` to reproduce a sequence of random numbers:

```
f := stats::poissonRandom(12, Seed = 1): f() $ k = 1..10
```

$$12, 15, 8, 9, 15, 4, 12, 9, 7, 10$$

```
g := stats::poissonRandom(12, Seed = 1): g() $ k = 1..10
```

$$12, 15, 8, 9, 15, 4, 12, 9, 7, 10$$

```
f() = g(), f() = g()
```

$$11 = 11, 11 = 11$$

```
delete f, g:
```

Parameters

`m`

The mean: an arithmetical expression representing a nonnegative real number

Options

`Seed`

Option, specified as `Seed = s`

Initializes the random generator with the integer seed `s`. s can also be the option `CurrentTime`, to make the seed depend on the current time.

This option serves for generating generators that return predictable sequences of pseudo-random numbers. The generator is initialized with the seed `s` which may be an arbitrary integer. Several generators with the same initial seed produce the same sequence of numbers.

When this option is used, the mean `m` must be convertible to a positive floating-point number at the time when the random generator is generated.

Return Values

procedure.

See Also

MuPAD Functions

`stats::poissonCDF` | `stats::poissonPF` | `stats::poissonQuantile`

stats::quadraticMean

Quadratic mean of a data sample

Syntax

```
stats::quadraticMean(x1, x2, …)
```

```
stats::quadraticMean([x1, x2, …])
```

```
stats::quadraticMean(s, <c>)
```

Description

`stats::quadraticMean`(x_1, x_2, …, x_n) returns the quadratic mean

$$\sqrt{\frac{1}{n}\left(\sum_{i=1}^{n} x_i^2\right)}$$

of the data x_i.

The column index `c` is optional, if the data are given by a `stats::sample` object containing only one non-string column. Cf. “Example 3” on page 30-371.

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

Examples

Example 1

We calculate the quadratic mean of three values:

```
stats::quadraticMean(a, b, c)
```

$$\sqrt{\frac{a^2}{3}+\frac{b^2}{3}+\frac{c^2}{3}}$$

Alternatively, data may be passed as a list:

```
stats::quadraticMean([2, 3, 5])
```

$$\frac{\sqrt{3}\,\sqrt{38}}{3}$$

Example 2

We create a sample:

```
stats::sample([[a1, b1, c1], [a2, b2, c2]])
```

```
a1  b1  c1
a2  b2  c2
```

The quadratic mean of the second column is:

```
stats::quadraticMean(%, 2)
```

$$\sqrt{\frac{b1^2}{2}+\frac{b2^2}{2}}$$

Example 3

We create a sample consisting of one string column and one non-string column:

```
stats::sample([["1996", 1242], ["1997", 1353], ["1998", 1142]])
```

```
"1996"  1242
"1997"  1353
"1998"  1142
```

We compute the quadratic mean of the second column. In this case this column does not have to be specified, since it is the only non-string column:

```
float(stats::quadraticMean(%))
```

```
1248.644198
```

Parameters

x_1, x_2, …

The statistical data: arithmetical expressions.

s

A sample of domain type `stats::sample`.

c

An integer representing a column index of the sample `s`. This column provides the data `x1`, `x2` etc.

Return Values

arithmetical expression.

See Also

MuPAD Functions

`stats::geometricMean` | `stats::harmonicMean` | `stats::mean` | `stats::median` | `stats::modal` | `stats::stdev` | `stats::variance`

stats::reg

Regression (general linear and nonlinear least squares fit)

Syntax

```
stats::reg([x1, 1, …,xk, 1], …,[x1, m, …,xk, m], [y1, …,yk], <[w1, …,wk]>, f, [x1, xm], [p1, …
stats::reg([[x1, 1, …,x1, m, y1, <w1>], …,[xk, 1, …,xk, m, yk, <wk>]], f, [x1, …,xm], [p1, …,p
stats::reg(s, c1, …,cm, cy, <cw>, f, [x1, …,xm], [p1, …,pn], <StartingValues = [p1(0), …,
stats::reg(s, [c1, cm], cy, <cw>, f, [x1, …,xm], [p1, …,pn], <StartingValues = [p1(0), …,
```

Description

Consider a "model function" f with n parameters $p_1, \ldots, p_n$ relating a dependent variable y and m independent variables $x_1, \ldots, x_m$: $y = f(x_1, \ldots, x_m, p_1, \ldots, p_n)$. Given k different measurements $x_{1j}, \ldots, x_{kj}$ for the independent variables x_j and corresponding measurements $y_1, \ldots, y_k$ for the dependent variable y, one fits the parameters $p_1, \ldots, p_n$ by minimizing the "weighted quadratic deviation" ("chi-squared")

$$\chi^2(p_1, \ldots, p_n) = \sum_{i=1}^{k} w_i \left| y_i - f(x_{i1}, \ldots, x_{im}, p_1, \ldots, p_n) \right|^2 .$$

`stats::reg(..data.., f, [x.1, ... , x.m], [p.1, ... , p.n], [w.1, ..., w.n])` computes numerical approximations of the fit parameters $p_1, \ldots, p_n$.

All data must be convertible to real or complex floating-point values via `float`.

The number of measurements k must not be less than the number n of parameters p_i.

The model function f may be non-linear in the independent variables x_i and the fit parameters p_i. E.g., a model function such as `p1 + p2*x1^2 + exp(p3 + p4*x2)` with the independent variables `x1, x2` and the fit parameters `p1, p2, p3, p4` is accepted.

Note that the fitting of model functions with a non-linear dependence on the parameters p_i is much more costly than a linear regression, where the p_i enter linearly. The functional dependence of the model function on the variables x_i is of no relevance.

Note: There are rare cases where the implemented algorithm converges to a local minimum rather than to a global minimum. In particular, this problem may arise when the model involves periodic functions. It is recommended to provide suitable starting values for the fit parameters in this case. Cf. "Example 4" on page 30-376.

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

Environment Interactions

The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Examples

Example 1

We fit a linear function $y = p_1 + p_2\,x_1$ to four data pairs (x_{i1}, y_i) given by two lists:

```
stats::reg([0, 1, 2, 3], [1, 3, 5, 7],
           p1 + p2*x1, [x1], [p1, p2])
```

```
[[1.0, 2.0], 0.0]
```

The parameter values $p_1 = 1.0$, $p_2 = 2.0$ provide a perfect fit: up to numerical round-off, the quadratic deviation vanishes.

Example 2

We fit an exponential function $y = a\,e^{b\,x}$ to five data pairs (x_i, y_i). Weights are used to decrease the influence of the "exceptional pair" $(x, y) = (5.0, 6.5\times10^6)$ on the fit:

```
stats::reg([[1.1, 54, 1], [1.2, 73, 1], [1.3, 98, 1],
            [1.4, 133, 1], [5.0, 6.5*10^6, 10^(-4)]],
           a*exp(b*x), [x], [a, b])
```

$$[[1.992321622, 2.999602426], 0.2001899629]$$

Example 3

We create a sample with four columns. The first column is a counter labeling the measurements. This column is of no further relevance here. The second and third column provide measured data of two variables x_1 and x_2, respectively. The last column provides corresponding measurements of a dependent variable.

```
s := stats::sample([[1, 0, 0,  1.1], [2, 0, 1,  5.4],
                    [3, 1, 1,  8.5], [4, 1, 2, 18.5],
                    [5, 2, 1, 15.0], [6, 2, 2, 24.8]])
```

1	0	0	1.1
2	0	1	5.4
3	1	1	8.5
4	1	2	18.5
5	2	1	15.0
6	2	2	24.8

First, we try to model the data provided by the columns 2, 3, 4 by a function that is linear in the variables x_1, x_2. We specify the data columns by a list of column indices:

```
stats::reg(s, [2, 3, 4], p1 + p2*x1 + p3*x2,
           [x1, x2], [p1, p2, p3])
```

$$[[-0.9568181818, 4.688636364, 7.272727273], 15.23613636]$$

The quadratic deviation is rather large, indicating that a linear function is inappropriate to fit the data. Next, we extend the model and consider a polynomial fit function of degree 2. This is still a linear regression problem, because the fit parameters enter the model function linearly. We specify the data columns by a sequence of column indices:

```
stats::reg(s, 2, 3, 4,
           p1 + p2*x1 + p3*x2 + p4*x1^2 + p5*x2^2,
           [x1, x2], [p1, p2, p3, p4, p5])
```

$$[[1.1, 1.525, 1.5, 1.625, 2.8], 0.01]$$

Finally, we include a further term `p6*x1*x2` in the model, obtaining a perfect fit:

```
stats::reg(s, 2, 3, 4,
           p1 + p2*x1 + p3*x2 + p4*x1^2 + p5*x2^2 + p6*x1*x2,
           [x1, x2], [p1, p2, p3, p4, p5, p6])
```

$$\left[[1.1, 1.6, 1.35, 1.7, 2.95, -0.2], 1.796307589\,10^{-27}\right]$$

```
delete s:
```

Example 4

We create a sample of two columns:

```
s := stats::sample([[1, -1.44], [2, -0.82],
                    [3, 0.97], [4, 1.37]])
```

```
1  -1.44
2  -0.82
3   0.97
4   1.37
```

The data are to be modeled by a function of the form $y = p_1\, sin(p_2\, x)$, where the first column contains measurements of x and the second column contains corresponding data for y. Note that in this example there is no need to specify column indices, because the sample contains only two columns:

```
stats::reg(s, a*sin(b*x), [x], [a, b])
```

$$[[-1.499812823, 1.281963381], 0.00001255632629]$$

Fitting a periodic function may be problematic. We provide starting values for the fit parameters and obtain a quite different set of parameters approximating the data with the same quality:

```
stats::reg(s, a*sin(b*x), [x], [a, b], StartingValues = [2, 5])
```

```
[[1.499812823, 5.001221926], 0.00001255632629]
```

```
delete s:
```

Example 5

The blood sugar level `y` (in mmol/L) of a diabetic is measured over a period of 10 days with 5 measurements per day at `x1` = 7 (o'clock a.m.), `x1` = 12 (noon), `x1` = 15 (afternoon), `x1` = 19 (before dinner), and `x1` = 23 (bed time). These are the measurements:

```
Y:=  //hour: 7    12   15   19   23
    [     [ 7.2, 5.5, 6.8, 5.4, 6.0], // day 1
          [ 6.3, 5.0, 5.5, 5.8, 4.9], // day 2
          [ 6.5, 6.3, 4.8, 4.5, 5.0], // day 3
          [ 4.3, 5.2, 4.3, 4.7, 4.0], // day 4
          [ 7.1, 7.2, 6.7, 7.2, 5.5], // day 5
          [ 5.8, 5.5, 4.9, 5.0, 6.2], // day 6
          [ 6.2, 4.8, 5.0, 5.2, 5.3], // day 7
          [ 4.8, 5.8, 5.7, 6.2, 5.0], // day 8
          [ 5.2, 3.8, 4.8, 5.8, 4.7], // day 9
          [ 5.8, 4.7, 5.0, 6.5, 6.3]  // day 10
    ]:
```

We have a total of 50 measurements. Each measurement is a triple `[x1, x2, y]`, where *x1* is the hour of the day, *x2* is the day number, and `y` is the blood sugar level:

```
data:= [([ 7, x2, Y[x2][1]],
         [12, x2, Y[x2][2]],
         [15, x2, Y[x2][3]],
         [19, x2, Y[x2][4]],
         [23, x2, Y[x2][5]]
        ) $ x2 = 1 .. 10]:
```

We model the blood sugar `y` as a function of the hour of the day `x1` and the day number `x2` (trying to detect a general tendency). We assume a periodic dependence on `x1` with a period of 24 hours:

```
y := y0 + a*x2 + b*sin(2*PI/24*x1 + c):
```

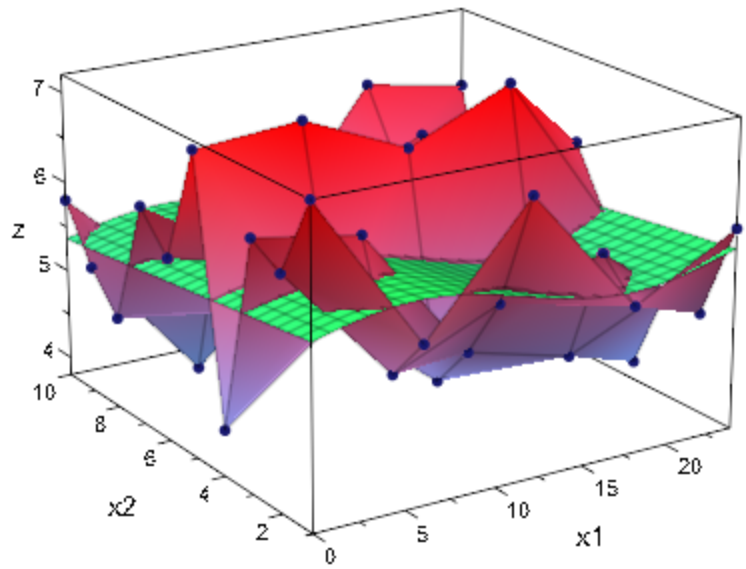
A least squares fit of the given data leads to the following parameters `y0, a, b, c`:

```
[y0abc, residue]:= stats::reg(data, y, [x1, x2], [y0, a, b, c]):
[y0, a, b, c]:= y0abc
```

$$[5.809945498,\ -0.04618181818,\ 0.2055298076,\ 0.07751988162]$$

The average blood sugar level is `y0` = 5.8... with an improvement of `a` = −0.046... per day. The amplitude of the daily variation of `y` is `b` = 0.205.... We visualize the measurements `Y` by a `plot::Matrixplot`. The least squares fit of our model function `y` is added as a function graph:

```
plot(plot::Matrixplot(Y, x1 = 0..24, x2 = 1..10),
     plot::Function3d(y, x1 = 0..24, x2 = 1..10,
                      Color = RGB::Green))
```




```
delete Y, data, y, y0abc, y0, a, b, c, residue:
```

Example 6

We consider a decaying radioactive source, whose activity N ("counts") is measured at intervals of 1 second. The physical model for the decay is $N(t) = \frac{N_0}{e^{\frac{t}{\tau}}}$, where $N(t)$ is

the count rate at time t, N_0 is the base rate at time $t = 0$ and τ is the lifetime of the radioactive source. Instead of taking data from an actual physical experiment, we create artificial data with a base rate $N_0 = 100$ and a lifetime $\tau = 300$:

```
T := [i $ i= 0 .. 100]:
N := [100*exp(-t/300) $ t in T]:
```

By construction, we obtain a perfect fit when estimating the paramaters N_0 and τ of the model:

```
stats::reg(T, N, N0*exp(-t/tau), [t], [N0, tau]);
```

$$\left[[100.0,\ 300.0],\ 1.133987551\ 10^{-30}\right]$$

We perturb the data:

```
N := [stats::poissonRandom(n)() $ n in N]:
```

Since the data n_i in N are Poissonian, their standard deviation is the square root of their mean: $\sigma(n_i) \approx \sqrt{n_i}$. Thus, suitable weights for a least squares estimation of the parameters are given by $w_i = \frac{1}{\sigma(n_i)^2} = \frac{1}{n_i}$:

```
W := [1/n $ n in N]:
```

With these weights, a least squares fit of the model parameters N_0 and τ is computed. The option `CovarianceMatrix` is used to get information on confidence intervals for the parameters:

```
[p, chisquared, C] :=
    stats::reg(T, N, W, N0*exp(-t/tau), [t], [N0, tau],
               CovarianceMatrix)
```

$$\left[[95.43096895,\ 309.3571484],\ 87.4642626,\ \begin{pmatrix} 3.998386217 & -61.38541859 \\ -61.38541859 & 1302.650416 \end{pmatrix}\right]$$

The square roots of the diagonal elements of the covariance matrix provides the statistical standard deviations of the fit parameters:

```
sqrt(float(C[1,1])), sqrt(float(C[2,2]))
```

$$1.999596513,\ 36.0922487$$

Thus, the estimate for the base rate N_0 is $N_0 \pm \sigma(N_0) \approx 95.4 \pm 2.0$, the estimate for the lifetime τ is $\tau \pm \sigma(\tau) \approx 309.4 \pm 36.1$. The correlation matrix of the fit parameters is obtained from the covariance matrix via `stats::correlationMatrix`:

```
stats::correlationMatrix(C)
```

$$\begin{pmatrix} 1.0 & -0.8505677452 \\ -0.8505677452 & 1.0 \end{pmatrix}$$

```
delete T, N, W, p, chisquared, C:
```

Parameters

$x_{1,1}, \ldots, x_{k,m}$

Numerical sample data for the independent variables. The entry $x_{i,j}$ represents the i-th measurement of the independent variable x_j.

$y_1, \ldots, y_k$

Numerical sample data for the dependent variable. The entry y_i represents the i-th measurement of the dependent variable.

$w_1, \ldots, w_k$

Weight factors: positive real numerical values. The entry w_i is used as a weight for the data $x_{i,1}, \ldots, x_{i,m}, y_i$ of the i-th measurement. If no weights are provided, then $w_i = 1$ is used.

f

The model function: an arithmetical expression representing a function of the independent variables $x_1, \ldots, x_m$ and the fit parameters $p_1, \ldots, p_n$. The expression must not contain any symbolic objects apart from $x_1, \ldots, x_m, p_1, \ldots, p_n$.

`x1, …,xm`

The independent variables: identifiers or indexed identifiers.

`p1, …,pn`

The fit parameters: identifiers or indexed identifiers.

`p1(0), …,pn(0)`

The user can assist the internal numerical search by providing numerical starting values $p_i(0)$ for the fit parameters p_i. These should be reasonably close to the optimal fit values. The starting values $p_i(0) = 1.0$ are used if no starting values are provided by the user.

`s`

A sample of domain type `stats::sample` containing the data $x_{i,j}$ for the independent variables, the data y_i for the dependent variable and, optionally, the weights w_i.

`cy`

A positive integer representing a column index of the sample `s`. This column provides the measurements y_i for the dependent variable.

`cw`

A positive integer representing a column index of the sample `s`. This column provides the weight factors w_i.

Options

`StartingValues`

Option, specified as `StartingValues = [p1(0), …,pn(0)]`

Positive integers representing column indices of the sample `s`. Column p_j provides the measurements $x_{i,j}$ for the independent variable x_j.

If the model function depends linearly on the fit parameters p_j (“linear regression”), then the optimized parameters are the solution of a linear system of equations. In this case

there is no need to provide starting values for a numerical search. In fact, initial values provided by the user are ignored.

If the model function depends non-linearly on the fit parameters p_j ("non-linear regression"), then the optimized fitting parameters are the solution of a non-linear optimization problem. There is no guarantee that the internal search for a numerical solution will succeed. It is recommended to assist the internal solver by providing reasonably good estimates for the optimal fit parameters.

`CovarianceMatrix`

Changes the return value from $[[p_1, \ldots, p_n], \chi^2]$ to $[[p_1, \ldots, p_n], \chi^2, C]$, where C is the covariance matrix of the estimators p_i given by $C_{i,i} = \sigma(p_i)^2$ and $C_{i,j} = cov(p_i, p_j)$ for $i \neq j$.

With this option, information on confidence intervals for the least squares estimators p_i are provided. In particular, the return value includes the covariance matrix C of type `Dom::Matrix()`. This matrix provides the variances $C_{ii} = \sigma(p_i)^2$ of the least squares estimators p_i and their covariances $C_{ij} = cov(p_i, p_j)$. The covariance matrix is defined via its inverse

$$\left(\frac{1}{C}\right)_{i,j} = \frac{\frac{\partial}{\partial p_j}\frac{\partial}{\partial p_i}\chi^2}{2},$$

Where

$$\chi^2(p_1, \ldots, p_n) = \sum_{i=1}^{k} w_i \left| y_i - f(x_{i1}, \ldots, x_{im}, p_1, \ldots, p_n) \right|^2 .$$

The covariance matrix of the least squares estimators only has a statistical meaning if the stochastic variances $\sigma(y_i)^2$ of the measurements y_i are known. These variances are to be included in the computation by choosing the weights $w_i = \frac{1}{\sigma(y_i)^2}$. Cf. "Example 6" on page 30-378.

The function `stats::correlationMatrix` serves for converting the covariance matrix to the corresponding correlation matrix. See "Example 6" on page 30-378

Return Values

Without the option `CovarianceMatrix`, a list $[[p_1, \ldots, p_n], \chi^2]$ is returned. It contains the optimized fit parameters p_i minimizing the quadratic deviation. The minimized value of this deviation is given by χ^2, it indicates the quality of the fit.

With the option `CovarianceMatrix`, a list $[[p_1, \ldots, p_n], \chi^2, C]$ is returned. The $n \times n$ matrix `C` is the covariance matrix of the fit parameters.

All returned data are floating-point values. `FAIL` is returned if a least square fit of the data is not possible with the given model function or if the internal numerical search failed.

Algorithms

`stats::reg` uses a Marquardt-Levenberg gradient expansion algorithm. Searching for the minimum of $\chi^2(p_1, \ldots, p_n)$, the algorithm does not simply follow the negative gradient, but the diagonal terms of the curvature matrix are increased by a factor that is optimized in each step of the search.

References

P.R. Bevington and D.K. Robinson, “Data Reduction and Error Analysis for The Physical Sciences”, McGraw-Hill, New York, 1992.

See Also

MuPAD Functions

`stats::correlationMatrix` | `stats::linReg` | `stats::sample`

stats::row

Select and re-arrange rows of a sample

Syntax

```
stats::row(s, r1, <r2, …>)

stats::row(s, r1 .. r2, <r3 .. r4, …>)
```

Description

`stats::row(s, ..)` creates a new sample from selected rows of the sample `s`.

`stats::row` is useful for selecting rows of interest or for re-arranging rows.

The rows of `s` specified by the remaining arguments of `stats::row` are used to build a new sample. The new sample contains the rows of `s` in the order specified by the call to `stats::row`. Rows can be duplicated by specifying the row index more than once.

Examples

Example 1

The following sample represents the “population” of a small town:

```
stats::sample([["1990", 10564], ["1991", 10956],
               ["1992", 11007], ["1993", 11123],
               ["1994", 11400], ["1995", 11645]])
```

```
"1990"  10564
"1991"  10956
"1992"  11007
"1993"  11123
"1994"  11400
"1995"  11645
```

We are only interested in the years 1990, 1991, 1992 and 1995. We create a new sample containing the rows of interest:

```
stats::row(%, 1..3, 6)
```

```
"1990"  10564
"1991"  10956
"1992"  11007
"1995"  11645
```

We reorder the sample:

```
stats::row(%, 4, 3, 2, 1)
```

```
"1995"  11645
"1992"  11007
"1991"  10956
"1990"  10564
```

Parameters

s

A sample of domain type `stats::sample`.

r₁, r₂, …

Positive integers representing row indices of the sample `s`. A range $r_1..r_2$ represents all rows from r_1 through r_2.

Return Values

Sample of domain type `stats::sample`.

See Also

MuPAD Functions

`stats::col` | `stats::concatCol` | `stats::concatRow` | `stats::selectRow`

stats::sample

Domain of statistical samples

Syntax

```
stats::sample([[a1, 1, a1, 2, …], [a2, 1, a2, 2, …], …])
```

```
stats::sample([a1, 1, a2, 1, …])
```

Description

sample represents a collection of statistical data, organized as a matrix. Usually, each row refers to an individual of the population described by the sample. Each column represents an attribute.

`stats::sample( [[a1, 1, …, a1, n], …, [am, 1, …, am, n]] )` creates a sample with m rows and n columns, $a_{i,j}$ being the entry in the i-th row, j-th column.

`stats::sample( [a1, 1, …, am, 1] )` creates a sample with m rows and one column.

Each row $[a_{i,1}, \ldots, a_{i,n}]$ must contain the same number of entries.

Elements of domain type `DOM_COMPLEX`, `DOM_EXPR`, `DOM_FLOAT`, `DOM_IDENT`, `DOM_INT`, or `DOM_RAT` are regarded as "data" and are stored in a sample as on input. All other types of input parameters are converted to strings (`DOM_STRING`).

If one element in a column is a string or is converted to a string, then all elements of that column are converted to strings.

This produces two kinds of columns: data columns and string columns.

Superdomain

`Dom::BaseDomain`

Axioms

`Ax::canonicalRep`

Categories

`Cat::Set`

Examples

Example 1

A sample is created from a list of rows:

```
stats::sample([[5, a], [b, 7.534], [7/4, c+d]])
```

```
  5      a
  b  7.534
7/4  c + d
```

For a sample with only one column one can use a flat list instead of a list of rows:

```
stats::sample([5, 3, 8])
```

```
5
3
8
```

Example 2

The following input creates a small sample with columns for “gender”, “age” and “height”, respectively:

```
stats::sample([["m", 26, 180], ["f", 22, 160],
               ["f", 48, 155], ["m", 30, 172]])
```

```
"m"  26  180
"f"  22  160
"f"  48  155
"m"  30  172
```

Note that all entries in a column are automatically converted to strings, if one entry of that column is a string:

```
stats::sample([[m, 26, 180], [f, 22, 160],
               ["f", 48, 155], [m, 30, 172]])
```

```
"m"  26  180
"f"  22  160
"f"  48  155
"m"  30  172
```

Example 3

The functions `float`, `has`, `map`, `nops`, `op`, and `subsop` are overloaded to work on samples as on lists of lists:

```
s := stats::sample([[a, 1], [b, 2], [c, 3]])
```

```
a  1
b  2
c  3
```

```
float(s), has(s, a), map(s, list -> [list[1], list[2]^2]),
nops(s), subsop(s, 1 = [d, 4]), op(s, [1, 2])
```

```
a  1.0          a  1       d  4
b  2.0  , TRUE, b  4  , 3, b  2  , 1
c  3.0          c  9       c  3
```

Indexing works like on `array`s:

```
s[1, 2] := x : s
```

```
a  x
b  2
c  3
```

```
delete s:
```

Example 4

The dot operator may be used to concatenate samples and lists (regarded a samples with one row):

```
s := stats::sample([[1, a], [2, b]]): s.[X, Y].s
```

```
1  a
2  b
X  Y
1  a
2  b
```

```
delete s:
```

Parameters

$a_{1,1}$, $a_{1,2}$, …

Arithmetical expressions or strings.

Methods

Mathematical Methods

`equal` — Test for equality

```
equal(s1, s2)
```

Conversion Methods

`convert` — Convert a list to a sample

```
convert(x)
```

convert_to — Convert a sample to a list of lists

```
convert_to(s, T)
```

expr — Convert a sample to a list of lists of expressions

```
expr(s)
```

Access Methods

size — Return the number of rows

```
size(s)
```

col2list — Return a particular column as a list

```
col2list(s, c, …)
```

append — Append a row

```
append(s, row)
```

_concat — Create a sample from the rows of several samples

```
_concat(s, s1, …)
```

delCol — Delete one or more columns

```
delCol(s, c)
```

delRow — Delete one or more rows

```
delRow(s, r)
```

float — Map the float function to all entries

```
float(s)
```

has — Test for the occurrence of elements

```
has(s, e)
```

If `e` is a list or a set, then this method tests, whether at least one of its elements is among the entries of `s`.

`_index` — Return a particular entry

```
_index(s, i, j)
```

Indexed calls such as `s[i, j]` call this method.

`set_index` — Assign a new value to an entry

```
set_index(s, i, j, x)
```

This method is called by indexed assignments of the form `s[i, j] := x`.

`map` — Map a function to the rows

```
map(s, f)
```

`nops` — Number of rows

```
nops(s)
```

`op` — Get the operands (rows)

```
op(s, i)
```

```
op(s, [i, j])
```

`subsop` — Replace a row

```
subsop(s, i = newrow, …)
```

`row2list` — Return a particular row as a list

```
row2list(s, r, …)
```

Technical Methods

`print` — Output

```
print(s)
```

`fastprint` — Fast output

```
fastprint(s)
```

stats::sample2list

Convert a sample to a list of lists

Syntax

```
stats::sample2list(s)
```

Description

`stats::sample2list(s)` converts the sample `s` to a list of lists.

The sub-lists of the list returned by `stats::sample2list(s)` are the rows of the sample `s`.

Examples

Example 1

First we create a sample from a list of lists:

```
stats::sample([[123, s, 1/2], [442, s, -1/2], [322, p, -1/2]])
```

```
123  s   1/2
442  s  -1/2
322  p  -1/2
```

The input list may be recovered by `stats::sample2list`:

```
stats::sample2list(%)
```

$$\left[\left[123,\ s,\ \frac{1}{2}\right],\ \left[442,\ s,\ -\frac{1}{2}\right],\ \left[322,\ p,\ -\frac{1}{2}\right]\right]$$

Parameters

s

A sample of domain type `stats::sample`.

Return Values

List of lists.

See Also

MuPAD Functions
`stats::unzipCol` | `stats::zipCol`

stats::selectRow

Select rows of a sample

Syntax

```
stats::selectRow(s, c, x, <Not>)
```

```
stats::selectRow(s, [c1, c2, …], [x1, x2, …], <Not>)
```

Description

`stats::selectRow(s, ...)` selects rows of the sample `s` having specific entries in specific places.

`stats::selectRow(s, c, x)` returns a sample consisting of all rows in `s`, which contain the data element `x` at the position `c`.

`stats::selectRow(s, [c1, c2, …], [x1, x2, …] )` returns a sample consisting of all rows in `s`, which contain the data element x_1 at the position c_1 *and* x_2 at the position c_2 etc. There must be as many positions c_1, c_2, … as data elements x_1, x_2, …

Examples

Example 1

We create a sample with two columns:

```
stats::sample([[a, 5], [c, 1], [a, 2], [b, 3]])
```

```
a  5
c  1
a  2
b  3
```

We select all rows with `a` as their first entry:

```
stats::selectRow(%, 1, a)
```

```
a  5
a  2
```

Example 2

We create a sample containing income and costs in the years 1997 and 1998:

```
stats::sample([[123, "costs", "97"], [442, "income", "98"],
               [11, "costs", "98"], [623, "income", "97"]])
```

```
123  "costs"   "97"
442  "income"  "98"
 11  "costs"   "98"
623  "income"  "97"
```

We select the row which has `"income"` in the second and `"97"` in the third column:

```
stats::selectRow(%, [2, 3], ["income", "97"])
```

```
623  "income"  "97"
```

We select the remaining rows:

```
stats::selectRow(%2, [2, 3], ["income", "97"], Not)
```

```
123  "costs"   "97"
442  "income"  "98"
 11  "costs"   "98"
```

Parameters

`s`

A sample of domain type `stats::sample`.

`c`, c_1, c_2, …

Integers representing column indices of the sample `s`.

`x, x1, x2, …`

Arithmetical expressions.

Options

`Not`

Causes `stats::selectRow` to select those rows which do *not* have the specified entries.

Return Values

Sample of domain type `stats::sample`.

See Also

MuPAD Functions
`stats::row`

stats::sortSample

Sort the rows of a sample

Syntax

```
stats::sortSample(s)
```

```
stats::sortSample(s, c1, c2, …)
```

```
stats::sortSample(s, [c1, c2, …])
```

Description

`stats::sortSample(s, ...)` sorts the rows of the sample `s`.

The sorting of rows only uses the entries of the specified columns. First, rows are sorted according to the elements of the first specified column. Those rows with identical elements in the first specified column are then ordered according to the elements in the second specified column etc.

If no columns are specified, then column 1 is used for sorting. In case of a tie, column 2 is used etc.

Numbers are sorted numerically, strings are sorted lexicographically. Identifiers are sorted according to the strategy used by the MuPAD `sort` command. Numbers come first, identifiers second.

Examples

Example 1

We create a sample with one column and sort it:

```
stats::sortSample(stats::sample([x, g2, 3, g1, 8/5, 2]))
```

```
8/5
  2
  3
 g1
 g2
  x
```

Example 2

We create a sample with two columns:

```
stats::sample([[b, 2], [a, 5], [a, 2], [c, 1], [b, 3]])
```

```
b  2
a  5
a  2
c  1
b  3
```

Note the different sorting priorities specified by the column indices:

```
stats::sortSample(%, 1), stats::sortSample(%, 2),
stats::sortSample(%, 1, 2), stats::sortSample(%, 2, 1)
```

```
a  2    c  1    a  2    c  1
a  5    a  2    a  5    a  2
b  3  , b  2  , b  2  , b  2
b  2    b  3    b  3    b  3
c  1    a  5    c  1    a  5
```

Example 3

We create a sample containing income and costs in the years 1997 and 1998:

```
stats::sample([[123, "costs", "97"], [720, "income", "98"],
               [623, "income", "97"], [150, "costs", "98"]])
```

```
123  "costs"   "97"
720  "income"  "98"
623  "income"  "97"
150  "costs"   "98"
```

We sort according to the year (third column):

```
stats::sortSample(%, 3)
```

```
623  "income"  "97"
123  "costs"   "97"
150  "costs"   "98"
720  "income"  "98"
```

We sort with priority on the year. Items of the same year are then sorted lexicographically (“costs” before “income”):

```
stats::sortSample(%2, 3, 2)
```

```
123  "costs"   "97"
623  "income"  "97"
150  "costs"   "98"
720  "income"  "98"
```

Parameters

s

A sample of domain type `stats::sample`.

c_1, c_2, …

Integers representing column indices of the sample `s`.

Return Values

Sample of domain type `stats::sample`.

See Also

MuPAD Functions

`stats::selectRow`

stats::stdev

Standard deviation of a data sample

Syntax

```
stats::stdev(x1, x2, …, <Sample | Population>)
```

```
stats::stdev([x1, x2, …], <Sample | Population>)
```

```
stats::stdev(s, <c>, <Sample | Population>)
```

Description

`stats::stdev( x1, x2, ..., xn)` returns the standard deviation

$$\sqrt{\frac{1}{n-1}\left(\sum_{i=1}^{n}(x_i-\overline{x})^2\right)},$$

where $\overline{x}$ is the arithmetic mean of the data x_i.

`stats::stdev( x1, x2, ..., xn, Population)` returns

$$\sqrt{\frac{1}{n}\left(\sum_{i=1}^{n}(x_i-\overline{x})^2\right)}.$$

The standard deviation is the square root of the variance.

The column index `c` is optional, if the data are given by a `stats::sample` object containing only one non-string column. Cf. “Example 3” on page 30-402.

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

Examples

Example 1

We calculate the standard deviation of three values:

```
stats::stdev(2, 3, 5)
```

$$\frac{\sqrt{3}\,\sqrt{7}}{3}$$

Alternatively, the data may be passed as a list:

```
stats::stdev([2, 3, 5])
```

$$\frac{\sqrt{3}\,\sqrt{7}}{3}$$

Example 2

We create a sample:

```
stats::sample([[a1, b1, c1], [a2, b2, c2]])
```

```
a1  b1  c1
a2  b2  c2
```

The standard deviation of the second column is:

```
expand(stats::stdev(%, 2))
```

$$\sqrt{2}\,\sqrt{\frac{b1^2}{4} - \frac{b1\,b2}{2} + \frac{b2^2}{4}}$$

Example 3

We create a sample consisting of one string column and one non-string column:

```
stats::sample([["1996", 1242], ["1997", 1353], ["1998", 1142]])
```

```
"1996"   1242
"1997"   1353
"1998"   1142
```

We compute the standard deviation of the second column. In this case this column does not have to be specified, since it is the only non-string column:

```
float(stats::stdev(%))
```

```
     105.5477775
```

We repeat the computation with the option `Population`:

```
float(stats::stdev(%2, Population))
```

```
     86.17939945
```

Parameters

x_1, x_2, …

The statistical data: arithmetical expressions

s

A sample of domain type `stats::sample`

c

An integer representing a column index of the sample `s`. This column provides the data x_1, x_2 etc.

Options

Population, Sample

With `Sample`, the data are regarded as a “sample”, not as a full population. The default is `Sample`.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`stats::geometricMean` | `stats::harmonicMean` | `stats::mean` | `stats::median` | `stats::modal` | `stats::quadraticMean` | `stats::variance`

stats::swGOFT

The Shapiro-Wilk goodness-of-fit test for normality

Syntax

```
stats::swGOFT(x1, x2, …)
```

```
stats::swGOFT([x1, x2, …])
```

```
stats::swGOFT(s, <c>)
```

Description

`stats::swGOFT`([x_1, x_2, …]) applies the Shapiro-Wilk goodness-of-fit test for the null hypothesis: "the data x_1, x_2, … are normally distributed (with unknown mean and variance)". The sample size must not be larger than 5000 and not smaller than 3.

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

An error is raised by `stats::swGOFT` if any of the data cannot be converted to a real floating-point number or if the sample size is too large or too small.

Let y_1, …, y_n be the input data x_1, …, x_n arranged in ascending order. `stats::swGOFT` returns the list `[PValue = p, StatValue = w]` containing the following information:

- `w` is the attained value of the Shapiro-Wilk statistic

$$W = \frac{\left(\sum_{i=1}^{n} a_i\, y_i\right)^2}{n\, S^2} = \frac{\left(\sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} a_{n+1-i}\left(y_{n+1-i} - y_i\right)\right)^2}{n\, S^2}.$$

  Here, the a_i are the Shapiro-Wilk coefficients, and `S^2` is the statistical variance of the sample.

- `p` is the observed significance level of the Shapiro-Wilk statistic W.

The observed significance level `PValue = p` returned by `stats::swGOFT` has to be interpreted in the following way: If `p` is smaller than a given significance level α<<1, the null hypothesis may be rejected at level α. If `p` is larger than α, the null hypothesis should not be rejected at level α.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We test a list of random data that purport to be a sample of normally distributed numbers:

```
f := stats::normalRandom(0, 1, Seed = 123):
data := [f() $ i = 1..400]:
stats::swGOFT(data)
```

```
[PValue = 0.4902802259, StatValue = 0.9963301659]
```

The observed significance level 0.490... is not small. Consequently, one should not reject the null hypothesis that the data are normally distributed.

Next, we dote the data with some uniformly continuous deviates:

```
impuredata := data . [frandom() $ i = 1..101]:
stats::swGOFT(impuredata)
```

```
[PValue = 0.00001713272044, StatValue = 0.9833725964]
```

The doted data may be rejected as a sample of normal deviates at significance levels as small as 0.000017....

```
delete f, data, impuredata:
```

Example 2

We create a sample consisting of one string column and two non-string columns:

```
s := stats::sample(
   [["1996", 1242, PI - 1/2],
    ["1997", 1353, PI + 0.3],
    ["1998", 1142, PI + 0.5],
    ["1999", 1201, PI - 1],
    ["2001", 1201, PI]
   ])
```

```
"1996"  1242  PI - 1/2
"1997"  1353  PI + 0.3
"1998"  1142  PI + 0.5
"1999"  1201    PI - 1
"2001"  1201        PI
```

We check whether the data of the third column are normally distributed:

```
stats::swGOFT(s, 3)
```

```
[PValue = 0.7314967372, StatValue = 0.9492034961]
```

The observed significance level returned by the test is not small: the test does not indicate that the data are not normally distributed.

```
delete s:
```

Parameters

x_1, x_2, …

The statistical data: real numerical values

s

A sample of domain type `stats::sample`

c

An integer representing a column index of the sample `s`. This column provides the data x_1, x_2 etc. There is no need to specify a column number `c` if the sample has only one column.

Return Values

List of two equations `[PValue = p, StatValue = w]` with floating-point values `p` and `w`. See the `Details' section below for the interpretation of these values.

Algorithms

The implemented algorithm for the computation of the Shapiro-Wilk coefficients, the Shapiro-Wilk statistic and the observed significance level is based on: Patrick Royston, “Algorithm AS R94”, Applied Statistics, Vol.44, No.4 (1995).

Following Royston, the Shapiro-Wilk coefficients a_i are computed by an approximation of

$$a = \frac{M}{V\left(\left(\frac{M^T}{V}\right)\left(\frac{M}{V}\right)\right)}$$

where M denotes the expected values of standard normal order statistic for a sample, V is the corresponding covariance matrix, and M^T is the transpose of M.

See Also

MuPAD Functions

`stats::csGOFT` | `stats::ksGOFT` | `stats::tTest`

stats::tabulate

Statistics of duplicate rows in a sample

Syntax

```
stats::tabulate(s)
stats::tabulate(s, c1, c2, …, <f>)
stats::tabulate(s, c1 .. c2, c3 .. c4, …, <f>)
stats::tabulate(s, [c1, f1], [c2, f2], …)
stats::tabulate(s, [c1, c2, …, f1], [c3, c4, …, f2], …)
```

Description

`stats::tabulate(s)` eliminates duplicate rows in the sample `s` and appends a column containing the multiplicities.

`stats::tabulate( s, c1, c2, …, f )` combines all rows that are identical except for entries in the specified columns c_1, c_2 etc. The function `f` is applied to these columns, its result replaces the values in these columns.

`stats::tabulate( s, [c1, f1], [c2, f2], … )` combines all rows that are identical except for entries in the columns c_1, c_2 etc. The functions f_1, f_2 etc. are applied to these columns, the results replace the values in these columns.

`stats::tabulate` regards rows as duplicates if they have identical entries in the columns that are *not* specified.

With `stats::tabulate( s, c1, c2, …, f )` the function `f` is applied to the entries of the duplicate rows in the specified columns. Duplicates are eliminated and replaced by a single instance of the row, the result of `f` is inserted into the corresponding columns.

The function `f` must accept as many parameters as there are duplicates. Typical applications involve functions such as `stats::mean` which accept arbitrarily many arguments.

E.g., with `stats::mean` duplicate rows are replaced by a single row, in which the entries of the columns c_1, c_2 etc. are replaced by the mean values of the corresponding entries of the duplicates.

If no function `f` is specified, then the default function `_plus` is used.

If column indices are specified more than once, extra columns with the result of the specified function are inserted into the sample.

Consecutive columns may be specified by ranges. E.g., the call `stats::tabulate(s, c[1]..c[2], dots, f)` is a short hand notation for `stats::tabulate(s, c1, c1 + 1, …, c2, …, f)`.

With `stats::tabulate(s, [c1, f1], [c2, f2], …)` pairs of columns and corresponding procedures are specified. Again, rows are regarded as duplicates if they have identical entries in the columns that are *not* specified. Duplicates are eliminated and replaced by a single instance of the row, the result of f_1 is inserted in column c_1, the result of f_2 is inserted in column c_2 etc.

If column indices are specified more than once, then extra columns with the result of the specified functions are inserted into the sample.

With `stats::tabulate(s, [c1, c2, …, f1], …)` it is possible to apply functions that act on several columns. The procedure f_1 has to accept a sequence of lists (each representing a column). The specified columns are replaced by a single column containing the result of f_1. If column indices are specified more than once, then extra columns with the result of the specified function(s) are inserted into the sample. See “Example 2” on page 30-411 and “Example 3” on page 30-412.

Examples

Example 1

We create a sample:

```
s := stats::sample([[a, A, 1], [a, A, 1], [a, A, 2],
                    [b, B, 5], [b, B, 10]])
```

```
a  A   1
a  A   1
a  A   2
b  B   5
b  B  10
```

Duplicate rows of the sample are counted. There are four unique rows, one occurring twice:

```
stats::tabulate(s)
```

```
a  A   1  2
a  A   2  1
b  B   5  1
b  B  10  1
```

In the following call, rows are regarded as duplicates if the entries in the first two columns coincide. We compute the mean value of the third entry of the duplicates:

```
stats::tabulate(s, 3, stats::mean)
```

```
a  A   4/3
b  B  15/2
```

We compute both the mean and the standard deviation of the data in the third column for the sub-samples labeled `a A' and `b B' by the first two columns:

```
stats::tabulate(s, [3, stats::mean], [3, stats::stdev])
```

```
a  A   4/3      3^(1/2)/3
b  B  15/2  (5*2^(1/2))/2
```

```
delete s:
```

Example 2

We create a sample containing columns for “gender”, “age” and “size”:

```
s := stats::sample([["f", 25, 166], ["m", 30, 180],
                    ["f", 54, 160], ["m", 40, 170],
                    ["f", 34, 170], ["m", 20, 172]])
```

```
"f"  25  166
"m"  30  180
"f"  54  160
"m"  40  170
"f"  34  170
"m"  20  172
```

We use `stats::mean` on the second and third column to calculate the average “age” and “size” of each gender:

```
stats::tabulate(s, 2..3, float@stats::mean)
```

```
"f"  37.66666667  165.3333333
"m"         30.0        174.0
```

With the next call both the mean and the standard deviation of “age” and “size” for each gender are inserted into the sample.

```
stats::tabulate(s,
  [2, float@stats::mean], [2, float@stats::stdev],
  [3, float@stats::mean], [3, float@stats::stdev])
```

```
"f"  37.66666667  14.84362939  165.3333333  5.033222957
"m"         30.0         10.0        174.0  5.291502622
```

We compute the Bravais-Pearson correlation coefficient between “age” and “size” for each gender:

```
stats::tabulate(s, [2, 3, float@stats::correlation])
```

```
"f"  -0.7540135991
"m"  -0.1889822365
```

```
delete s:
```

Example 3

We create a sample:

```
s := stats::sample([[a, x1, 1, 2], [b, x2, 2, 4],
```

```
                    [b, x1, 2, 4], [e, x2, 3, 5.5]])
```

```
a  x1  1    2
b  x2  2    4
b  x1  2    4
e  x2  3  5.5
```

We regard rows with the same entry in the second column as “of the same kind”. We tabulate the sample using different functions on the remaining columns:

```
stats::tabulate(s, [1, _plus], [3, _mult], [4, stats::mean])
```

```
a + b  x1  2     3
b + e  x2  6  4.75
```

One can apply customized procedures. In the following we define the procedure `plusmult`, which sums up the elements of two lists (representing columns) and then multiplies the sums.

```
plusmult := proc(x, y) begin _plus(op(x))*_plus(op(y)) end_proc:
```

This procedure is then used to combine the first and the third column. Simultaneously, the mean and the standard deviation of the fourth column is inserted into the sample.

```
stats::tabulate(s, [1, 3, plusmult], [4, stats::mean],
                [4, stats::stdev])
```

```
3*a + 3*b  x1     3      2^(1/2)
5*b + 5*e  x2  4.75  1.060660172
```

```
delete plusmult, s:
```

Parameters

s

A sample of domain type `stats::sample`

c_1, c_2, …

Integers representing column indices of the sample `s`

`f, f1, f2, …`

Procedures

Return Values

Sample of domain type `stats::sample`.

See Also

MuPAD Functions

`stats::calc`

stats::tCDF

Cumulative distribution function of Student's t-distribution

Syntax

```
stats::tCDF(a)
```

Description

`stats::tCDF(a)` returns a procedure representing the cumulative distribution function

$$x \to \frac{\Gamma\left(\frac{a+1}{2}\right)}{\Gamma\left(\frac{a}{2}\right)\sqrt{a\,\pi}} \int_{-\infty}^{x} \frac{1}{\left(1+\frac{t^2}{a}\right)^{\frac{a+1}{2}}}\,dt$$

of Student's t-distribution with shape parameter ('degrees of freedom') `a>0`.

The procedure `f := stats::tCDF(a)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

If `a` can be converted to a positive floating point number `x` is a real number, the return value `f(x)` is a floating-point number.

`f(infinity)` produces `0.0`; `f(-infinity)` produces `1.0`.

In all other cases, `f(x)` returns the symbolic call `stats::tCDF(a)(x)`.

The procedure `f := stats::tCDF(a)` can also be called in the form `f(x, Symbolic)` with arithmetical expressions `x`.

If `a` is a positive integer, explicit symbolic expressions in `x` are returned. Otherwise, the function behaves as if called without the option `Symbolic`. Cf. “Example 3” on page 30-417.

Numerical values of `a` are only accepted if they are positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the cumulative distribution function with a = $\frac{3}{4}$ at various points:

```
f := stats::tCDF(3/4):
f(-infinity), f(-4), f(0), f(1/2), f(0.75), f(PI), f(infinity)
```

$$0.0,\ 0.1085580939,\ 0.5,\ 0.6374082038,\ 0.689341961,\ 0.8707365121,\ 1.0$$

Nonpositive numerical values of the shape parameter lead to an error:

```
stats::tCDF(-1)(0.75)
```

```
Error: The shape parameter must be positive. [stats::tCDF]
```

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::tCDF(a): f(x), f(1/3), f(0.4)
```

$$\text{stats::tCDF}(a)(x),\ \text{stats::tCDF}(a)\left(\frac{1}{3}\right),\ \text{stats::tCDF}(a)(0.4)$$

When a positive real number is assigned to a, the call $f(x)$ returns a floating-point number if x is numerical:

```
a := sqrt(10): f(PI)
```

$$0.9759846335$$

```
delete f, a:
```

Example 3

We demonstrate the option `Symbolic`. Without this option, the CDF function only produces explicit results if both `a` and `x` are numerical values:

```
stats::tCDF(3)(x)
```

$$\text{stats::tCDF}(3)(x)$$

If the shape parameter is a positive integer, an explicit symbolic representation of the t-distribution exists for any x:

```
f := stats::tCDF(3): f(x, Symbolic)
```

$$\frac{\arctan\left(\frac{\sqrt{3}\,x}{3}\right)}{\pi} + \frac{\sqrt{3}\,x}{3\left(\frac{\pi x^2}{3} + \pi\right)} + \frac{1}{2}$$

No internal floating-point conversions occur even if all input parameters are exact numerical values:

```
f(sqrt(2), Symbolic) = f(sqrt(2))
```

$$\frac{\arctan\left(\frac{\sqrt{2}\,\sqrt{3}}{3}\right)}{\pi} + \frac{\sqrt{2}\,\sqrt{3}}{5\,\pi} + \frac{1}{2} = 0.8738922518$$

If the shape parameter is not a positive integer, the option `Symbolic` has no effect. The function f behaves as if called without this option:

```
f := stats::tCDF(PI):
f(sqrt(2), Symbolic) = f(sqrt(2))
```

$$0.8758345238 = 0.8758345238$$

```
f := stats::tCDF(PI):
f(x, Symbolic)
```

$$\text{stats::tCDF}(\pi)(x, \text{Symbolic})$$

```
delete f:
```

Parameters

a

The shape parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`stats::tPDF` | `stats::tQuantile` | `stats::tRandom`

stats::tPDF

Probability density function of Student's t-distribution

Syntax

```
stats::tPDF(a)
```

Description

`stats::tPDF(a)` returns a procedure representing the probability density function

$$x \to \frac{\frac{\Gamma\left(\frac{a+1}{2}\right)}{\Gamma\left(\frac{a}{2}\right)\sqrt{a\,\pi}}}{\left(\frac{x^2}{a}+1\right)^{\frac{a}{2}+\frac{1}{2}}}$$

of Student's t-distribution with shape parameter ('degrees of freedom') `a > 0`.

The procedure `f := stats::tPDF(a)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

If `x` is a floating-point number and `a` can be converted to a floating-point number, then `f(x)` returns a floating-point number.

`f(infinity)` and `f(-infinity)` produce the result `0.0`.

In all other cases, the expression $\frac{\frac{\Gamma\left(\frac{a+1}{2}\right)}{\Gamma\left(\frac{a}{2}\right)\sqrt{a}\,\sqrt{\pi}}}{\left(\frac{x^2}{a}+1\right)^{\frac{a}{2}+\frac{1}{2}}}$ is returned.

If floating-point results are desired, call `f(x)` with a floating-point value `x`.

Numerical values for a are only accepted if they are real and positive.

Examples

Example 1

We evaluate the probability density function with $a = \frac{3}{4}$ at various points:

```
f := stats::tPDF(3/4):
f(-infinity), f(-PI), f(1/2), f(0.5), f(3), f(infinity)
```

$$0.0,\ \frac{2\sqrt{3}\,\Gamma\left(\frac{7}{8}\right)}{3\sqrt{\pi}\,\Gamma\left(\frac{3}{8}\right)\left(\frac{4\pi^2}{3}+1\right)^{7/8}},\ \frac{3^{3/8}\,4^{1/8}\,\Gamma\left(\frac{7}{8}\right)}{2\sqrt{\pi}\,\Gamma\left(\frac{3}{8}\right)},\ 0.232826673,\ \frac{2\sqrt{3}\,13^{1/8}\,\Gamma\left(\frac{7}{8}\right)}{39\sqrt{\pi}\,\Gamma\left(\frac{3}{8}\right)},\ 0.0$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::tPDF(a): f(x), f(0.3)
```

$$\frac{\Gamma\left(\frac{a}{2}+\frac{1}{2}\right)}{\sqrt{\pi}\,\sqrt{a}\,\Gamma\left(\frac{a}{2}\right)\left(\frac{x^2}{a}+1\right)^{\frac{a}{2}+\frac{1}{2}}},\ \frac{0.5641895835\,\Gamma\left(\frac{a}{2}+\frac{1}{2}\right)}{\sqrt{a}\,\Gamma\left(\frac{a}{2}\right)\left(\frac{0.09}{a}+1\right)^{\frac{a}{2}+\frac{1}{2}}}$$

When numerical values are assigned to a, the function f starts to produce floating-point values for floating-point arguments:

```
a := PI: f(0.3)
```

$$0.347916859$$

```
delete f, a:
```

Parameters

a

The shape parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`stats::tCDF` | `stats::tQuantile` | `stats::tRandom`

stats::tQuantile

Quantile function of Student's t-distribution

Syntax

```
stats::tQuantile(a)
```

Description

`stats::tQuantile(a)` returns a procedure representing the quantile function (inverse) of the cumulative distribution function `stats::tCDF(a)`. For $0 \le x \le 1$, the solution of *stats*::*tCDF*(*a*)(*y*) = *x* is given by *y* = *stats*::*tQuantile*(*a*)(*x*).

The procedure `f := stats::tQuantile(a)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number, ±`infinity`, or a symbolic expression:

If `x` is a real number between `0` and `1` and the shape parameter `a` can be converted to a positive real floating-point number, then `f(x)` returns a real floating-point number approximating the solution `y` of `stats::tCDF(a)(y) = x`.

The calls `f(1/2)` and `f(0.5)` produce `0.0` for all values of `a`.

The calls `f(0)` and `f(0.0)` produce `-infinity` for all values of `a`.

The calls `f(1)` and `f(1.0)` produce `infinity` for all values of `a`.

In all other cases, `f(x)` returns the symbolic call `stats::tQuantile(a)(x)`.

Numerical values of *x* are only accepted if $0 \le x \le 1$.

Numerical values of the shape parameter `a` are only accepted if they are real and positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the quantile function with `a` = π at various points:

```
f := stats::tQuantile(PI):
f(0), f(1/10), f(0.5), f(1 - 10^(-10)), f(1)
```

$$-\infty,\ -1.618021174,\ 0.0,\ 1639.390923,\ \infty$$

The value `f(x)` satisfies `stats::tCDF(PI)(f(x)) = x`:

```
stats::tCDF(PI)(f(0.987654))
```

$$0.987654$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::tQuantile(a): f(x), f(9/10)
```

$$\text{stats::tQuantile}(a)(x),\ \text{stats::tQuantile}(a)\left(\frac{9}{10}\right)$$

When a positive real value is assigned to the shape parameter `a`, the function `f` starts to produce floating-point values:

```
a := 17: f(0.999), f(1 - sqrt(2)/10^5)
```

$$3.64576738,\ 5.65913443$$

Numerical values for x are only accepted if $0 \le x \le 1$:

```
f(0.5)
```

```
    0.0
```

```
f(2)
```

```
Error: An argument x with 0 <= x <= 1 is expected. [f]
```

```
delete f, a:
```

Parameters

a

The shape parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`stats::tCDF` | `stats::tPDF` | `stats::tRandom`

stats::tRandom

Generate a random number generator for Student deviates (t-deviates)

Syntax

```
stats::tRandom(a, <Seed = s>)
```

Description

`stats::tRandom(a)` returns a procedure that produces t-deviates (random numbers) with shape parameter ('degrees of freedom') `a>0`.

The procedure `f := stats::tRandom(a)` can be called in the form `f()`. The return value of `f()` is either a floating-point number or a symbolic expression:

If `a` can be converted to a positive floating point number, then `f()` returns a real floating point number.

In all other cases, `stats::tRandom(a)()` is returned symbolically.

Numerical values of `a` are only accepted if they are real and positive.

The values `X = f()` are distributed randomly according to the cumulative distribution function of the t-distribution with shape parameter `a`. For any real x, the probability that $X \leq x$ is given by

$$\frac{\Gamma\left(\frac{a+1}{2}\right)}{\Gamma\left(\frac{a}{2}\right)\sqrt{a\,\pi}} \int_{-\infty}^{x} \frac{1}{\left(1+\frac{t^2}{a}\right)^{\frac{a+1}{2}}}\,\mathrm{d}t$$

.

Without the option `Seed = s`, an initial seed is chosen internally. This initial seed is set to a default value when MuPAD is started. Thus, each time MuPAD is started or re-initialized with the `reset` function, random generators produce the same sequences of numbers.

Note: In contrast to the function `random`, the generators produced by `stats::tRandom` do not react to the environment variable `SEED`.

For efficiency, it is recommended to produce sequences of K random numbers via

```
f := stats::tRandom(a): f() $k = 1..K;
```

rather than by

```
stats::tRandom(a)() $k = 1..K;
```

The latter call produces a sequence of generators each of which is called once. Also note that

```
stats::tRandom(a, Seed = n)() $k = 1..K;
```

does not produce a random sequence, because a sequence of freshly initialized generators would be created each of them producing the same number.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We generate t-deviates with shape parameter `a` = 23:

```
f := stats::tRandom(23): f() $ k = 1..4
```

$$-0.4417960021,\ 0.8951424894,\ 1.291568047,\ -0.8938637441$$

```
delete f:
```

Example 2

With symbolic parameters, no random floating-point numbers can be produced:

```
f := stats::tRandom(a): f()
```

$$\text{stats::tRandom}(a)()$$

When the shape parameter `a` evaluates to a positive real number, `f` starts to produce random floating-point numbers:

```
a := sqrt(99): f() $ k = 1..4
```

$$0.2115095721,\ 0.3326980658,\ 0.6483250281,\ 4.341741815$$

```
delete f, a:
```

Example 3

We use the option `Seed = s` to reproduce a sequence of random numbers:

```
f := stats::tRandom(PI, Seed = 1): f() $ k = 1..4
```

$$0.006866430292,\ -1.483500533,\ -0.6688788771,\ 0.4017860587$$

```
g := stats::tRandom(PI, Seed = 1): g() $ k = 1..4
```

$$0.006866430292,\ -1.483500533,\ -0.6688788771,\ 0.4017860587$$

```
f() = g(), f() = g()
```

$$0.03860593782 = 0.03860593782,\ -1.259521158 = -1.259521158$$

```
delete f, g:
```

Parameters

a

The shape parameter: an arithmetical expression representing a positive real value

Options

Seed

Option, specified as `Seed = s`

Initializes the random generator with the integer seed `s`. s can also be the option `CurrentTime`, to make the seed depend on the current time.

This option serves for generating generators that return predictable sequences of pseudo-random numbers. The generator is initialized with the seed `s` which may be an arbitrary integer. Several generators with the same initial seed produce the same sequence of numbers.

When this option is used, the shape parameter `a` must be convertible to a positive floating-point number at the time when the random generator is generated.

Return Values

procedure.

Algorithms

The implemented algorithm for the computation of the t-deviates uses a gamma deviate `X` with shape parameter `a/2` and a normal deviate `Y` to arrive at a t-deviate `X/sqrt(Y/a)`. For more information see: D. Knuth, Seminumerical Algorithms (1998), Vol. 2, p. 135.

See Also

MuPAD Functions

`stats::tCDF` | `stats::tPDF` | `stats::tQuantile`

stats::tTest

T-test for a mean

Syntax

`stats::tTest(`x_1`, `x_2`, …, m, <Normal>)`

`stats::tTest([`x_1`, `x_2`, …], m, <Normal>)`

`stats::tTest(s, <c>, m, <Normal>)`

Description

`stats::tTest( [`x_1`, `x_2`, …], m )` tests the null hypothesis: "the true mean of the data x_i is larger than `m`".

`stats::tTest` accepts numerical data as well as symbolic data.

If all data are real floating-point numbers, the returned values `p` and `t` are floating-point numbers.

If `m` is a floating-point number, the sample data are converted to floating-point numbers automatically.

For a sample x_1, x_2, … of size n, `stats::tTest` computes $t = \sqrt{\frac{n}{v}}(X - m)$, where

$$X = \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)$$

is the empirical mean of the data and

$$v = \frac{1}{n-1}\left(\sum_{i=1}^{n} (x_i - X)^2\right)$$

is the empirical variance.

`stats::tTest(data, m)` returns the list `[PValue = p, StatValue = t]`, where the observed significance level `p` is computed as `p = stats::tCDF(n - 1)(t)`.

`stats::tTest(data, m, Normal)` returns the list `[PValue = p, StatValue = t]`, where the observed significance level `p` is computed as `p = stats::normalCDF(0, 1)(t)`. For large n, this is an approximation of `stats::tCDF(n - 1)(t)`.

Intuitively, `p` corresponds to the "probability" that the true mean of the data (the expectation value of the underlying distribution) is larger than m.

The most relevant information returned by `stats::tTest` is the observed significance level `PValue = p`. It has to be interpreted in the following way:

The t-test may be used as a one-tailed test of the null hypothesis: "the true mean of the data is larger than m". In this case, the null hypothesis may be rejected at level α if the observed significance level p satisfies $p < \alpha$.

Alternatively, the t-test may also be used as a one-tailed test of the null hypothesis: "the true mean of the data is smaller than m". In this case, the null hypothesis may be rejected at level α if the observed "significance level" p satisfies $p > 1 - \alpha$.

Alternatively, the t-test may also be used as a two-tailed test of the null hypothesis: "the true mean of the data is m". If the observed "significance level" `p` returned by `stats::tTest` satisfies either $p < \frac{\alpha}{2}$ or $p > 1 - \frac{\alpha}{2}$ for some given level $0 < \alpha < 1$, this null hypothesis may be rejected at level α.

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

10 experiments produced the values 1, - 2, 3, - 4, 5, - 6, 7, - 8, 9, 10, which are assumed to be normally distributed with unknown mean and variance. The empirical mean of the sample data is 1.5. There is only a small probability p = 0.057... that the true mean is larger than 5.0:

```
data := [1, -2, 3, -4, 5, -6, 7, -8, 9, 10]:
stats::tTest(data, 5.0)
```

$$[\text{PValue} = 0.05756660092,\ \text{StatValue} = -1.743955077]$$

We compare this result with the observed significance level computed via a standard normal distribution:

```
stats::tTest(data, 5.0, Normal)
```

$$[\text{PValue} = 0.04058346175,\ \text{StatValue} = -1.743955077]$$

The approximation of the observed significance level p by the standard normal distribution is rather poor because of the small sample size. Next, we consider a larger sample. The true mean of the random data should be 10:

```
r := stats::normalRandom(10, 12, Seed = 0):
data := [r() $ i = 1..100]:
stats::tTest(data, 10);
```

$$[\text{PValue} = 0.2129644942,\ \text{StatValue} = -0.7994751641]$$

```
stats::tTest(data, 10, Normal)
```

$$[\text{PValue} = 0.212007471,\ \text{StatValue} = -0.7994751641]$$

With the observed significance level of p = 0.212..., the data are not disqualified as having the true mean 10. For samples of this size, the normal distribution approximates the t-distribution well.

```
delete data, r:
```

Parameters

x_1, x_2, …

The statistical data: arithmetical expressions

m

The estimate for the true mean of the data: an arithmetical expression

s

A sample of domain type `stats::sample`.

c

An integer representing a column index of the sample `s`. This column provides the data x_1, x_2 etc. There is no need to specify a column number `c` if the sample has only one non-string column.

Options

Normal

Compute the observed significance level by a standard normal distribution instead of a t-distribution.

Return Values

a list of two equations `[PValue = p, StatValue = t]` with numerical values `p` and `t`. See the `Details' section below for the interpretation of these values.

If the variance of the data vanishes, `FAIL` is returned.

Algorithms

If the data are normally distributed with expectation value ('true mean') μ, the variable $T = \sqrt{\frac{n}{v}}(X - \mu)$ is t-distributed with n - 1 degrees of freedom. The probability of the event that T attains values not larger than t is $Pr(T \le t)$=`stats::tCDF(n - 1)(t)`.

See Also

MuPAD Functions

`stats::csGOFT` | `stats::ksGOFT` | `stats::mean` | `stats::normalCDF` | `stats::stdev` | `stats::swGOFT` | `stats::tCDF`

stats::uniformCDF

Cumulative distribution function of the uniform distribution

Syntax

```
stats::uniformCDF(a, b)
```

Description

`stats::uniformCDF(a, b)` returns a procedure representing the cumulative distribution function

$$x \to \frac{x-a}{b-a}$$

of the uniform distribution on the interval $[a, b]$.

The procedure `f := stats::uniformCDF(a, b)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

If $x < a$ can be decided, then `f(x)` returns 0. If $x > b$ can be decided, then `f(x)` returns the value 1. If $a \le x$ and $x \le b$ can be decided, then `f(x)` returns the value `(x - a)/(b - a)`.

If x is a real floating-point number and both a and b can be converted to real floating-point numbers, then these values are returned as floating-point numbers. Otherwise, symbolic expressions are returned.

The function `f` reacts to properties of identifiers set via `assume`. If x is a symbolic expression with the property $x \le a$, or $x \ge b$, or $a \le x$ and $x \le b$, then the corresponding values are returned.

`f(x)` returns the symbolic call `stats::uniformCDF(a, b)(x)` if it cannot be decided whether x lies in the interval $[a, b]$.

Numerical values for `a` and `b` are only accepted if they are real and $a \le b$.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the cumulative distribution function on the interval [- 3, 2 π] at various points:

```
f := stats::uniformCDF(-3, 2*PI):
f(-infinity), f(-3), f(0.5), f(2/3), f(3.0), f(PI), f(infinity)
```

$$0,\ 0,\ 0.3770257605,\ \frac{11}{3\ (2\ \pi + 3)},\ 0.6463298751,\ \frac{\pi + 3}{2\ \pi + 3},\ 1$$

```
delete f:
```

Example 2

If `x` is a symbolic object without properties, then it cannot be decided whether $a \le x \le b$ holds. A symbolic function call is returned:

```
f := stats::uniformCDF(a, b): f(x)
```

$$\text{stats::uniformCDF}(a,\ b)(x)$$

With suitable properties, it can be decided whether $a \le x \le b$ holds. An explicit expression is returned:

```
assume(x < a): f(x)
```

$$0$$

Note that `assume(x < a)` attached properties both to `a` and `x`. With the next call, we overwrite the property attached to x. However, the property attached to a has to be 'unassumed' as well to avoid inconsistent assumptions $x < a$ and $x > b$:

```
unassume(a): assume(x > b): f(x)
```

$$1$$

```
assume(a <= x <= b): f(x)
```

$$\frac{a-x}{a-b}$$

```
assume(b > a): f(a + (b - a)/3)
```

$$\frac{1}{3}$$

```
unassume(x): unassume(a): unassume(b): delete f:
```

Example 3

We use symbolic arguments:

```
f := stats::uniformCDF(a, b): f(3), f(3.0)
```

$$\text{stats::uniformCDF}(a, b)(3),\ \text{stats::uniformCDF}(a, b)(3.0)$$

When numerical values are assigned to a and b, the function `f` starts to produce numerical values:

```
a := 0: b := PI: f(3), f(3.0)
```

$$\frac{3}{\pi},\ 0.9549296586$$

```
delete f, a, b:
```

Parameters

a, b

arithmetical expressions representing real values; $a \leq b$ is assumed.

Return Values

procedure.

See Also

MuPAD Functions

`stats::uniformPDF` | `stats::uniformQuantile` | `stats::uniformRandom`

stats::uniformPDF

Probability density function of the uniform distribution

Syntax

```
stats::uniformPDF(a, b)
```

Description

`stats::uniformPDF(a, b)` returns a procedure representing the probability density function

$$x \to \frac{1}{b-a}$$

of the uniform distribution on the interval $[a, b]$.

The procedure `f := stats::uniformPDF(a, b)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

If $x < a$ or $x > b$ can be decided, then `f(x)` returns 0. If $a \le x$ and $x \le b$ can be decided, then `f(x)` returns the value `1/(b - a)`.

If x is a real floating-point number and both a and b can be converted to real floating-point numbers, then these values are returned as floating-point numbers. Otherwise, symbolic expressions are returned.

The function `f` reacts to properties of identifiers set via `assume`. If x is a symbolic expression with the property $x < a$, or $x > b$, or $a \le x$ and $x \le b$, then the corresponding values are returned.

`f(x)` returns the symbolic call `stats::uniformPDF(a, b)(x)` if it cannot be decided whether x lies in the interval $[a, b]$.

Numerical values for `a` and `b` are only accepted if they are real and $a \le b$.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the probability density function on the interval [- 3, 2 π] at various points:

```
f := stats::uniformPDF(-3, 2*PI):
f(-infinity), f(-PI), f(-3.0), f(1/2), f(0.5), f(PI), f(infinity)
```

$$0,\ 0,\ 0.1077216458,\ \frac{1}{2\,\pi + 3},\ 0.1077216458,\ \frac{1}{2\,\pi + 3},\ 0$$

```
delete f:
```

Example 2

If `x` is a symbolic object without properties, then it cannot be decided whether $a \le x \le b$ hold. A symbolic function call is returned:

```
f := stats::uniformPDF(a, b): f(x)
```

$$\text{stats::uniformPDF}(a,\ b)(x)$$

With suitable properties, it can be decided whether $a \le x \le b$ holds. An explicit expression is returned:

```
assume(x < a): f(x)
```

$$0$$

Note that `assume(x < a)` attached properties both to `a` and `x`. With the next call, we overwrite the property attached to x. However, the property attached to a has to be 'unassumed' as well to avoid inconsistent assumptions $x < a$ and $x > b$:

```
unassume(a): assume(x > b): f(x)
```

$$0$$

```
assume(a <= x <= b): f(x)
```

$$-\frac{1}{a-b}$$

```
assume(b > a): f(a + (b - a)/3)
```

$$-\frac{1}{a-b}$$

```
unassume(x): unassume(a): unassume(b): delete f:
```

Example 3

We use symbolic arguments:

```
f := stats::uniformPDF(a, b): f(x)
```

$$\text{stats::uniformPDF}(a, b)(x)$$

When numerical values are assigned to a and b, the function `f` starts to produce numerical values:

```
a := 0: b := PI: f(3), f(3.0)
```

$$\frac{1}{\pi},\ 0.3183098862$$

```
delete f, a, b:
```

Parameters

a, b

arithmetical expressions representing real values; $a \le b$ is assumed.

Return Values

procedure.

See Also

MuPAD Functions

`stats::uniformCDF` | `stats::uniformQuantile` | `stats::uniformRandom`

stats::uniformQuantile

Quantile function of the uniform distribution

Syntax

```
stats::uniformQuantile(a, b)
```

Description

`stats::uniformQuantile(a, b)` returns a procedure representing the quantile function (inverse) of the cumulative distribution function `stats::uniformCDF(a, b)` of the uniform distribution on the interval $[a, b]$. For $0 \le x \le 1$, the quantile function is given by $x \to a + x\,(b - a)$.

The procedure `f := stats::uniformQuantile(a, b)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

If `x` is a real number between `0` and `1` and `a` and `b` can be converted to floating-point numbers, then `f(x)` returns the value $a + x\,(b - a)$ as a floating-point number. Otherwise, this value is returned as a symbolic expression.

Numerical values of x are only accepted if $0 \le x \le 1$.

Numerical values for `a` and `b` are only accepted if they are real and $a \le b$.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the quantile function over the interval $\left[2,\ \frac{11}{4}\right]$ at various points:

```
f := stats::uniformQuantile(2, 11/4):
f(0), f(1/10), f(0.5), f(1 - 10^(-5)), f(1)
```

$$2,\ \frac{83}{40},\ 2.375,\ \frac{1099997}{400000},\ \frac{11}{4}$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::uniformQuantile(a, b): f(x), f(9/10)
```

$$a - x\,(a - b),\ \frac{a}{10} + \frac{9\,b}{10}$$

When positive real values are assigned to `a` and `b`, the function `f` starts to produce numerical values:

```
a := 3: b := 11/2: f(0.999), f(1 - sqrt(2)/10^5)
```

$$5.4975,\ \frac{11}{2} - \frac{\sqrt{2}}{40000}$$

```
delete f, a, b:
```

Parameters

a, b

arithmetical expressions representing real values; $a \le b$ is assumed.

Return Values

procedure.

See Also

MuPAD Functions

`stats::uniformCDF` | `stats::uniformPDF` | `stats::uniformRandom`

stats::uniformRandom

Generate a random number generator for uniformly continous deviates

Syntax

```
stats::uniformRandom(a, b, <Seed = s>)
```

Description

`stats::uniformRandom(a, b)` returns a procedure that produces uniformly continous deviates (random numbers) on the interval $[a, b)$.

The procedure `f := stats::uniformRandom(a, b)` can be called in the form `f()`. The return value of `f()` is either a floating-point number or a symbolic expression:

If `a` and `b` can be converted to floating-point numbers, then `f()` returns a floating point number between a and b.

In all other cases, `stats::uniformRandom(a, b)()` is returned symbolically.

Numerical values of `a` and `b` are only accepted if they are real and $a \le b$.

The values `X = f()` are distributed randomly according to the cumulative distribution function of the uniform distribution on the interval $[a, b)$. For any $a \le x \le b$, the probability that $X \le x$ is given by $\frac{(x-a)}{(b-a)}$.

Without the option `Seed = s`, an initial seed is chosen internally. This initial seed is set to a default value when MuPAD is started. Thus, each time MuPAD is started or re-initialized with the `reset` function, random generators produce the same sequences of numbers.

Note: In contrast to the function `random`, the generators produced by `stats::uniformRandom` do not react to the environment variable `SEED`.

For efficiency, it is recommended to produce sequences of K random numbers via `f := stats::uniformRandom(a, b): f() $ k = 1..K` rather than by `stats::uniformRandom(a, b)() $ k = 1..K` The latter call produces a sequence of generators each of which is called once. Also note that

```
stats::uniformRandom(a, b, Seed = n)() $k = 1..K;
```

does not produce a random sequence, because a sequence of freshly initialized generators would be created each of them producing the same number.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We generate uniform deviates on the interval $[2, 7)$:

```
f := stats::uniformRandom(2, 7): f() $ k = 1..4
```

$$3.351790828,\ 6.155185894,\ 2.76578258,\ 6.974063904$$

```
delete f:
```

Example 2

With symbolic parameters, no random floating-point numbers can be produced:

```
f := stats::uniformRandom(a, b): f()
```

$$\text{stats::uniformRandom}(a,\ b)()$$

When a and b evaluate to real numbers, `f` starts to produce random floating point numbers:

```
a := PI: b := 10: f() $ k = 1..4
```

```
4.967800682, 4.377232316, 6.242162399, 7.796955809
```

```
delete f, a, b:
```

Example 3

We use the option `Seed = s` to reproduce a sequence of random numbers:

```
f := stats::uniformRandom(0, 10, Seed = 10^3): f() $ k = 1..4
```

```
8.633422729, 0.1225672185, 6.622938516, 5.069443372
```

```
g := stats::uniformRandom(0, 10, Seed = 10^3): g() $ k = 1..4
```

```
8.633422729, 0.1225672185, 6.622938516, 5.069443372
```

```
f() = g(), f() = g()
```

```
0.6809567809 = 0.6809567809, 0.0395345751 = 0.0395345751
```

```
delete f, g:
```

Parameters

a, b

arithmetical expressions representing real values; $a \le b$ is assumed.

Options

Seed

Option, specified as `Seed = s`

Initializes the random generator with the integer seed `s`. s can also be the option `CurrentTime`, to make the seed depend on the current time.

This option serves for generating generators that return predictable sequences of pseudo-random numbers. The generator is initialized with the seed `s` which may be an arbitrary integer. Several generators with the same initial seed produce the same sequence of numbers.

When this option is used, the parameters `a` and `b` must be convertible to floating-point numbers at the time when the random generator is generated.

Return Values

procedure.

Algorithms

Uniform deviates on the interval $[a, b)$ are produced via `a + (b - a)*frandom()`.

See Also

MuPAD Functions

`frandom` | `random` | `stats::uniformCDF` | `stats::uniformPDF` | `stats::uniformQuantile`

stats::unzipCol

Extract columns from a list of lists

Syntax

```
stats::unzipCol(list)
```

Description

`stats::unzipCol` extracts the columns of a matrix structure encoded by a list of lists.

`stats::unzipCol` treats a list of lists like a list of rows of a `stats::sample` and extracts the columns. In conjunction with `stats::sample2list` it is useful for extracting the columns of a `stats::sample`.

`stats::unzipCol` is the inverse of `stats::zipCol`.

Examples

Example 1

We extract the columns from a list of rows representing a matrix structure:

```
stats::unzipCol([[a11, a12], [a21, a22], [a31, a32]])
```

```
[a11, a21, a31], [a12, a22, a32]
```

Example 2

A list of rows is used to create a sample:

```
stats::sample([[123, s, 1/2], [442, s, -1/2], [322, p, -1/2]])
```

```
123   s    1/2
442   s   -1/2
322   p   -1/2
```

We re-convert the sample to a list of lists:

```
stats::sample2list(%)
```

$$\left[\left[123,\ s,\ \frac{1}{2}\right],\ \left[442,\ s,\ -\frac{1}{2}\right],\ \left[322,\ p,\ -\frac{1}{2}\right]\right]$$

Finally, we extract the columns:

```
stats::unzipCol(%)
```

$$[123,\ 442,\ 322],\ [s,\ s,\ p],\ \left[\frac{1}{2},\ -\frac{1}{2},\ -\frac{1}{2}\right]$$

Parameters

`list`

A list of lists.

Return Values

Sequence of lists to be regarded as columns.

See Also

MuPAD Functions

`stats::col` | `stats::sample2list` | `stats::zipCol`

stats::variance

Variance of a data sample

Syntax

```
stats::variance(x1, x2, …, <Sample | Population>)
```

```
stats::variance([x1, x2, …], <Sample | Population>)
```

```
stats::variance(s, <c>, <Sample | Population>)
```

Description

`stats::variance( x1, x2, ..., xn)` returns the variance

$$\frac{1}{n-1}\left(\sum_{i=1}^{n}(x_i-\overline{x})^2\right),$$

where $\overline{x}$ is the arithmetic mean of the data x_i.

`stats::variance( x1, x2, ..., xn, Population)` returns

$$\frac{1}{n}\left(\sum_{i=1}^{n}(x_i-\overline{x})^2\right).$$

The variance is the square of the standard deviation.

The column index `c` is optional, if the data are given by a `stats::sample` object containing only one non-string column. Cf. “Example 3” on page 30-452.

External statistical data stored in an ASCII file can be imported into a MuPAD session via `import::readdata`. In particular, see Example 1 of the corresponding help page.

Examples

Example 1

We calculate the variance of three values:

```
stats::variance(2, 3, 5)
```

$$\frac{7}{3}$$

Alternatively, the data may be passed as a list:

```
stats::variance([2, 3, 5])
```

$$\frac{7}{3}$$

Example 2

We create a sample:

```
stats::sample([[a1, b1, c1], [a2, b2, c2]])
```

```
a1  b1  c1
a2  b2  c2
```

The variance of the second column is:

```
expand(stats::variance(%, 2))
```

$$\frac{b1^2}{2} - b1\ b2 + \frac{b2^2}{2}$$

Example 3

We create a sample consisting of one string column and one non-string column:

```
stats::sample([["1996", 1242], ["1997", 1353], ["1998", 1142]])
```

```
"1996"   1242
"1997"   1353
"1998"   1142
```

We compute the variance of the second column. In this case this column does not have to be specified, since it is the only non-string column:

```
float(stats::variance(%))
```

```
     11140.33333
```

We repeat the computation with the option `Population`:

```
float(stats::variance(%2, Population))
```

```
     7426.888889
```

Parameters

x_1, x_2, …

The statistical data: arithmetical expressions

s

A sample of domain type `stats::sample`

c

An integer representing a column index of the sample `s`. This column provides the data x_1, x_2, ….

Options

`Population`, `Sample`

With `Sample`, the data are regarded as a “sample”, not as a full population. The default is `Sample`.

Return Values

Arithmetical expression.

See Also

MuPAD Functions

`stats::geometricMean` | `stats::harmonicMean` | `stats::mean` | `stats::median` | `stats::modal` | `stats::quadraticMean` | `stats::stdev`

stats::weibullCDF

Cumulative distribution function of the Weibull distribution

Syntax

```
stats::weibullCDF(a, b)
```

Description

`stats::weibullCDF(a, b)` returns a procedure representing the cumulative distribution function

$$x \to 1 - e^{-\left(\frac{x}{b}\right)^a}$$

of the Weibull distribution with shape parameter $a > 0$ and scale parameter $b > 0$.

The procedure `f := stats::weibullCDF(a, b)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

If $x \leq 0$ can be decided, then `f(x)` returns `0`. If $x \geq 0$ can be decided, then `f(x)` returns the value `1 - exp(-(x/b)^a)`.

If x is a floating-point number and both a and b can be converted to positive floating-point numbers, then these values are returned as floating-point numbers. Otherwise, symbolic expressions are returned.

The function `f` reacts to properties of identifiers set via `assume`. If x is a symbolic expression with the property $x \leq 0$ or $x \geq 0$, the corresponding values are returned.

The call `f(- infinity )` returns `0`.

The call `f( infinity )` returns `1`.

`f(x)` returns the symbolic call `stats::weibullCDF(a, b)(x)` if neither $x \leq 0$ nor $x \geq 0$ can be decided.

Numerical values for a and b are only accepted if they are real and positive.

Environment Interactions

The function is sensitive to the environment variable DIGITS which determines the numerical working precision. The procedure generated by stats::weibullCDF reacts to properties of identifiers set via assume.

Examples

Example 1

We evaluate the cumulative distribution function with a = 2 and b = 1 at various points:

```
f := stats::weibullCDF(2, 1):
f(-infinity), f(-3), f(0.5), f(2/3), f(PI), f(infinity)
```

$$0,\ 0,\ 0.2211992169,\ 1-e^{-\frac{4}{9}},\ 1-e^{-\pi^2},\ 1$$

```
delete f:
```

Example 2

If x is a symbolic object without properties, then it cannot be decided whether $x \geq 0$ holds. A symbolic function call is returned:

```
f := stats::weibullCDF(a, b): f(x)
```

$$\text{stats::weibullCDF}(a,\ b)(x)$$

With suitable properties, it can be decided whether $x \geq 0$ holds. An explicit expression is returned:

```
assume(0 < x): f(x)
```

$$1-e^{-\left(\frac{x}{b}\right)^a}$$

```
unassume(x): delete f:
```

Example 3

We use symbolic arguments:

```
f := stats::weibullCDF(a, b): f(x)
```

$$\text{stats::weibullCDF}(a, b)(x)$$

When numerical values are assigned to `a` and `b`, the function `f` starts to produce numerical values:

```
a := 2: b := 1: f(3), f(3.0)
```

$$1 - e^{-9},\ 0.9998765902$$

```
delete f, a, b:
```

Parameters

a

The shape parameter: an arithmetical expression representing a positive real value

b

The scale parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`stats::weibullPDF` | `stats::weibullQuantile` | `stats::weibullRandom`

stats::weibullPDF

Probability density function of the Weibull distribution

Syntax

```
stats::weibullPDF(a, b)
```

Description

`stats::weibullPDF(a, b)` returns a procedure representing the probability density function

$$x \to \frac{a\,x^{a-1}\,e^{-\left(\frac{x}{b}\right)^a}}{b^a}$$

of the Weibull distribution with shape parameter `a` > 0 and scale parameter `b` > 0.

The procedure `f := stats::weibullPDF(a, b)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number or a symbolic expression:

If $x \le 0$ can be decided, then `f(x)` returns `0`. If $x > 0$ can be decided, then `f(x)` returns the value $\frac{a\,x^{a-1}\,e^{-\left(\frac{x}{b}\right)^a}}{b^a}$.

If x is a floating-point number and both a and b can be converted to positive floating-point numbers, then these values are returned as floating-point numbers. Otherwise, symbolic expressions are returned.

The function `f` reacts to properties of identifiers set via `assume`. If x is a symbolic expression with the property $x \le 0$ or $x > 0$, the corresponding values are returned.

`f(- infinity )` and `f( infinity )` return `0`.

`f(x)` returns the symbolic call `stats::weibullPDF(a, b)(x)` if neither $x \le 0$ nor $x > 0$ can be decided.

Numerical values for a and b are only accepted if they are real and positive.

Environment Interactions

The function is sensitive to the environment variable DIGITS which determines the numerical working precision. The procedure generated by stats::weibullPDF reacts to properties of identifiers set via assume.

Examples

Example 1

We evulate the probability density function with a = 2 and b = 1 at various points:

```
f := stats::weibullPDF(2, 1): f(1/5), f(0.5), f(-1), f(x)
```

$$\frac{2\,e^{-\frac{1}{25}}}{5},\ 0.7788007831,\ 0,\ \text{stats::weibullPDF}(2,\ 1)(x)$$

```
delete f:
```

Example 2

If x is a symbolic object without properties, then it cannot be decided whether $x > 0$ holds. A symbolic function call is returned:

```
f := stats::weibullPDF(a, b): f(x)
```

$$\text{stats::weibullPDF}(a,\ b)(x)$$

With suitable properties, it can be decided whether $x > 0$ holds. An explicit expression is returned:

```
assume(0 < x): f(x)
```

$$\frac{a\,x^{a-1}\,e^{-\left(\frac{x}{b}\right)^a}}{b^a}$$

```
unassume(x): delete f:
```

Example 3

We use symbolic arguments:

```
f := stats::weibullPDF(a, b): f(x), f(3)
```

$$\text{stats::weibullPDF}(a, b)(x),\ \frac{3^{a-1}\,a\,e^{-\left(\frac{3}{b}\right)^a}}{b^a}$$

When numerical values are assigned to `a` and `b`, the function `f` starts to produce numerical values:

```
a := 2: b := 1: f(3), f(3.0)
```

$$6\,e^{-9},\ 0.0007404588245$$

```
delete f, a, b:
```

Parameters

a

The shape parameter: an arithmetical expression representing a positive real value

b

The scale parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`stats::weibullCDF` | `stats::weibullQuantile` | `stats::weibullRandom`

stats::weibullQuantile

Quantile function of the Weibull distribution

Syntax

```
stats::weibullQuantile(a, b)
```

Description

`stats::weibullQuantile(a, b)` returns a procedure representing the quantile function (inverse)

$$x \to b\,(-\ln(1-x))^{1/a}$$

of the cumulative distribution function `stats::weibullCDF(a, b)`. For $0 \le x \le 1$, the solution of *stats*::*weibullCDF*(a, b)(y) = x is given by

$$y = \text{stats::weibullQuantile}(a, b)(x)$$

.

The procedure `f := stats::weibullQuantile(a, b)` can be called in the form `f(x)` with an arithmetical expression `x`. The return value of `f(x)` is either a floating-point number, `infinity`, or a symbolic expression:

If `x` is a real float point number between `0` and `1` and `a` and `b` can be converted to positive floating-point numbers, then `f(x)` returns a floating-point number.

The calls `f(1)` and `f(1.0)` produce `infinity`.

In all other cases, $f(x)$ returns the symbolic expression `a - ln(1 - x)/b`.

Numerical values of x are only accepted if $0 \le x \le 1$.

Numerical values of a and b are only accepted if they are real and positive.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We evaluate the quantile function with `a` = 2 and `b` = $\frac{3}{4}$ at various points:

```
f := stats::weibullQuantile(2, 3/4):
f(0), f(1/10), f(0.5), f(1 - 10^(-10)), f(1)
```

$$0,\ \frac{3\sqrt{-\ln\left(\frac{9}{10}\right)}}{4},\ 0.6244159584,\ \frac{3\sqrt{\ln(10000000000)}}{4},\ \infty$$

The value `f(x)` satisfies `stats::weibullCDF(2, 3)(f(x))` = `x`:

```
stats::weibullCDF(2, 3/4)(f(0.987654321))
```

$$0.987654321$$

```
delete f:
```

Example 2

We use symbolic arguments:

```
f := stats::weibullQuantile(a, b): f(x), f(1/PI), f(0.99)
```

$$b\,(-\ln(1-x))^{1/a},\ b\left(-\ln\left(1-\frac{1}{\pi}\right)\right)^{1/a},\ 4.605170186^{1/a}\,b$$

When suitable numerical values are assigned to `a` and `b`, the function `f` starts to produce numerical values:

```
a := 7: b := 1/8: f(0.999), f(999/1000)
```

$$0.1647461562,\ \frac{\ln(1000)^{1/7}}{8}$$

Numerical values for x are only accepted if $0 \le x \le 1$:

```
f(0.5)
```

$$0.1186235162$$

```
f(2)
```

```
Error: An argument x with 0 <= x <= 1 is expected. [f]
```

```
delete f, a, b:
```

Parameters

a

The shape parameter: an arithmetical expression representing a positive real value

b

The scale parameter: an arithmetical expression representing a positive real value

Return Values

procedure.

See Also

MuPAD Functions

`stats::weibullCDF` | `stats::weibullPDF` | `stats::weibullRandom`

stats::weibullRandom

Generate a random number generator for Weibull deviates

Syntax

```
stats::weibullRandom(a, b, <Seed = s>)
```

Description

`stats::weibullRandom(a, b)` returns a procedure that produces Weibull deviates (random numbers) with shape parameter `a` > 0 and scale parameter `b` > 0.

The procedure `f := stats::weibullRandom(a, b)` can be called in the form `f()`. The return value of `f()` is either a floating-point number or a symbolic expression:

If `a` and `b` can be converted to positive floating-point numbers, then `f()` returns a nonnegative floating-point number.

In all other cases, `stats::weibullRandom(a, b)()` is returned symbolically.

Numerical values of `a` and `b` are only accepted if they are real and positive.

The values `X = f()` are distributed randomly according to the cumulative distribution function of the Weibull distribution with parameters `a` and `b`. For any $0 \le x$, the probability that $X \le x$ is given by $1 - \frac{1}{e^{\left(\frac{x}{b}\right)^a}}$.

Without the option `Seed = s`, an initial seed is chosen internally. This initial seed is set to a default value when MuPAD is started. Thus, each time MuPAD is started or re-initialized with the `reset` function, random generators produce the same sequences of numbers.

Note: In contrast to the function `random`, the generators produced by `stats::weibullRandom` do not react to the environment variable `SEED`.

For efficiency, it is recommended to produce sequences of K random numbers via

```
f := stats::weibullRandom(a, b): f() $k = 1..K;
```

rather than by

```
stats::weibullRandom(a, b)() $k = 1..K;
```

The latter call produces a sequence of generators each of which is called once. Also note that

```
stats::weibullRandom(a, b, Seed = n)() $k = 1..K;
```

does not produce a random sequence, because a sequence of freshly initialized generators would be created each of them producing the same number.

Environment Interactions

The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Examples

Example 1

We generate Weibull deviates with parameters a = 2 and b = $\frac{3}{4}$:

```
f := stats::weibullRandom(2, 3/4): f() $ k = 1..4
```

```
0.8577612188, 0.3226575883, 1.027334357, 0.05408701056
```

```
delete f:
```

Example 2

With symbolic parameters, no random floating-point numbers can be produced:

```
f := stats::weibullRandom(a, b): f()
```

```
stats::weibullRandom(a, b)()
```

When positive real numbers are assigned to a and b, the function f starts to produce random floating point numbers:

```
a := PI: b := 1/8: f() $ k = 1..4
```

```
0.1366558143, 0.148384594, 0.1161455634, 0.09243509442
```

```
delete f, a, b:
```

Example 3

We use the option Seed = s to reproduce a sequence of random numbers:

```
f := stats::weibullRandom(PI, 3, Seed = 1): f() $ k = 1..4
```

```
2.503931345, 1.888623862, 3.833514832, 3.584419593
```

```
g := stats::weibullRandom(PI, 3, Seed = 1): g() $ k = 1..4
```

```
2.503931345, 1.888623862, 3.833514832, 3.584419593
```

```
f() = g(), f() = g()
```

```
1.742746285 = 1.742746285, 5.240957365 = 5.240957365
```

```
delete f, g:
```

Parameters

a

The shape parameter: an arithmetical expression representing a positive real value

b

The scale parameter: an arithmetical expression representing a positive real value

Options

Seed

Option, specified as `Seed = s`

Initializes the random generator with the integer seed `s`. s can also be the option `CurrentTime`, to make the seed depend on the current time.

This option serves for generating generators that return predictable sequences of pseudo-random numbers. The generator is initialized with the seed `s` which may be an arbitrary integer. Several generators with the same initial seed produce the same sequence of numbers.

When this option is used, the parameters `a` and `b` must be convertible to positive floating-point numbers at the time when the random generator is generated.

Return Values

procedure.

Algorithms

The implemented algorithm for the computation of the Weibull deviates uses the quantile function of the Weibull distribution applied to unformly distributed random numbers on the interval $[0, 1)$.

See Also

MuPAD Functions

`stats::weibullCDF` | `stats::weibullPDF` | `stats::weibullQuantile`

stats::zipCol

Convert a sequence of columns to a list of lists

Syntax

```
stats::zipCol(column1, column2, …)
```

Description

`stats::zipCol(..)` converts a sequence of columns to a format suitable for creating a `stats::sample`.

`stats::zipCol` is useful for converting column data given in lists to a list of lists accepted by `stats::sample`.

`stats::zipCol` is the inverse of `stats::unzipCol`.

Examples

Example 1

We convert a single column to a nested list:

```
stats::zipCol([a, b, c])
```

$$[[a], [b], [c]]$$

This list is accepted by `stats::sample`:

```
stats::sample(%)
```

```
a
b
c
```

Example 2

We build a sample consisting of two columns:

```
column1 := [122, 442, 322]: column2 := [s, s, p]:
stats::zipCol(column1, column2)
```

$$[[122, s], [442, s], [322, p]]$$

```
stats::sample(%)
```

```
122  s
442  s
322  p
```

Parameters

`column1, column2, …`

Lists.

Return Values

List of lists.

See Also

MuPAD Functions

`stats::sample2list` | `stats::unzipCol`

stringlib – Manipulating Strings

stringlib::collapseWhitespace
stringlib::contains
stringlib::format
stringlib::formatf
stringlib::formatTime
stringlib::lower
stringlib::maskMeta
stringlib::order
stringlib::pos
stringlib::random
stringlib::readText
stringlib::remove
stringlib::split
stringlib::subs
stringlib::subsop
stringlib::upper
stringlib::validIdent

stringlib::collapseWhitespace

Collapse whitespace in strings

Syntax

```
stringlib::collapseWhitespace(string)
```

Description

`stringlib::collapseWhitespace(string)` replaces each whitespace sequence in `string` by one space.

The characters `" "` (space), `"\n"` (newline), `"\t"` (tabulator) and `"\r"` (return) are called whitespace.

`stringlib::collapseWhitespace(string)` replaces all whitespace characters by one space and following all sequences of spaces by one space in `string`.

When `string` contains no whitespace or only single space characters, the string is returned without changes.

`stringlib::collapseWhitespace` is a function of the system kernel.

Examples

Example 1

In the following examples all whitespace is collapsed:

```
stringlib::collapseWhitespace("          ")
```

```
    ""
```

```
stringlib::collapseWhitespace("MuPAD    is       nice.")
```

```
"MuPAD is nice."
```

```
stringlib::collapseWhitespace("          ")
```

```
" "
```

In this example no whitespace can be collapsed:

```
stringlib::collapseWhitespace(""),
stringlib::collapseWhitespace("12345"),
stringlib::collapseWhitespace("MuPAD is nice.")
```

```
"", "12345", "MuPAD is nice."
```

Example 2

`stringlib::collapseWhitespace` can be useful for output MuPAD code. The procedure is only an expample:

```
f:= proc(x) local y; begin y:= 2*x; x + y end
```

```
proc f(x) ... end
```

```
print(f)
```

```
proc f(x) ... end
```

```
expr2text(f)
```

```
"proc(x)   name f;   local y; begin   y := 2*x;    x + y end_proc"
```

```
stringlib::collapseWhitespace(%)
```

```
"proc(x) name f; local y; begin y := 2*x; x + y end_proc"
```

Parameters

`string`

Any MuPAD string

Return Values

Given string with collapsed whitespace

See Also

MuPAD Functions

`stringlib::remove` | `stringlib::subs` | `subs`

stringlib::contains

Test for substring

Syntax

```
stringlib::contains(string1, string2, options)
```

Description

`stringlib::contains(string1, string2)` checks, whether `string1` contains another string `string2`.

Examples

Example 1

If called without options, `stringlib::contains` simply returns `TRUE` or `FALSE`.

```
stringlib::contains("abcdeabcdeabcde", "bc")
```

```
    TRUE
```

```
stringlib::contains("abcdeabcdeabcde", "cb")
```

```
    FALSE
```

```
stringlib::contains("abcdeabcdeabcde", "bc", Index)
```

```
    2
```

```
stringlib::contains("abcdeabcdeabcde", "cb", Index)
```

```
    FALSE
```

```
stringlib::contains("abcdeabcdeabcde", "bc", IndexList)
```

```
    [2, 7, 12]
```

```
stringlib::contains("abcdeabcdeabcde", "cb", IndexList)
```

```
    []
```

Example 2

The following call does *not* return `[1,2]` because the first matching substring has not ended when the second begins.

```
stringlib::contains("aaa", "aa", IndexList)
```

```
    [1]
```

Parameters

`string1`, `string2`

Non empty string

Options

`Index`

Causes the first index position at which `string2` appears in `string1` to be returned as integer. The return value is `FALSE`, if `string2` occurs nowhere in `string1`.

`IndexList`

Causes the list of all positions at which `string2` appears in `string1` to be returned. The returned list is empty if `string2` occurs nowhere in `string1`.

An occurrence of `string2` is not detected if overlapped by the tail of a previously detected occurrence. See “Example 2” on page 31-6.

Return Values

TRUE, an integer, or a list of integers that determines the position (if an option is given), when `string1` contains `string2`, otherwise FALSE or an empty list.

See Also

MuPAD Functions

`contains` | `stringlib::pos` | `strmatch`

stringlib::format

Formatting a string

Syntax

```
stringlib::format(string1, width, <Left | Center | Right>, <fill_char>)
```

Description

`stringlib::format` adjusts the length of a string.

If `width` is less than the length of the given string `string1`, the substring consisting of the first `width` characters of `string1` is returned.

If `width` exceeds the length of `string1`, the given string will be filled with the necessary number of spaces or the optional `fill_char`. These are inserted at the end in case of left alignment, or at the beginning in case of right alignment. In case of centering, the same number of filling characters is placed at the beginning and at the end, but one more is placed at the end if their total number is odd.

If `alignment` is not given, left alignment is used by default.

Examples

Example 1

By default, a string of length 5 is adjusted to length 10 by inserting five space characters at the end. Since white spaces are collapsed in typesetting output, we use `print`:

```
print(Plain, stringlib::format("abcde", 10))
```

```
"abcde     "
```

In the case of centering, three spaces are inserted at the end and two at the beginning.

```
print(Plain, stringlib::format("abcde", 10, Center))
```

```
"  abcde   "
```

Instead of the space character, also any other character may be used as a filling character.

```
stringlib::format("abcde", 10, Right, ".")
```

```
".....abcde"
```

```
stringlib::format("abcde", 10, ".")
```

```
"abcde....."
```

Parameters

`string1`

String

`width`

Integer that determines the length of the returned string

`fill_char`

One-character string to fill up the result string

Options

`Left`

Determines that the string will be aligned left

`Center`

Determines that the string will be centered

`Right`

Determines that the string will be aligned right

Return Values

String of length `width` containing the given string

See Also

MuPAD Functions
`stringlib::formatf`

stringlib::formatf

Convert a floating-point number to a string

Syntax

```
stringlib::formatf(f, digits, <strlength>)
```

Description

`stringlib::formatf(f, d)` converts the floating point number `f` into a string after rounding it to `d` digits after the decimal point.

If `d` is a positive integer, a rounded fixed-point representation with `d` digits after the decimal point is returned. If `d` is zero, then a rounded fixed-point representation with one zero after the decimal point is returned. If `d` is negative, then `f` is rounded to `-d` digits before the decimal point and a fixed-point representation with one zero after the decimal point is returned.

The representation of a negative number starts with the sign and no additional spaces. The representation of a nonnegative number starts with a single space character.

If a third argument is specified, then the string returned consists of exactly `strlength` characters. If the converted number `f` requires less room, then it is padded on the left with spaces. If the converted number `f` requires more room, then the last characters are truncated.

Examples

Example 1

Convert the number `123.456` with two characters after the point into a string. Since white spaces are collapsed in typesetting output, we use `print`:

```
print(Plain, stringlib::formatf(123.456, 2))
```

```
" 123.46"
```

The same for `-123.456`:

```
print(Plain, stringlib::formatf(-123.456, 2))
```

```
"-123.46"
```

Convert the number `123.456` with two characters after the point into a string of the length `10`:

```
print(Plain, stringlib::formatf(123.456, 2, 10))
```

```
"    123.46"
```

If the string should only have the length `3`, the whole number does not fit into the string:

```
print(Plain, stringlib::formatf(123.456, 2, 3))
```

```
" 12"
```

Rounding to no number after point:

```
print(Plain, stringlib::formatf(123.456, 0))
```

```
" 123.0"
```

Rounding to one number in front of point:

```
print(Plain, stringlib::formatf(123.456, -1))
```

```
" 120.0"
```

Parameters

f

Floating point number

`digits`

Integer which determines the precision of the number

`strlength`

Integer which determines the length of the returned string

Return Values

`stringlib::formatf` returns a string.

See Also

MuPAD Functions

`stringlib::format`

stringlib::formatTime

Textual description of a time length

Syntax

```
stringlib::formatTime(t)
```

Description

`stringlib::formatTime` returns a textual description such as “5 minutes, 20 seconds” of a time value such as `320 * unit::sec` given as its argument.

When given an integer or floating-point number, `stringlib::formatTime` interprets it as milliseconds, for compatibility with `time`. Cf. “Example 2” on page 31-15.

`stringlib::formatTime` rounds its input to only use two types of unit, cf. “Example 1” on page 31-14.

Examples

Example 1

`stringlib::formatTime` takes a time description and renders it as English text:

```
stringlib::formatTime(1234*unit::sec)
```

```
"20 minutes, 34.0 seconds"
```

Excessive precision is avoided:

```
stringlib::formatTime(12345678*unit::sec)
```

```
"4 months, 3.26 weeks"
```

Example 2

`time` and `rtime` return integers interpreted as milliseconds. `stringlib::formatTime` thus interprets integers (and, for consistency, floating-point numbers) as milliseconds:

```
stringlib::formatTime(1)
```

```
"0.001 seconds"
```

```
stringlib::formatTime(rtime(system("sleep 2")))
```

```
"2.411 seconds"
```

Parameters

`t`

The time to convert: An integer or floating-point value (regarded as milliseconds) or an expression involving time units.

Return Values

string.

See Also

MuPAD Functions
`rtime` | `time`

stringlib::lower

Convert to lower-case

Syntax

```
stringlib::lower(string1)
```

Description

`stringlib::lower(string1)` converts each upper-case letter in the string `string1` to lower-case. All other characters remains unchanged.

If the string contains no upper-case letters, the given string is returned unchanged.

Examples

Example 1

Convert a string to lower-case:

```
stringlib::lower("MuPAD"),
stringlib::lower("Mupad"),
stringlib::lower("MUPAD"),
stringlib::lower("mupad")
```

```
"mupad", "mupad", "mupad", "mupad"
```

Example 2

Compare strings not case sensitive:

```
str_eq:= (str1, str2) ->
           bool(stringlib::lower(str1) = stringlib::lower(str2)):
str_eq("MuPAD", "mupad"), str_eq("Mupad", "MUPAD")
```

```
TRUE, TRUE
```

Parameters

`string1`

Any string

Return Values

String

See Also

MuPAD Functions

`stringlib::format` | `stringlib::upper`

stringlib::maskMeta

Mask regular expression special characters

Syntax

```
stringlib::maskMeta(str)
```

Description

`stringlib::maskMeta(str)` generates a regular expression (for use with `strmatch`) that matches exactly the string `str`.

As of MuPAD version 3.2, `strmatch` uses regular expression matching. To search for some verbatim substring therefore requires "escaping" special characters such as `*` or `()`. `stringlib::maskMeta` performs this task.

Examples

Example 1

Trying to find `"a+b"` in the string `"a+b+c"` via `strmatch` fails due to the special nature of `"+"` in regular expressions, but, for almost the same reason, `"a*b"` is found:

```
strmatch("a+b+c", "a+b"),
strmatch("a+b+c", "a*b")
```

```
FALSE, TRUE
```

Using `stringlib::maskMeta`, we lose the ability of using regular expressions, but can easily search for verbatim strings:

```
strmatch("a+b+c", stringlib::maskMeta("a+b")),
strmatch("a+b+c", stringlib::maskMeta("a*b"))
```

```
TRUE, FALSE
```

The output of `stringlib::maskMeta` is just another string, so combinations with other strings (containing regular expression meta-characters) is possible:

```
strmatch("a+b+c", "^".stringlib::maskMeta("a+b")),
strmatch("a+b+c", "^".stringlib::maskMeta("b+c"))
```

```
TRUE, FALSE
```

Parameters

`str`

Non empty string

Return Values

String

See Also

MuPAD Functions

`stringlib::pos` | `stringlib::subs` | `strmatch`

stringlib::order

Sorting procedure for Sort

Syntax

```
stringlib::order(options)
```

Description

`stringlib::order()` returns a procedure that compares two strings and returns `TRUE` when they are in lexicographical order, otherwise `FALSE`. This procedure can be used as the second argument of `sort`.

Examples

Example 1

Sort strings in lexicographical order:

```
sort(["ab", "a", "abc", "B", "ba", "Ca", "bB", "bb"],
     stringlib::order())
```

```
["B", "Ca", "a", "ab", "abc", "bB", "ba", "bb"]
```

Sort strings in lexicographical order without case sensitivity:

```
sort(["ab", "a", "abc", "B", "ba", "Ca", "bB", "bb"],
     stringlib::order(Nocase))
```

```
["a", "ab", "abc", "B", "ba", "bb", "bB", "Ca"]
```

Sort strings in reverse lexicographical order:

```
sort(["ab", "a", "abc", "B", "ba", "Ca", "bB", "bb"],
```

```
     stringlib::order(Reverse))
```

```
["bb", "ba", "bB", "abc", "ab", "a", "Ca", "B"]
```

Sort strings in reverse lexicographical order without case sensitivity:

```
sort(["ab", "a", "abc", "B", "ba", "Ca", "bB", "bb"],
     stringlib::order(ReverseNocase))
```

```
["Ca", "bB", "bb", "ba", "B", "abc", "ab", "a"]
```

Options

`Lexicographical`

Return a procedure that yields `TRUE` when the two given strings are in lexicographical order.

`Nocase`

Return a procedure that yields `TRUE` when the two given strings are in lexicographical order without case sensitivity.

`Reverse`

Return a procedure that yields `TRUE` when the two given strings are in reverse lexicographical order.

`ReverseNocase`

Return a procedure that yields `TRUE` when the two given strings are in reverse lexicographical order without case sensitivity.

Return Values

Procedure that can be used as second argument of `sort`

See Also

MuPAD Functions
`sort`

stringlib::pos

Position of a substring

Syntax

```
stringlib::pos(string1, string2, <pos>)
```

Description

`stringlib::pos` returns the position of a substring in a string.

The third optional argument must be less than the length of `string1`.

If `string1` does not contain `string2`, then `FAIL` will be returned.

Examples

Example 1

In case of several occurrences of the substring, the position of the first is returned.

```
stringlib::pos("abcdeabcdeabcde", "bc")
```

```
    2
```

Example 2

If a starting point for the search is given, `stringlib::pos` returns the first position at which the substring occurs after that starting point.

```
stringlib::pos("abcdeabcdeabcde", "bc", 5)
```

```
    7
```

Example 3

The result is FAIL if the substring does not occur at all or after the given starting point.

```
stringlib::pos("abcdeabcdeabcde", "bc", 14)
```

```
FAIL
```

Parameters

string1, string2

Non empty string

pos

Integer that determines the first position to search

Return Values

Integer that determines the position or FAIL.

See Also

MuPAD Functions

length | stringlib::contains

stringlib::random

Create a random string

Syntax

```
stringlib::random(<l>, <characters>, options)
```

Description

`stringlib::random()` returns a random string of the default length `7`.

`stringlib::random(l)` with a number or a range `l` returns a random string with length `l` or length in the given range. When the prefix and/or suffix is longer than the given length, `stringlib::random` raises an error message.

`stringlib::random(characters)` with a given list oder set of characters builds the random string of the given characters. When the characters are strings, they are used as single characters, however, the length is exceeded.

`stringlib::random(l, characters)` is a combination of the both last calls. When both parameters are given, the order is significant.

`stringlib` defines the lists `stringlib::lowerLetters`, `stringlib::upperLetters`, `stringlib::digits` and `stringlib::punctuation` with the characters lower letters, upper letters, digits and punctuation.

Examples

Example 1

Create a random string of the default length `7`:

```
stringlib::random()
```

```
"jaR<2oH"
```

Create a random string of the length 3:

```
stringlib::random(3)
```

```
"uRT"
```

Create a random string of the length 2 only of digits:

```
stringlib::random(2, stringlib::digits)
```

```
"53"
```

Create a random string of the length 3 only of digits with prefix "+":

```
stringlib::random(3, stringlib::digits, Prefix = "+")
```

```
"+70"
```

Create a random string of the length 5 only of digits with suffix ".0":

```
stringlib::random(5, stringlib::digits, Suffix = ".0")
```

```
"456.0"
```

Create a random strings of the length 3 to 8 only of letters:

```
stringlib::random(3..8, stringlib::lowerLetters.
                        stringlib::upperLetters) $ k = 1..5
```

```
"BDkkIz", "zkkvYp", "mFobeC", "Vxsjuk", "HHTJbWh"
```

Create a random string of the length 2 to 8 with letters and casual a punctuation:

```
stringlib::random(2..8, (stringlib::lowerLetters.
                         stringlib::upperLetters $ 4).
                        stringlib::punctuation) $ k = 1..12
```

```
"<vlvl<a", "rFxvu", "GUHEBR", "fP", "dIqxr", "wO"Md", "RdFdRlqH", "nv)", "Bo", "HGaiqPt&",
  "AqnF", "do"
```

Create a random string of the length 6 to 8 with letters and equivalent punctuation:

```
stringlib::random(6..8, (stringlib::lowerLetters.
                         stringlib::upperLetters).
                        (stringlib::punctuation $ 2)) $ k = 1..10
```

"n^iY)E", "=l_?sZ", "{i^(Gh>h", "/A+#}/", "L-DM&G-U", "(,^JzNL;", "ZH;|%;i", ")t^.[^}T",
"-v*Es=Z>", "\:AdF{R"

Create random names of the length 4 to 6:

```
stringlib::random(4..6, Name) $ k = 1..12
```

"pbg1", "A98HWb", "eAaR1", "ve_6", "rtxoi", "Iwaw", "Td8cQR", "PAsudJ", "a4AN", "d2ixaC",
"DvLXzh", "yTMCa5"

Create a random password of the length 8 to 10, but without some special characters:

```
EX := {"\\", "\"", "|", "'", "?", "*", "[", "]"}:
stringlib::random(8..10, Exclude = EX)
```

"NRpjhuRkc"

Parameters

`l`

The length of the returned string: a nonnegative integer or a range of nonnegative integers

`characters`

A list or set of characters

`options`

Any of the described options

Options

`Exclude`

Option, specified as `Exclude = characters`

The returned string does not contain characters given in the set or list `characters`.

`Name`

The returned string is a valid MuPAD object name.

`Prefix`

Option, specified as `Prefix = string`

Adds `string` in front of each random string. The length of the returned string is the given length or the default length including the prefix.

`Suffix`

Option, specified as `Suffix = string`

Appends `string` to each random string. The length of the returned string is the given length or the default length including the suffix.

Return Values

Random string of the given length or the default length including the prefix resp. suffix.

See Also

MuPAD Functions
`random` | `SEED` | `stringlib::lower` | `stringlib::upper` | `substring`

stringlib::readText

Read text file

Syntax

```
stringlib::readText(filename)
```

```
stringlib::readText(filename, String, <NoNL>)
```

```
stringlib::readText(filename, String, <Separator = string>)
```

```
stringlib::readText(..., <Encoding = "encodingValue">)
```

Description

`stringlib::readText(filename)` reads all lines of the text file with name "`filename`" and returns a list of strings, one string per line. The linebreaks are not included at the end of each string. The file must be a text file, otherwise the file cannot be read.

`stringlib::readText(filename, Encoding = "encodingValue")` uses the specified encoding to read the file. For the supported encodings, see "Options" on page 31-31. You can use this option with the previously specified syntaxes.

Examples

Example 1

First create a text file that can be read:

```
fprint(Unquoted, Text, "test.txt",
       "This file contains three lines.\n",
       "// this line is a MuPAD comment\n",
       "..............................\n"):
```

By default, `stringlib::readText` returns a list of all lines:

```
stringlib::readText("test.txt")
```

```
["This file contains three lines.", "// this line is a MuPAD comment", "...............................", ""]
```

Because the third line was ended by a newline, the file contains four lines, the last line is empty.

The file can be read as *one* string:

```
stringlib::readText("test.txt", String)
```

```
"This file contains three lines. // this line is a MuPAD comment ..... "
```

When the newlines should be removed, option `NoNL` can be used:

```
stringlib::readText("test.txt", String, NoNL)
```

```
"This file contains three lines.// this line is a MuPAD comment....."
```

Otherwise the newlines can be replaced by another separator:

```
stringlib::readText("test.txt", String, Separator = " ;; ")
```

```
"This file contains three lines. ;; // this line is a MuPAD comment ;; ..... ;; "
```

Example 2

To specify the encoding to read data, use Encoding. The Encoding option applies only to text files that are opened using a file name and not a file descriptor. Open a file and write strings in the encoding “`UTF-8`”:

```
fprint(Unquoted, Text, Encoding="UTF-8", "readtext_test",
       "File to test stringlib::readText\n",
       "Testing to encode characters such as abcäöü"):
```

Specify the encoding to read the file:

```
stringlib::readText("readtext_test", Encoding="UTF-8")
```

```
["File to test stringlib::readText", "Testing to encode characters such as abcäöü"]
```

If you do not specify an encoding, the default system encoding is used. Thus, your output might vary from that shown next. Characters unrecognized by the default system encoding are replaced by the default substitution character for that encoding:

```
stringlib::readText("readtext_test")
```

```
["File to test stringlib::readText", "Testing to encode characters such as abc??????"]
```

Parameters

`filename`

The name of a file as string

`string`

Any string

Options

`String`

With this option, `stringlib::readText` returns *one* string that contains all contents of the read file, including the line breaks as separator of the lines.

`NoNL`

With option `NoNL`, the returned string does not contain the linebreaks between the lines.

`Separator`

Option, specified as `Separator = string`

This option causes `stringlib::readText` to separate all lines by `string` instead of the line break `"\n"`.

`Encoding`

This option lets you specify the character encoding to use. The allowed encodings are:

"Big5"	"ISO-8859-1"	"windows-932"
"EUC-JP"	"ISO-8859-2"	"windows-936"
"GBK"	"ISO-8859-3"	"windows-949"
"KSC_5601"	"ISO-8859-4"	"windows-950"
"Macintosh"	"ISO-8859-9"	"windows-1250"
"Shift_JIS"	"ISO-8859-13"	"windows-1251"
"US-ASCII"	"ISO-8859-15"	"windows-1252"
"UTF-8"		"windows-1253"
		"windows-1254"
		"windows-1257"

The default encoding is system dependent. If you specify the encoding incorrectly, characters might read incorrectly. Characters unrecognized by the encoding are replaced by the default substitution character for the specified encoding.

Encodings not listed here can be specified but might not produce correct results.

Return Values

List of strings, or one string

See Also

MuPAD Functions
`fprint` | `ftextinput` | `stringlib::subs`

stringlib::remove

Delete substrings

Syntax

```
stringlib::remove(string1, string2, <First>)
```

Description

With `stringlib::remove`, a substring can be deleted from another string.

After `string2` has been found, the search for further occurrences of it continues after its last letter; hence only the first of several overlapping occurrences is detected. See “Example 3” on page 31-34.

Examples

Example 1

By default, out of several occurrences of the given substring *all* are removed.

```
stringlib::remove("abcdeabcdeabcde", "bc")
```

```
"adeadeade"
```

Example 2

Using the option `First` causes `stringlib::remove` to remove only the first occurrence of the given substring.

```
stringlib::remove("abcdeabcdeabcde", "bc", First)
```

```
"adeabcdeabcde"
```

Example 3

In the following example, the given substring occurs twice, where both instances of it do overlap. Only the first occurrence is removed.

```
stringlib::remove("aaa", "aa")
```

```
"a"
```

Parameters

`string1`, `string2`

Non empty string

Options

`First`

Determines that only the first appearance of `string2` in `string1` will be deleted

Return Values

Given string without the deleted parts

See Also

MuPAD Functions

`delete` | `stringlib::subs` | `stringlib::subsop`

stringlib::split

Split a string

Syntax

```
stringlib::split(string, <separator>)
```

Description

`stringlib::split(string, separator)` splits `string` in all parts separated by the string given as `separator`, that is not included in the returned strings.

If no separator is given, a single space is used as separator.

A returned part can be the empty string.

When the given string does not contain the separator, a list with the unchanged string is returned.

Examples

Example 1

The given string is splitted into the numbers separated by comma:

```
stringlib::split("1,2,3,4,5", ",")
```

```
["1", "2", "3", "4", "5"]
```

In the next example is the separator a comma followed by a space:

```
stringlib::split("1, 2, 3, 4, 5", ", ")
```

```
["1", "2", "3", "4", "5"]
```

Without separator a single space is used as separator:

```
stringlib::split("1, 2, 3, 4, 5")
```

```
["1,", "2,", "3,", "4,", "5"]
```

Example 2

The parts can be empty strings – five empty strings separated by four single spaces:

```
stringlib::split("    ", " ")
```

```
["", "", "", "", ""]
```

The following string (five spaces) consists of two empty strings and a single space separated by two double spaces:

```
stringlib::split("     ", "  ")
```

```
["", "", " "]
```

Example 3

When the string does not contain the separator, a list with the unchanged string is returned:

```
stringlib::split("1,2,3,4,5", ".")
```

```
["1,2,3,4,5"]
```

Parameters

`string`, `separator`

Any non-empty MuPAD string

Return Values

List of all parts of `string` without all parts `separator`; the string itself, if `string` does not contain `separator`.

See Also

MuPAD Functions

`stringlib::collapseWhitespace` | `stringlib::contains` | `strmatch` | `substring`

stringlib::subs

Substitution in a string

Syntax

```
stringlib::subs(string, substring = replacement, <First>)
```

Description

`stringlib::subs` substitutes a substring by another string.

By default, every occurrence of the string `substring` in `string` is replaced by `replacement`. The option `First` causes only the first appearance of `substring` to be replaced.

The result is not searched again for instances of `substring`. See “Example 3” on page 31-39.

Among several overlapping occurrences of `substring`, the leftmost one is replaced.

Examples

Example 1

The string `replacement` may be empty.

```
stringlib::subs("abcdeabcdeabcde", "bc" = "")
```

```
"adeadeade"
```

Example 2

Every `substring` is replaced unless the option `First` is given.

```
stringlib::subs("abcdeabcdeabcde", "bc" = "xxx")
```

```
"axxxdeaxxxdeaxxxde"
```

```
stringlib::subs("abcdeabcdeabcde", "bc" = "xxx", First)
```

```
"axxxdeabcdeabcde"
```

Example 3

The substitution may produce a new instance of `substring`, but this one is not replaced.

```
stringlib::subs("aab", "ab"="b")
```

```
"ab"
```

Example 4

Collapse all whitespace in strings (see `stringlib::collapseWhitespace`):

```
f := proc(x) local y; begin y := 2*x; x + y end_proc:
string := expr2text(f)
```

```
"proc(x)   name f;   local y; begin   y := 2*x;    x + y end_proc"
```

```
string := stringlib::subs(string, "\n" = " "):
string := stringlib::subs(string, "  " = " "):
string := stringlib::subs(string, "  " = " ")
```

```
"proc(x) name f; local y; begin y := 2*x; x + y end_proc"
```

Parameters

`string`

Non empty string

substring

Non empty string that should be replaced

replacement

Any string that replaced `substring`

Options

First

Determines that only the first appearance of `substring` in `string` will be replaced

Return Values

Given string with `substring` replaced by `replacement`

See Also

MuPAD Functions

`stringlib::pos` | `stringlib::remove` | `stringlib::subsop` | `subs`

stringlib::subsop

Substitution in a string

Syntax

```
stringlib::subsop(string, index = replacement)
```

Description

`stringlib::subsop` removes one or more characters at a given position and inserts another substring at that position instead.

The char with index `index` in `string` (if `index` is an integer) or the range of chars (if `index` is a range of integers) is removed. Instead `replacement` is inserted at that position. The inserted string need not have the same length.

Examples

Example 1

Delete the first character:

```
stringlib::subsop("abcdeabcdeabcde", 1 = "")
```

```
"bcdeabcdeabcde"
```

The 2nd to 3rd character will be replaced by `"xxx"`:

```
stringlib::subsop("abcdeabcdeabcde", 2..3 = "xxx")
```

```
"axxxdeabcdeabcde"
```

Delete the characters 2 to 11:

```
stringlib::subsop("abcdeabcdeabcde", 2..11 = "")
```

```
"abcde"
```

Parameters

`string`

Non empty string

`index`

Integer or range of integers that determines the chars to be replaced

`replacement`

Any string to replace the given char or range

Return Values

Given string with the replacement

See Also

MuPAD Functions

`stringlib::pos` | `stringlib::remove` | `stringlib::subs` | `subsop`

stringlib::upper

Convert to upper-case

Syntax

```
stringlib::upper(string1)
```

Description

`stringlib::upper(string1)` converts each lower-case letter in the string `string1` to upper-case. All other characters remains unchanged.

If the string contains no lower-case letters, the given string is returned unchanged.

Examples

Example 1

Convert a string to upper-case:

```
stringlib::upper("MuPAD"),
stringlib::upper("Mupad"),
stringlib::upper("MUPAD"),
stringlib::upper("mupad")
```

```
"MUPAD", "MUPAD", "MUPAD", "MUPAD"
```

Example 2

Compare strings not case sensitive:

```
str_eq:= (str1, str2) ->
           bool(stringlib::upper(str1) = stringlib::upper(str2)):
str_eq("MuPAD", "mupad"), str_eq("Mupad", "MUPAD")
```

```
TRUE, TRUE
```

Parameters

`string1`

Any string

Return Values

String

See Also

MuPAD Functions

`stringlib::format` | `stringlib::lower`

stringlib::validIdent

Validate identifier name

Syntax

```
stringlib::validIdent(string)
```

Description

`stringlib::validIdent(string)` returns `TRUE`, when `string` is a valid identifier name, otherwise `FALSE`.

A valid identifier name in MuPAD must follow the rules:

- The first character must be a letter or the character `"_"`.
- All following characters must be letters or digits or the character `"_"`.
- An identifier consists of at least one character up to `512` characters.

Names in backticks ` are not determined as valid names.

Examples

Example 1

The example splits a set of names into valid identifier names and invalid identifier names:

```
split({"a", "1", "_111", "____", "A0b.C", "MuPAD", "1ABCDE", "xyz00"},
      stringlib::validIdent)
```

$$[\{\text{"MuPAD"}, \text{"\_111"}, \text{"\_\_\_\_"}, \text{"a"}, \text{"xyz00"}\}, \{\text{"1"}, \text{"1ABCDE"}, \text{"A0b.C"}\}, \varnothing]$$

`stringlib::random` called with option `Name` returns always valid identifier names. The function `map` applies `stringlib::validIdent` to each of the `1000` generated random names:

```
map({stringlib::random(1..10, Name) $ k = 1..1000},
    stringlib::validIdent)
```

```
{TRUE}
```

Parameters

`string`

A string

Return Values

TRUE or FALSE

See Also

MuPAD Functions

`domtype` | `stringlib::random`

Symbol – Typesetting Symbols

Symbol::accentPrime
Symbol::accentAsterisk
Symbol::accentTilde
Symbol::accentHat
Symbol::accentRightArrow
Symbol::accentDot
Symbol::accentDoubleDot
Symbol::accentTripleDot
Symbol::accentOverBar
Symbol::accentUnderBar
Symbol::new
Symbol::subScript
Symbol::subSuperScript
Symbol::superScript

Symbol::accentPrime

Adds a prime accent to an identifier

Syntax

```
Symbol::accentPrime(a)
```

Description

Creates a new identifier with a prime accent, such as a'.

Examples

Example 1

`Symbol::accentPrime` adds a “prime” to an identifier:

```
Symbol::accentPrime(x) = x + f(x)
```

$$x' = x + f(x)$$

Parameters

a

An identifier

Return Values

Identifier

Symbol::accentAsterisk

Adds an asterisk accent to an identifier

Syntax

```
Symbol::accentAsterisk(a)
```

Description

Creates a new identifier with an asterisk accent, such as a^*.

Examples

Example 1

Asterisk accents are often used to denote special values:

```
f(Symbol::accentAsterisk(x)) = 0
```

$$f(x^*) = 0$$

Parameters

a

An identifier

Return Values

Identifier

Symbol::accentTilde

Adds a tilde accent to an identifier

Syntax

```
Symbol::accentTilde(a)
```

Description

Creates a new identifier with a tilde accent, such as $\tilde{a}$.

Examples

Example 1

The most common use of the tilde accent is to denote an approximation:

```
x = Symbol::accentTilde(x) + O(h^2)
```

$$x = \tilde{x} + O(h^2)$$

Parameters

a

An identifier

Return Values

Identifier

Symbol::accentHat

Adds a hat accent to an identifier

Syntax

```
Symbol::accentHat(a)
```

Description

Creates a new identifier with a hat accent, such as $\hat{a}$.

Examples

Example 1

One of the common uses of the hat (or circumflex) accents is to denote Laplace transforms:

```
f = sin(x) ==> Symbol::accentHat(f)
 = laplace(sin(x), x, t)
```

$$f = \sin(x) \Rightarrow \hat{f} = \frac{1}{t^2 + 1}$$

Parameters

a

An identifier

Return Values

Identifier

Symbol::accentRightArrow

Adds a right arrow accent to an identifier

Syntax

```
Symbol::accentRightArrow(a)
```

Description

Creates a new identifier with a right arrow accent, such as $\vec{a}$.

Examples

Example 1

Arrow accents are usually used to denote vectors:

```
Symbol::accentRightArrow(b)
= matrix([1, 2, Symbol::dots, n])
```

$$\vec{b} = \begin{pmatrix} 1 \\ 2 \\ \vdots \\ n \end{pmatrix}$$

To denote $\vec{0}$, the null vector, start with the *identifier* 0:

```
Symbol::accentRightArrow(`0`)
```

$$\vec{0}$$

Parameters

a

An identifier

Return Values

Identifier

Symbol::accentDot

Adds a dot accent to an identifier

Syntax

```
Symbol::accentDot(a)
```

Description

Creates a new identifier with a dot accent, such as $\dot{a}$.

Examples

Example 1

In physics, a dot accent is often used as a shorthand for the derivative with respect to time:

```
Symbol::accentDot(y) =
- Symbol::omega^2 * sin(x)
```

$$\dot{y} = -\omega^2 \sin(x)$$

Parameters

a

An identifier

Return Values

Identifier

Symbol::accentDoubleDot

Adds a double dot accent to an identifier

Syntax

```
Symbol::accentDoubleDot(a)
```

Description

Creates a new identifier with a double dot accent, such as $a\ddot{}$.

Examples

Example 1

In physics, the double dot accent usually denotes the second derivative with respect to time:

```
Symbol::accentDoubleDot(x) = -x
```

$$\ddot{x} = -x$$

Parameters

a

An identifier

Return Values

Identifier

Symbol::accentTripleDot

Adds a triple dot accent to an identifier

Syntax

```
Symbol::accentTripleDot(a)
```

Description

Creates a new identifier with a triple dot accent, such as $\dddot{a}$.

Examples

Example 1

Triple dots, where used, usually denote the third derivative with repect to time:

```
Symbol::accentTripleDot(x)(t) = diff(x(t), t$3)
```

$$\dddot{x}(t) = \frac{\partial^3}{\partial t^3} x(t)$$

Parameters

a

An identifier

Return Values

Identifier

Symbol::accentOverBar

Adds an overbar to an identifier

Syntax

```
Symbol::accentOverBar(a)
```

Description

Creates a new identifier with an overbar, such as $\overline{a}$.

Examples

Example 1

The overbar is used in statistics to denote the arithmetical mean of an observable quantity:

```
Symbol::accentOverBar(x) = sum(x[i], i=1..n)/n
```

$$\overline{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

Parameters

a

An identifier

Return Values

Identifier

Symbol::accentUnderBar

Adds an underbar to an identifier

Syntax

```
Symbol::accentUnderBar(a)
```

Description

Creates a new identifier with an underbar, such as $\underline{a}$.

Examples

Example 1

One of the areas where the underbar is used frequently is interval analysis, where an interval is usually given as follows:

```
x = [Symbol::accentUnderBar(x),
     Symbol::accentOverBar(x)]
```

$$x = [\underline{x}, \overline{x}]$$

Parameters

a

An identifier

Return Values

Identifier

Symbol::new

Functional access to symbols

Syntax

```
Symbol(x)
```

Description

`Symbol::new(symname)` or simply `Symbol(symname)` creates the typesetting symbol corresponding to symname.

The typesetting symbols can be accessed in two different ways: Most symbols can be input by typing `Symbol::symname`, where symname is taken from the lists in the introduction. For some symbol names (such as `not` or `I`), this is not possible in the MuPAD language. What is possible in any case is to invoke `Symbol` as a function, taking a string representation as its argument, as in `Symbol("not")`.

Examples

Example 1

The symbols accessed via `Symbol` can be used like ordinary identifiers:

```
Symbol::heartsuit in
   {Symbol::heartsuit, Symbol::spades}
```

$$\heartsuit \in \{\heartsuit, \spadesuit\}$$

```
expand((Symbol::alpha + Symbol::beta)^4);
```

$$\alpha^4 + 4\,\alpha^3\,\beta + 6\,\alpha^2\,\beta^2 + 4\,\alpha\,\beta^3 + \beta^4$$

Some symbol names are MuPAD keywords and can not be accessed via slot calls. They can be given as function calls:

```
Symbol("minus"), Symbol("div"), Symbol("in"),
Symbol("and"), Symbol("subset"), Symbol("NIL"),
Symbol("UNKNOWN"), Symbol("FAIL"), Symbol("E"),
Symbol("I"), Symbol("not")
```

$$-,\ /,\ \in,\ \wedge,\ \subset,\ \mathrm{Nil},\ \mathrm{unknown},\ \mathrm{Fail},\ E,\ I,\ \neg$$

Parameters

`symname`

A symbol name: a string

Return Values

identifier

See Also

MuPAD Functions

`print`

Symbol::subScript

Combines two expressions to a new subscripted identifier

Syntax

```
Symbol::subScript(a, b)
```

Description

Creates a new subscripted identifier, such as a_b. If the arguments are not yet identifiers, they are first converted into identifiers.

You can also use _, ^, $, {, and } to create arguments with superscripts and subscripts. For these arguments a new superscripted identifier appears on top of the existing ones: a^{b^c} or $a^{b^d_c}$.

Examples

Example 1

Even if *X1* and *X2* look identical, only *X1* is an identifier whereas *X2* is an `_index`-expression:

```
X1 := Symbol::subScript(x, 1):  X2 := x[1]:
X1, X2;   domtype(X1), domtype(X2)
```

$$x_1, x_1$$

DOM_IDENT, DOM_EXPR

Pre-scripts are possible by subscripting the empty identifier `` ` `` and appending an identifer:

```
Symbol::subScript(``, 1).x
```

Example 2

You can use the nested form of the function:

```
Symbol::subScript(x, Symbol::subScript(i, j))
```

Example 3

Use `Symbol::subSuperScript` or `$` to create an expression with both sub- and superscript properly aligned one above the other:

```
Symbol::subSuperScript(a,b,c)
```

If you use the shortcut `$`, put the expression in single quotation marks:

```
`a$bc`
```

If you use `a_b^c`, it creates the subscripted expression `a_b` and then attaches the superscript `c` to that expression. In this case, the letters `b` and `c` do not appear one above the other.

```
a_b^c
```

The same happens when you use a nested call to `Symbol::subScript` and `Symbol::superScript`:

```
Symbol::superScript(Symbol::subScript(a, b), c)
```

Example 4

If you want to create identifiers in which the five special characters (_, ^, $, {, }) appear explicitly, use string arguments:

```
Symbol::subScript("a","b_c")
```

Parameters

`a, b`

Arbitrary expressions

Return Values

Identifier

See Also

MuPAD Functions

`Symbol::subSuperScript` | `Symbol::superScript`

Symbol::subSuperScript

Combines three expressions to a new combined sub- and superscripted identifier

Syntax

```
Symbol::subSuperScript(a, b, c)
```

Description

Creates a new combined sub- and superscripted identifier, such as a_b^c. If the arguments are not yet identifiers, they are first converted into identifiers.

You can also use _, ^, $, {, and } to create arguments with superscript and subscript. For these arguments a new indexed identifier appears on top of the existing ones: $a_{b_c}^{d_e}$ or $a_{b\,c}^{e\,d}$.

Examples

Example 1

Input of an identifier with sub- and superscript:

```
X1 := Symbol::subSuperScript(x, 1, 2):
X1, domtype(X1)
```

$$x_1^2,\ \text{DOM\_IDENT}$$

Pre-scripts are possible by subsuperscripting the empty identifier `` and appending an identifer:

```
Symbol::subSuperScript(``, 1, 2).X
```

$${}_1^2\mathrm{X}$$

And at last scripts all around `X`:

```
Symbol::subSuperScript(``, 1, 2).Symbol::subSuperScript(X, 3, 4)
```

Example 2

You can use the nested form of the function:

```
Symbol::subSuperScript(x, a, Symbol::subSuperScript(b, i, j))
```

Example 3

Use `Symbol::subSuperScript` or `$` to create an expression with both sub- and superscript properly aligned one above the other:

```
Symbol::subSuperScript(a,b,c)
```

If you use the shortcut `$`, put the expression in single quotation marks:

```
`a$bc`
```

If you use `a_b^c`, it creates the subscripted expression `a_b` and then attaches the superscript `c` to that expression. In this case, the letters `b` and `c` do not appear one above the other.

```
a_b^c
```

The same happens when you use a nested call to `Symbol::subScript` and `Symbol::superScript`:

```
Symbol::superScript(Symbol::subScript(a, b), c)
```

Example 4

If you want to create identifiers in which the five special characters (_, ^, $, {, }) appear explicitly, use string arguments:

```
Symbol::subSuperScript("a","b_c","d_e")
```

Parameters

`a, b, c`

Arbitrary expressions

Return Values

Identifier

See Also

MuPAD Functions

`Symbol::subScript` | `Symbol::superScript`

Symbol::superScript

Combines two expressions to a new superscripted identifier

Syntax

```
Symbol::superScript(a, b)
```

Description

Creates a new superscripted identifier, such as a^b. If the arguments are not yet identifiers, they are first converted into identifiers.

You can also use _, ^, $, {, and } to create arguments with superscripts and subscripts. For these arguments a new superscripted identifier appears on top of the existing ones: a^{b^c} or $a^{b^d_c}$.

Examples

Example 1

Even if *X1* and *X2* look identical, only *X1* is an identifier whereas *X2* is a `_power`-expression:

```
X1 := Symbol::superScript(x, 2):  X2 := x^2:
X1, X2;   domtype(X1), domtype(X2)
```

$$x^2, x^2$$

DOM_IDENT, DOM_EXPR

Pre-scripts are possible by superscripting the empty identifier `` and appending an identifer:

```
Symbol::superScript(``, 1).x
```

Example 2

You can use the nested form of the function:

```
Symbol::superScript(x, Symbol::superScript(i, j))
```

Example 3

Use `Symbol::subSuperScript` or `$` to create an expression with both sub- and superscript properly aligned one above the other:

```
Symbol::subSuperScript(a,b,c)
```

If you use the shortcut `$`, put the expression in single quotation marks:

```
`a$bc`
```

If you use `a_b^c`, it creates the subscripted expression `a_b` and then attaches the superscript `c` to that expression. In this case, the letters `b` and `c` do not appear one above the other.

```
a_b^c
```

The same happens when you use a nested call to `Symbol::subScript` and `Symbol::superScript`:

```
Symbol::superScript(Symbol::subScript(a, b), c)
```

Example 4

If you want to create identifiers in which the five special characters (_, ^, $, {, }) appear explicitly, use string arguments:

```
Symbol::superScript("a","b^c")
```

Parameters

`a, b`

Arbitrary expressions

Return Values

Identifier

See Also

MuPAD Functions

`Symbol::subScript` | `Symbol::subSuperScript`

Type – Type Checking and Mathematical Properties

Type::AlgebraicConstant
Type::AnyType
Type::Arithmetical
Type::Boolean
Type::Complex
Type::ConstantIdents
Type::Constant
Type::Equation
Type::Even
Type::Function
Type::Imaginary
Type::IndepOf
Type::Indeterminate
Type::Integer
Type::Intersection
Type::Interval
Type::ListOf
Type::ListProduct
Type::NegInt
Type::NegRat
Type::Negative
Type::NonNegInt
Type::NonNegRat
Type::NonNegative
Type::NonZero
Type::Numeric
Type::Odd
Type::PolyExpr
Type::PolyOf
Type::PosInt

Type::PosRat
Type::Positive
Type::Predicate
Type::Prime
Type::Product
Type::Property
Type::RatExpr
Type::Rational
Type::Real
Type::Relation
Type::Residue
Type::SequenceOf
Type::Series
Type::SetOf
Type::Set
Type::Singleton
Type::TableOfEntry
Type::TableOfIndex
Type::TableOf
Type::Union
Type::Unknown
Type::Zero

Type::AlgebraicConstant

Type representing algebraic constants

Syntax

```
testtype(obj, Type::AlgebraicConstant)
```

Description

`Type::AlgebraicConstant` represents algebraic constants.

In MuPAD, algebraic constants are characterized as follows: a complex number is an algebraic constant, if both its real part and its imaginary part are rational. Sums and products of algebraic constants are again algebraic constants. Further, rational powers of algebraic constants are again algebraic constants.

Taken together, these rules characterize algebraic constants over the rationals defined as usual, i.e., as roots of polynomial expressions.

This type does not represent a property: it cannot be used in `assume` to mark an identifier as an algebraic constant.

Examples

Example 1

The following number is composed of radicals involving rational numbers and therefore is an algebraic constant:

```
testtype((3^(1/2)*I + 1/8)^(1/7), Type::AlgebraicConstant)
```

```
TRUE
```

The following objects are not algebraic constants:

```
testtype(2^I, Type::AlgebraicConstant),
testtype(PI, Type::AlgebraicConstant)
```

FALSE, FALSE

Example 2

Symbolic objects cannot represent algebraic constants:

```
testtype(x, Type::AlgebraicConstant)
```

FALSE

Example 3

The following call selects the algebraic constants in an expression:

```
select(x + PI + 2^(1/2) + I, testtype, Type::AlgebraicConstant)
```

$\sqrt{2} + i$

Parameters

`obj`

Any MuPAD object

Return Values

See `testtype`

See Also

MuPAD Functions

`testtype` | `Type::Constant`

Type::AnyType

Type representing arbitrary MuPAD objects

Syntax

```
testtype(obj, Type::AnyType)
```

Description

`Type::AnyType` represents arbitrary MuPAD objects.

This type is meant to represent arbitrary MuPAD objects in constructors of composite types such as `Type::ListOf`.

This type does not represent a property: it cannot be used in `assume`.

Examples

Example 1

Any object matches this type:

```
testtype(3, Type::AnyType),
testtype(x, Type::AnyType),
testtype(array(1..1, [x]), Type::AnyType),
testtype(Dom::Matrix(), Type::AnyType)
```

```
TRUE, TRUE, TRUE, TRUE
```

This type is meant for constructing composite types. The following call tests, whether an object is a list with arbitrary elements:

```
testtype([3, x, array(1..1, [x]), Dom::Matrix()],
         Type::ListOf(Type::AnyType))
```

```
TRUE
```

Parameters

obj

Any MuPAD object

Return Values

`testtype` always returns TRUE

See Also

MuPAD Functions
`testtype`

Type::Arithmetical

Type representing arithmetical objects

Syntax

```
testtype(obj, Type::Arithmetical)
```

Description

`Type::Arithmetical` represents arithmetical objects.

In MuPAD, arithmetical objects are objects that represent complex numbers if all identifiers in them also represent complex numbers. Arithmetical objects include numbers, most expressions, and elements of certain library domains. In particular, the latter include `rectform` objects and series expansions of domain type `Series::Puiseux`.

Certain infinite objects such as `dirac(0)`, `infinity`, or `complexInfinity` are also defined to be arithmetical expressions.

The following objects are *not* regarded as arithmetical objects:

- equations and inequalities,
- Boolean objects and Boolean expressions involving `and`, `or`, `not`,
- lists,
- sets and set expressions involving `union`, `intersect`, `minus`,
- polynomials of domain type `DOM_POLY`,
- functions and procedures,
- arrays and tables.

This type does not represent a property: it cannot be used in `assume` to mark an identifier as an arithmetical object.

Examples

Example 1

The expression $\frac{BAB}{A}$ may represent a matrix if A and B are matrices, or it may represent a number if A and B are numbers. However, MuPAD regards identifiers as numbers: hence they commute with each other, and a product of identifiers represents a number, too:

```
A^(-1) * B * A * B
```

$$B^2$$

```
testtype(%, Type::Arithmetical)
```

TRUE

Example 2

Numbers and expressions are regarded as arithmetical objects:

```
testtype(3 + I, Type::Arithmetical),
testtype(x + sqrt(2) + I*PI, Type::Arithmetical),
testtype(x/y + y/x, Type::Arithmetical)
```

TRUE, TRUE, TRUE

Equations and inequalities are not regarded as arithmetical objects:

```
testtype(x^2 = 2, Type::Arithmetical),
testtype(x <> 2, Type::Arithmetical),
testtype(x < 2, Type::Arithmetical),
testtype(x >= 2, Type::Arithmetical)
```

FALSE, FALSE, FALSE, FALSE

Sets, lists, tables and arrays are not arithmetical:

```
testtype({a, b, c}, Type::Arithmetical),
testtype(array(1..1, [x]), Type::Arithmetical)
```

```
FALSE, FALSE
```

Most domain objects such as matrices of some matrix domain are not arithmetical:

```
testtype(Dom::Matrix()([[1, 2], [3, 4]]), Type::Arithmetical)
```

```
FALSE
```

Parameters

`obj`

Any MuPAD object

Return Values

See `testtype`

See Also

MuPAD Functions
`testtype`

Type::Boolean

Type representing boolean expressions

Syntax

```
testtype(obj, Type::Boolean)
```

Description

`Type::Boolean` represents logical formulas.

Boolean expressions are all of the following objects: the Boolean constants `TRUE`, `FALSE`, and `UNKNOWN`; identifiers; equations and inqualities; expressions with operator `and`, `or`, `not`, `xor`, `_implies`, `_equiv` if each operand is a Boolean expression; or results returned by `solvelib::isEmpty`.

Examples

Example 1

Identifiers and boolean constants are Boolean expressions:

```
testtype(TRUE, Type::Boolean), testtype(a, Type::Boolean)
```

```
TRUE, TRUE
```

Example 2

In order that an expression be Boolean, it is not sufficient that only its operator is a logical operator; also its operands must be Boolean expressions.

```
testtype(a >= 3 and b, Type::Boolean);
testtype(a+b and c, Type::Boolean)
```

```
TRUE

FALSE
```

Parameters

`obj`

Any MuPAD object

Return Values

See `testtype`

See Also

MuPAD Functions
`testtype`

Type::Complex

Type and property representing complex numbers

Syntax

```
testtype(obj, Type::Complex)
```

```
assume(x, Type::Complex)
```

```
is(ex, Type::Complex)
```

Description

`Type::Complex` represents complex numbers. This type can also be used as a property to mark identifiers as complex numbers.

The call `testtype(obj, Type::Complex)` checks, whether `obj` is a complex number and returns `TRUE`, if it holds, otherwise `FALSE`.

`testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT` and `DOM_COMPLEX`. This does not include arithmetical expressions such as `exp(1)`, which are not identified as of type `Type::Complex`.

The call `assume(x, Type::Complex)` marks the identifier `x` as a complex number.

The call `is(ex, Type::Complex)` derives, whether the expression `ex` is a complex number (or this property can be derived).

This type represents a property that can be used in `assume` and `is`.

Examples

Example 1

The following numbers are of type `Type::Complex`:

```
testtype(2, Type::Complex),
testtype(3/4, Type::Complex),
testtype(0.123, Type::Complex),
testtype(1 + I/3, Type::Complex),
testtype(1.0 + 2.0*I, Type::Complex)
```

```
TRUE, TRUE, TRUE, TRUE, TRUE
```

The following expressions are exact representations of complex numbers. Syntactically, however, they are not of type `Type::Complex`:

```
testtype(exp(3), Type::Complex),
testtype(PI^2 + 5, Type::Complex),
testtype(sin(2) + PI*I, Type::Complex)
```

```
FALSE, FALSE, FALSE
```

Example 2

Identifiers may be assumed to represent a complex number:

```
assume(x, Type::Complex): is(x, Type::Complex)
```

```
TRUE
```

The real numbers are a subset of the complex numbers:

```
assume(x, Type::Real): is(x, Type::Complex)
```

```
TRUE
```

Without further information, it cannot be decided whether a complex number is real:

```
assume(x, Type::Complex): is(x, Type::Real)
```

```
UNKNOWN
```

```
unassume(x):
```

Parameters

obj

Any MuPAD object

x

An identifier

ex

An arithmetical expression

Return Values

See `assume`, `is` and `testtype`

See Also

MuPAD Functions

`assume` | `is` | `testtype` | `Type::Imaginary` | `Type::Property` | `Type::Real`

Type::ConstantIdents

Set of constant identifiers in MuPAD

Syntax

```
contains(Type::ConstantIdents, obj)
```

Description

`Type::ConstantIdents` is the set `{ CATALAN , E , EULER , I , PI }`.

`Type::ConstantIdents` is the set of identifiers that represent constants. As of version 4.0, these are CATALAN, E (= `exp(1)`), EULER, I, PI.

These constants will be returned by the function `indets`, but they cannot be treated like other identifiers. For example, they cannot have properties or be the left-hand side of an assignment.

See “Example 1” on page 33-15 for an application.

`Type::Constant` makes use of `Type::ConstantIdents`, see “Example 2” on page 33-16.

Examples

Example 1

MuPAD implements π as the identifier PI.

```
domtype(PI)
```

```
DOM_IDENT
```

However, `PI` is constant:

```
testtype(PI, Type::Constant)
```

```
TRUE
```

Still, `indets` regards `PI` as an identifier with no value (which is syntactically correct), and you can even use `PI` as an indeterminate of a polynomial:

```
indets(PI/2*x);
poly(PI/2*x)
```

$$\{\pi, x\}$$

$$\mathrm{poly}\left(\frac{\pi\, x}{2}, [\pi, x]\right)$$

To find the "real" indeterminates, use the following call:

```
indets(PI/2*x) minus Type::ConstantIdents
```

$$\{x\}$$

Example 2

In the following, the `solve` command solves for all identifiers found in the equation:

```
solve(x^2 = KHINTCHINE)
```

$$\{[\mathrm{KHINTCHINE} = z^2, x = z]\}$$

Assume you want MuPAD to regard the identifier `KHINTCHINE` as a constant. (Probably, it should represent the Khintchine constant K, which is approximately 2.685452.) First of all, you should make sure that the identifier does not have a value yet and protect it:

```
testtype(KHINTCHINE, DOM_IDENT);
protect(KHINTCHINE, ProtectLevelError)
```

```
TRUE
```

```
ProtectLevelNone
```

Next, add `KHINTCHINE` to `Type::ConstantIdents` (note that we have to unprotect the identifier `Type`, because `Type::ConstantIdents` is a slot of it):

```
old_protection := unprotect(Type):
Type::ConstantIdents := Type::ConstantIdents union {KHINTCHINE}:
protect(Type, old_protection):
Type::ConstantIdents
```

$$\{i, \text{CATALAN}, \text{EULER}, \text{KHINTCHINE}, \pi, e\}$$

Now, MuPAD regards `KHINTCHINE` as a constant:

```
testtype(sin(PI + KHINTCHINE), Type::Constant)
```

$$\text{TRUE}$$

After clearing the remember table of `solve`, we now obtain:

```
solve(Remember, Clear):
solve(x^2 = KHINTCHINE)
```

$$\{[x = \sqrt{\text{KHINTCHINE}}], [x = -\sqrt{\text{KHINTCHINE}}]\}$$

Parameters

`obj`

Any MuPAD object

Return Values

See `contains`

See Also

MuPAD Functions

`contains` | `indets` | `Type::Constant` | `Type::Indeterminate`

Type::Constant

Type representing constant objects

Syntax

```
testtype(obj, Type::Constant)
```

Description

`Type::Constant` represents constant objects, i.e., objects not containing symbolic identifiers.

Numbers, strings, Boolean constants, `NIL`, `FAIL` and the identifiers PI, EULER and CATALAN in the set `Type::ConstantIdents` are regarded as constant objects. A composite object is constant, if all its operands are constant.

Any function is identified as a constant, if all arguments are constant, also if the function is not defined (e.g., an identifier).

This type does not represent a property: it cannot be used in `assume` to mark an identifier as a constant.

Examples

Example 1

The following objects are elementary constants:

```
testtype(3, Type::Constant),
testtype(sin(3/2), Type::Constant),
testtype(TRUE, Type::Constant),
testtype("MuPAD", Type::Constant),
testtype(FAIL, Type::Constant)
```

TRUE, TRUE, TRUE, TRUE, TRUE

The following expression contains an indeterminate x and, consequently, is not a constant object:

```
testtype(exp(x + 1), Type::Constant)
```

```
FALSE
```

All constant operands of an expression are selected:

```
select(x^2 + 3*x - 2, testtype, Type::Constant)
```

```
-2
```

Any function call is considered constant, if the arguments are constant:

```
testtype(f(1, 2, 3, 4), Type::Constant)
```

```
TRUE
```

Parameters

`obj`

Any MuPAD object

Return Values

See `testtype`

See Also

MuPAD Functions
`testtype`

Type::Equation

Type representing equations

Syntax

```
testtype(obj, Type::Equation(<lhs_type, <rhs_type>>))
```

Description

`Type::Equation` represents equations. The types of the left hand side and the right hand side can be specified.

The call `testtype(obj, Type::Equation(lhs_type,rhs_type))` checks whether `type(obj)` yields `" _equal "` and `testtype(lhs(obj), lhs_type)` and `testtype(rhs(obj), rhs_type)` both yield `TRUE` and returns `TRUE`, if all holds, otherwise `FALSE`.

The two optional parameters `lhs_type` and `rhs_type` determine the types of the left hand side and the right hand side, respectively.

The default values of `lhs_type` and `rhs_type` are `Type::AnyType`.

Note: The equations `lhs=rhs` and `rhs=lhs` are considered different! E.g., the equation `x=3` matches the type `Type::Equation(DOM_IDENT,DOM_INT)`, but it does not match the type `Type::Equation(DOM_INT,DOM_IDENT)`.

This type does not represent a property, it cannot be used in an `assume` call.

Examples

Example 1

The following object is an equation:

```
testtype(x = 3, Type::Equation())
```

```
TRUE
```

The following calls test, whether the object is an equation with an unknown on the left hand side and a positive integer on the right hand side:

```
testtype(x = 3, Type::Equation(Type::Unknown, Type::PosInt)),
testtype(x = 0, Type::Equation(Type::Unknown, Type::PosInt))
```

```
TRUE, FALSE
```

Parameters

`obj`

Any MuPAD object

`lhs_type`

The type of the left hand side; a type can be an object of the library `Type` or one of the possible return values of `domtype` and `type`

`rhs_type`

The type of the right hand side

Return Values

See `testtype`

See Also

MuPAD Functions
`testtype`

Type::Even

Type and property representing even integers

Syntax

```
testtype(obj, Type::Even)
```

```
assume(x, Type::Even)
```

```
is(ex, Type::Even)
```

Description

`Type::Even` represents even integers. This type can also be used as a property to mark identifiers as even integers.

The call `testtype(obj, Type::Even)` checks, whether `obj` is an even number and returns `TRUE`, if it holds, otherwise `FALSE`.

`testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT` and checks, if `bool(domtype(x/2) = DOM_INT)` holds.

The call `assume(x, Type::Even)` marks the identifier `x` as an even number.

The call `is(ex, Type::Even)` derives, whether the expression `ex` is an even number (or this property can be derived).

This type represents a property that can be used in `assume` and `is`.

Examples

Example 1

The following numbers are of type `Type::Even`:

```
testtype(2, Type::Even),
testtype(-4, Type::Even),
```

```
testtype(8, Type::Even),
testtype(-11114, Type::Even),
testtype(4185296581467695598, Type::Even)
```

```
TRUE, TRUE, TRUE, TRUE, TRUE
```

Example 2

We use this type as a property:

```
assume(x, Type::Even):
```

The following calls to `is` derive the properties of a composite expression from the properties of its indeterminates:

```
is(3*x^2, Type::Even), is(x + 1, Type::Even)
```

```
TRUE, FALSE
```

```
is(x, Type::Integer), is(2*x, Type::Integer),
is(x/2, Type::Integer), is(x/3, Type::Integer)
```

```
TRUE, TRUE, TRUE, UNKNOWN
```

```
assume(y, Type::Odd): is(x + y, Type::Even)
```

```
FALSE
```

```
is(2*(x + y), Type::Even)
```

```
TRUE
```

```
delete x, y:
```

Parameters

`obj`

Any MuPAD object

x

An identifier or a mathematical expression containing identifiers

ex

An arithmetical expression

Return Values

See `assume`, `is` and `testtype`

See Also

MuPAD Functions

`assume` | `is` | `testtype` | `Type::Odd` | `Type::Property`

Type::Function

Type representing functions

Syntax

```
testtype(obj, Type::Function)
```

Description

`Type::Function` represents all MuPAD functions (procedures, executable objects etc).

The call `testtype(obj, Type::Function)` checks, whether `obj` is an executable MuPAD object. The call returns `TRUE` or `FALSE`, respectively.

"Executable objects" in MuPAD are procedures (of type `DOM_PROC`), function environments (of type `DOM_FUNC_ENV`), and pure kernel functions (of type `DOM_EXEC`).

Additionally, symbolic function iterates`f@@n` (representing the map `x -> f(..(f(x))..)`) and symbolic function compositions`f@g` (representing the function `x -> f(g(x))` are regarded as executable objects.

This type does not represent a property.

Examples

Example 1

`Type::Function` accepts procedures:

```
testtype(proc(x) begin x^2 end, Type::Function)
```

```
    TRUE
```

`Type::Function` accepts simple procedures generated with the "arrow operator" `->`, too:

```
testtype(x -> x^2, Type::Function)
```

```
TRUE
```

`sin` is a function environment, accepted by `Type::Function`:

```
testtype(sin, Type::Function)
```

```
TRUE
```

The first operand of the function environment `print` is a pure kernel function, accepted by `Type::Function`:

```
testtype(op(print, 1), Type::Function)
```

```
TRUE
```

The 3-fold iterate of the function `diff` is accepted by `Type::Function`:

```
testtype(diff@@3, Type::Function)
```

```
TRUE
```

The composition of functions is accepted by `Type::Function`:

```
testtype(f@g, Type::Function)
```

```
TRUE
```

Any other MuPAD object is determined as non executable object by `Type::Function`:

```
map([1, TRUE, x, {}], testtype, Type::Function)
```

```
[FALSE, FALSE, FALSE, FALSE]
```

Parameters

`obj`

Any MuPAD object

Return Values

See `testtype`

See Also

MuPAD Functions
`testtype`

Type::Imaginary

Type and property representing imaginary numbers

Syntax

```
testtype(obj, Type::Imaginary)
```

```
assume(x, Type::Imaginary)
```

```
is(ex, Type::Imaginary)
```

Description

`Type::Imaginary` represents complex numbers with vanishing real part. This type can also be used as a property to mark identifiers as imaginary numbers.

The call `testtype(obj, Type::Imaginary)` checks, whether `obj` is an imaginary number (or zero) and returns `TRUE`, if it holds, otherwise `FALSE`.

`testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_COMPLEX` and checks, whether `iszero(Re(obj))` holds, or whether `iszero(obj)` is `TRUE`. This does not include arithmetical expressions such as `I*exp(1)`, which are not identified as of type `Type::Imaginary`.

The call `assume(x, Type::Imaginary)` marks the identifier `x` as an imaginary number.

The call `is(ex, Type::Imaginary)` derives, whether the expression `ex` is an imaginary number (or this property can be derived).

This type represents a property that can be used in `assume` and `is`.

The call `assume(Re(x) = 0)` has the same meaning as `assume(x, Type::Imaginary)`.

Examples

Example 1

The following numbers are of type `Type::Imaginary`:

```
testtype(5*I, Type::Imaginary),
testtype(3/2*I, Type::Imaginary),
testtype(-1.23*I, Type::Imaginary)
```

TRUE, TRUE, TRUE

The following expressions are exact representations of imaginary numbers. However, syntactically they are not of type `Type::Imaginary`, because their domain type is not `DOM_COMPLEX`:

```
testtype(exp(3)*I, Type::Imaginary),
testtype(PI*I, Type::Imaginary),
testtype(sin(2*I), Type::Imaginary)
```

FALSE, FALSE, FALSE

In contrast to `testtype`, the function `is` performs a semantical test:

```
is(exp(3)*I, Type::Imaginary),
is(PI*I, Type::Imaginary),
is(sin(2*I), Type::Imaginary)
```

TRUE, TRUE, TRUE

Example 2

Identifiers may be assumed to represent an imaginary number:

```
assume(x, Type::Imaginary): is(x, Type::Imaginary), Re(x), Im(x)
```

TRUE, 0, $-x\,\mathrm{i}$

The imaginary numbers are a subset of the complex numbers:

```
is(x, Type::Complex)

    TRUE

unassume(x):
```

Parameters

obj

Any MuPAD object

x

An identifier

ex

An arithmetical expression

obj

Any MuPAD object

Return Values

See `assume`, `is` and `testtype`

See Also

MuPAD Functions

`assume` | `is` | `testtype` | `Type::Complex` | `Type::Property`

Type::IndepOf

Type representing objects that do not contain given identifiers

Syntax

```
testtype(obj, Type::IndepOf(x))
```

```
testtype(obj, Type::IndepOf({x1, x2, …}))
```

Description

`Type::IndepOf(x)` represents objects that do not contain the identifier `x`.

`Type::IndepOf({x1, x2, ...})` represents objects that do not contain any of the identifiers `x1`, `x2` etc.

The call `testtype(obj, Type::IndepOf(x))` checks, whether `obj` does not contain the identifier `x` and returns a corresponding `TRUE` or `FALSE`.

`Type::IndepOf` uses `has` to check whether the object contains at least one of the specified identifiers.

This type does not represent a property.

Examples

Example 1

The following expression depends on `x`:

```
testtype(x^2 - x + 3, Type::IndepOf(x))
```

```
    FALSE
```

It is independend of `y`:

```
testtype(x^2 - x + 3, Type::IndepOf(y))
```

TRUE

The following expression is independend of `x` and `y`:

```
testtype(2*(a + b)/c, Type::IndepOf({x, y}))
```

TRUE

The following call selects all operands of the expression that are independend of `x`:

```
select(sin(y) + x^2 - 3*x + 2, testtype, Type::IndepOf(x))
```

$\sin(y) + 2$

Parameters

`obj`

Any MuPAD object

`x, x1, x2`

Identifiers of domain type `DOM_IDENT`

Return Values

See `testtype`

See Also

MuPAD Functions

`has` | `indets` | `testtype` | `Type::Indeterminate`

Type::Indeterminate

Type representing indeterminates

Syntax

```
testtype(obj, Type::Indeterminate)
```

Description

`Type::Indeterminate` represents all objects that MuPAD regards as indeterminates: identifiers except those in `Type::ConstantIdents`, plus indexed identifiers.

This type does not represent a property: it cannot be used in `assume` to mark an identifier as an algebraic constant.

Examples

Example 1

The following call selects all indeterminates from a list:

```
delete x:
testtype(x, Type::Indeterminate),
testtype(sqrt(2), Type::Indeterminate);
```

TRUE, FALSE

```
l := [x, x[2], x[sqrt(2) + I], x[x], PI, PI[1], sin(x), 3.0];
select(l, testtype, Type::Indeterminate)
```

$$\left[x,\ x_2,\ x_{\sqrt{2}+i},\ x_x,\ \pi,\ \pi_1,\ \sin(x),\ 3.0\right]$$

$$\left[x,\ x_2,\ x_{\sqrt{2}+i},\ x_x,\ \pi_1\right]$$

Note that `testtype` evaluates its arguments:

```
y := 5;
testtype(y, Type::Indeterminate),
testtype(hold(y), Type::Indeterminate);
```

```
    5


    FALSE, TRUE
```

Parameters

`obj`

Any MuPAD object

Return Values

See `testtype`

See Also

MuPAD Domains
`DOM_IDENT`

MuPAD Functions
`_index` | `indets` | `testtype` | `Type::ConstantIdents` | `Type::IndepOf`

Type::Integer

Type and property representing integers

Syntax

```
testtype(obj, Type::Integer)
```

```
assume(x, Type::Integer)
```

```
is(ex, Type::Integer)
```

Description

`Type::Integer` represents integers. This type can also be used as a property to mark identifiers as integers.

The call `testtype(obj, Type::Integer)` checks, whether `obj` is an integer number and returns `TRUE`, if it holds, otherwise `FALSE`.

`testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT`.

The call `assume(x, Type::Integer)` marks the identifier `x` as an integer number.

The call `is(ex, Type::Integer)` derives, whether the expression `ex` is an integer number (or this property can be derived).

This type represents a property that can be used in `assume` and `is`.

Examples

Example 1

The following numbers are of type `Type::Integer`:

```
testtype(0, Type::Integer), testtype(55, Type::Integer),
testtype(-111, Type::Integer)
```

```
TRUE, TRUE, TRUE
```

Example 2

We use this type as a property:

```
assume(x, Type::Integer):
```

The following calls to `is` derive the properties of a composite expression from the properties of its indeterminates:

```
is(3*x, Type::Real), is(2*x, Type::Even), is(x/2, Type::Integer)
```

```
TRUE, TRUE, UNKNOWN
```

```
assume(y, Type::Integer): is(x + y^2, Type::Integer)
```

```
TRUE
```

```
unassume(x), unassume(y):
```

Parameters

`obj`

Any MuPAD object

`x`

An identifier or a mathematical expression containing identifiers

`ex`

An arithmetical expression

Return Values

See `assume`, `is` and `testtype`

See Also

MuPAD Functions

`assume` | `is` | `testtype` | `Type::Property` | `Type::Real`

Type::Intersection

Type representing the intersection of several types

Syntax

```
testtype(obj, (obj_type, …))
```

Description

`Type::Intersection(type1, type2, ...)` represents all objects having all of the types `type1, type2, ...`

The call `testtype(obj, Type::Intersection(obj_types, ...))` checks, whether `obj` has all the given types obj_types, ...

The call `testtype(obj, Type::Intersection(obj_types, ...))` is thus equivalent to the call `_lazy_and(map(obj_types, x -> testtype(obj, x)))`, testing `obj` against all types in turn.

obj_types, ... must be a (nonempty) sequence of types (see `testtype`).

This type does not represent a property.

Examples

Example 1

Check, whether the given object is a positive and odd integer:

```
testtype(1, Type::Intersection(Type::PosInt, Type::Odd))
```

```
    TRUE
```

2 however, is not a positive and a odd number:

```
testtype(2, Type::Intersection(Type::PosInt, Type::Odd))
```

```
    FALSE
```

Example 2

`testtype` is used to select positive and odd integers:

```
SET:= {-2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2, 3}:
select(SET, testtype, Type::Intersection(Type::PosInt, Type::Odd))
```

```
    {1, 3}
```

```
delete SET:
```

Parameters

`obj`

Any MuPAD object

`obj_type, …`

A sequence of types; a type can be an object of the library Type or one of the possible return values of `domtype` and `type`

Return Values

See `testtype`

See Also

MuPAD Functions
`testtype`

Type::Interval

Property representing intervals

Syntax

```
Type::Interval(a, b, <ndomain>)
Type::Interval([a], b, <ndomain>)
Type::Interval(a, [b], <ndomain>)
Type::Interval([a], [b], <ndomain>)
Type::Interval([a, b], <ndomain>)
```

Description

`Type::Interval(a, b, ...)` represents the interval (a, b).

`Type::Interval([a], b, ...)` represents the interval $[a, b)$.

`Type::Interval(a, [b], ...)` represents the interval $(a, b]$.

`Type::Interval([a], [b], ...)` represents the interval $[a, b]$.

`Type::Interval([a, b], ...)` represents the interval $[a, b]$.

With the default domain `Type::Real`, the type object created by `Type::Interval` represents a real interval, i.e., the set of all real numbers between the border points `a` and `b`. If another domain is specified, then the type object represents the intersection of the real interval with the set represented by the domain. E.g., `Type::Interval(a, b, Type::Rational)` represents the set of all rational numbers between `a` and `b`, and `Type::Interval([a, b], Type::Residue(0, 2))` represents the set of all even integers between `a` and `b` including `a` and `b`.

The type object represents a property that may be used in `assume` and `is`. With

```
assume(x, Type::Interval(a, b, ndomain))
```

the identifier `x` is marked as a number from the interval represented by the type object. With

```
is(x, Type::Interval(a, b, ndomain))
```

one queries, whether `x` is contained in the interval.

Interval types should not be used in `testtype`. No MuPAD object matches these types syntactically, i.e., `testtype` always returns `FALSE`.

Examples

Example 1

The following type object represents the open interval (- 1, 1):

```
Type::Interval(-1, 1)
```

$$\text{Type::Interval}(-1, 1, \text{Type::Real})$$

The following calls are equivalent: both create the type representing a closed interval:

```
Type::Interval([-1], [1]), Type::Interval([-1, 1])
```

$$\text{Type::Interval}([-1], [1], \text{Type::Real}), \text{Type::Interval}([-1], [1], \text{Type::Real})$$

The following call creates the type representing the set of all integers from -10 to 10:

```
Type::Interval([-10, 10], Type::Integer)
```

$$\text{Type::Interval}([-10], [10], \text{Type::Integer})$$

The following call creates the type representing the set of all rational numbers in the interval $[0, 1)$:

```
Type::Interval([0], 1, Type::Rational)
```

$$\text{Type::Interval}([0], 1, \text{Type::Rational})$$

The following calls create the types representing the sets of all even/odd integers in the interval $[-10, 10]$:

```
Type::Interval([-10], [10], Type::Even),
Type::Interval([-10], [10], Type::Odd)
```

$$\text{Type::Interval}([-10], [10], \text{Type::Residue}(0, 2, \text{Type::Integer})),\ \text{Type::Interval}([-10], [10], \text{Type::Residue}(1, 2, \text{Type::Integer}))$$

Example 2

We use intervals as a property. The following call marks x as a real number from the interval $[0, 2)$:

```
assume(x, Type::Interval([0], 2)):
```

Consequently, $x^2 + 1$ lies in the interval $[1, 5)$:

```
is(x^2 + 1 >= 1), is(x^2 + 1 < 5)
```

$$\text{TRUE}, \text{TRUE}$$

The following call marks x as an integer larger than -10 and smaller than 100:

```
assume(x, Type::Interval(-10, 100, Type::Integer)):
```

Consequently, x^3 is an integer larger than -730 and smaller than 970300:

```
is(x^3, Type::Integer), is(x^3 >= -729), is(x^3 < 970300),
is(x^3, Type::Interval(-10^3, 100^3, Type::Integer))
```

$$\text{TRUE}, \text{TRUE}, \text{TRUE}, \text{TRUE}$$

```
is(x <= -730), is(x^3 >= 970300)
```

$$\text{FALSE}, \text{FALSE}$$

```
is(x > 0), is(x^3, Type::Interval(0, 10, Type::Integer))
```

```
UNKNOWN, UNKNOWN
```

```
unassume(x):
```

Parameters

a, b

The borders of the interval: arithmetical objects

ndomain

A type object such as `Type::Real`, `Type::Integer` or `Type::Rational` representing a subset of the real numbers or a property representing a residue class as `Type::Residue(0, 2)`. The default domain is `Type::Real`.

Return Values

`Type` object

See Also

MuPAD Functions

`assume` | `is` | `testtype` | `Type::Integer` | `Type::Rational` | `Type::Real` | `Type::Residue`

Type::ListOf

Type representing lists of objects with the same type

Syntax

```
testtype(obj, (obj_type, <min_nr, <max_nr>>))
```

Description

`Type::ListOf` describes lists of objects of a specified type.

The call `testtype(obj, Type::ListOf(obj_types, ...))` checks, whether `obj` is a list with elements of the given type obj_type, ... and returns `TRUE`, if it holds, otherwise `FALSE`.

The two optional parameters `min_nr` and `max_nr` determine the minimum and maximum number of elements in the analyzed list. If the numbers are not be given, the number of elements in the list will not be checked. If only the minimum is given, only the minimal number of elements in the list is checked.

Note especially that `Type::Union` provides a way to allow more than one type for the list elements.

This type does not represent a property.

Examples

Example 1

Is the given list a list of identifiers?

```
testtype([a, b, c, d, e, f], Type::ListOf(DOM_IDENT))
```

```
    TRUE
```

Is the given list a list of at least five real numbers?

```
testtype([0, 0.5, 1, 1.5, 2, 2.5, 3], Type::ListOf(Type::Real, 5))
```

```
TRUE
```

Example 2

`testtype` is used to select lists with exactly two identifiers:

```
S := {[a], [a, b], [d, 1], [0, d], [e], [d, e]}:
select(S, testtype, Type::ListOf(DOM_IDENT, 2, 2))
```

$$\{[a, b], [d, e]\}$$

Parameters

`obj`

Any MuPAD object

`obj_type`

The type of the objects; a type can be an object of the library Type or one of the possible return values of `domtype` and `type`

`min_nr`

The minimal number of objects as nonnegative integer

`max_nr`

The maximal number of objects as nonnegative integer

Return Values

See `testtype`

See Also

MuPAD Domains
DOM_LIST

MuPAD Functions
testtype | Type::ListProduct | Type::SetOf | Type::Union

Type::ListProduct

Type representing lists

Syntax

```
testtype(obj, Type::ListProduct(typedef, …))
```

Description

With `Type::ListProduct`, lists with different object types can be identified.

The call `testtype(obj, Type::ListProduct(typedef))` checks, whether `obj` is a list of objects, which have the types given by `typedef` and returns `TRUE`, if it holds, otherwise `FALSE`.

`obj` must have the same number of arguments as the sequence `typedef`. The elements of `obj` are checked one after another: the first element of `obj` is checked against the type given by the first element of `typedef`, and so on. All elements and types must match.

typedef, ... must be a nonempty sequence of types. A type can be an object of the library Type or one of the possible return values of `domtype` and `type`.

This type does not represent a property.

Examples

Example 1

The argument is a list of a positive integer followed by an identifier:

```
testtype([5, x], Type::ListProduct(Type::PosInt, Type::Unknown))
```

```
    TRUE
```

Is the argument is a sequence of five positive integers?

```
testtype([5, 3, 5, -1, 0], Type::ListProduct(Type::PosInt $ 5))
```

```
    FALSE
```

Parameters

`obj`

Any MuPAD object

`typedef`

A sequence of types; a type can be an object of the library Type or one of the possible return values of `domtype` and `type`

Return Values

See `testtype`

See Also

MuPAD Functions

`testtype` | `Type::ListOf` | `Type::Product`

Type::NegInt

Type and property representing negative integers

Syntax

```
testtype(obj, Type::NegInt)
```

```
assume(x, Type::NegInt)
```

```
is(ex, Type::NegInt)
```

Description

`Type::NegInt` represents negative integers. `Type::NegInt` is a property, too, which can be used in an `assume` call.

The call `testtype(obj, Type::NegInt)` checks, whether `obj` is a negative integer number and returns `TRUE`, if it holds, otherwise `FALSE`.

`testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT` and checks, if `bool(obj < 0)` holds.

The call `assume(x, Type::NegInt)` marks the identifier `x` as a negative integer number.

The call `is(ex, Type::NegInt)` derives, whether the expression `ex` is a negative integer number (or this property can be derived).

This type represents a property that can be used in `assume` and `is`.

Examples

Example 1

The following numbers are of type `Type::NegInt`:

```
testtype(-2, Type::NegInt),
testtype(-3, Type::NegInt),
testtype(-55, Type::NegInt),
testtype(-1, Type::NegInt),
testtype(-111111111, Type::NegInt)
```

TRUE, TRUE, TRUE, TRUE, TRUE

Example 2

Assume an identifier is a negative integer:

```
assume(x, Type::NegInt):
is(x, Type::NegInt)
```

TRUE

Negative integers are integers, of course:

```
assume(x, Type::NegInt):
is(x, Type::Integer)
```

TRUE

However, integers can be negative or not:

```
assume(x, Type::Integer):
is(x, Type::NegInt)
```

UNKNOWN

```
delete x:
```

Parameters

obj

Any MuPAD object

x

An identifier or a mathematical expression containing identifiers

ex

An arithmetical expression

Return Values

See `testtype`, `assume` and `is`

See Also

MuPAD Functions
`assume` | `is` | `testtype` | `Type::Property`

Type::NegRat

Type and property representing negative rational numbers

Syntax

```
testtype(obj, Type::NegRat)
```

```
assume(x, Type::NegRat)
```

```
is(ex, Type::NegRat)
```

Description

`Type::NegRat` represents negative rational numbers. `Type::NegRat` is a property, too, which can be used in an `assume` call.

The call `testtype(obj, Type::NegRat)` checks, whether `obj` is a negative rational number and returns `TRUE`, if it holds, otherwise `FALSE`.

`testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT` and `DOM_RAT` and checks, if `bool(obj < 0)` holds.

The call `assume(x, Type::NegRat)` marks the identifier `x` as a negative rational number.

The call `is(ex, Type::NegRat)` derives, whether the expression `ex` is a negative rational number (or this property can be derived).

This type represents a property that can be used in `assume` and `is`.

Examples

Example 1

The following numbers are of type `Type::NegRat`:

```
testtype(-2, Type::NegRat),
testtype(-3/4, Type::NegRat),
testtype(-55/111, Type::NegRat),
testtype(-1, Type::NegRat),
testtype(-111/111111, Type::NegRat)
```

```
TRUE, TRUE, TRUE, TRUE, TRUE
```

Example 2

Assume an identifier is negative rational:

```
assume(x, Type::NegRat):
is(x, Type::NegRat)
```

```
TRUE
```

Also negative rational numbers are rational:

```
assume(x, Type::NegRat):
is(x, Type::Rational)
```

```
TRUE
```

However, rational numbers can be negative rational or not:

```
assume(x, Type::Rational):
is(x, Type::NegRat)
```

```
UNKNOWN
```

```
delete x:
```

Parameters

`obj`

Any MuPAD object

x

An identifier or a mathematical expression containing identifiers

ex

An arithmetical expression

Return Values

See `testtype`, `assume` and `is`

See Also

MuPAD Functions
`assume` | `is` | `testtype` | `Type::Property`

Type::Negative

Type and property representing negative numbers

Syntax

```
testtype(obj, Type::Negative)
```

```
assume(x, Type::Negative)
```

```
is(ex, Type::Negative)
```

Description

`Type::Negative` represents negative numbers. `Type::Negative` is a property, too, which can be used in an `assume` call.

The call `testtype(obj, Type::Negative)` checks, whether `obj` is a negative real number and returns `TRUE`, if it holds, otherwise `FALSE`.

`testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT`, `DOM_RAT` and `DOM_FLOAT` and checks, if `bool(obj < 0)` holds. This does not include arithmetical expressions such as `-exp(1)`, which are not identified as of type `Type::Negative`.

The call `assume(x, Type::Negative)` marks the identifier `x` as a negative real number.

The call `is(ex, Type::Negative)` derives, whether the expression `ex` is a negative real number (or this property can be derived).

This type represents a property that can be used in `assume` and `is`.

Instead of `Type::Negative` the assumption can also be `assume(x < 0)`.

Examples

Example 1

The following numbers are of type `Type::Negative`:

```
testtype(-2, Type::Negative),
testtype(-3/4, Type::Negative),
testtype(-0.123, Type::Negative),
testtype(-1, Type::Negative),
testtype(-1.02, Type::Negative)
```

```
TRUE, TRUE, TRUE, TRUE, TRUE
```

The following expressions are exact representations of negative numbers, but syntactically they are not of `Type::Negative`:

```
testtype(-exp(1), Type::Negative),
testtype(-PI^2 - 5, Type::Negative),
testtype(-sin(2), Type::Negative)
```

```
FALSE, FALSE, FALSE
```

Example 2

Assume an identifier is negative:

```
assume(x, Type::Negative):
is(x, Type::Negative)
```

```
TRUE
```

This is equal to:

```
assume(x < 0):
is(x < 0)
```

```
TRUE
```

Also negative numbers are real:

```
assume(x, Type::Negative):
is(x, Type::Real)
```

```
    TRUE
```

However, real numbers can be negative or not:

```
assume(x, Type::Real):
is(x, Type::Negative)
```

```
    UNKNOWN
```

```
delete x:
```

Parameters

`obj`

Any MuPAD object

`x`

An identifier or a mathematical expression containing identifiers

`ex`

An arithmetical expression

Return Values

See `testtype`, `assume` and `is`

See Also

MuPAD Functions

`assume` | `is` | `testtype` | `Type::Property` | `Type::Real`

Type::NonNegInt

Type and property representing nonnegative integers

Syntax

```
testtype(obj, Type::NonNegInt)
```

```
assume(x, Type::NonNegInt)
```

```
is(ex, Type::NonNegInt)
```

Description

`Type::NonNegInt` represents nonnegative integers. `Type::NonNegInt` is a property, too, which can be used in an `assume` call.

The call `testtype(obj, Type::NonNegInt)` checks, whether `obj` is a nonnegative integer number and returns `TRUE`, if it holds, otherwise `FALSE`.

`testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT` and checks, if `bool(obj >= 0)` holds.

The call `assume(x, Type::NonNegInt)` marks the identifier `x` as a nonnegative integer number.

The call `is(ex, Type::NonNegInt)` derives, whether the expression `ex` is a nonnegative integer number (or this property can be derived).

This type represents a property that can be used in `assume` and `is`.

Examples

Example 1

The following numbers are of type `Type::NonNegInt`:

```
testtype(2, Type::NonNegInt),
testtype(3/4, Type::NonNegInt),
testtype(55/111, Type::NonNegInt),
testtype(1, Type::NonNegInt),
testtype(111/111111, Type::NonNegInt)
```

```
TRUE, FALSE, FALSE, TRUE, FALSE
```

Example 2

Assume an identifier is nonnegative rational:

```
assume(x, Type::NonNegInt):
is(x, Type::NonNegInt)
```

```
TRUE
```

Also nonnegative integers are integers:

```
assume(x, Type::NonNegInt):
is(x, Type::Integer)
```

```
TRUE
```

However, integers can be nonnegative or not:

```
assume(x, Type::Integer):
is(x, Type::NonNegInt)
```

```
UNKNOWN
```

```
delete x:
```

Parameters

`obj`

Any MuPAD object

x

An identifier or a mathematical expression containing identifiers

ex

An arithmetical expression

Return Values

See `testtype`, `assume` and `is`

See Also

MuPAD Functions
`assume` | `is` | `testtype` | `Type::Integer` | `Type::Property`

Type::NonNegRat

Type and property representing nonnegative rational numbers

Syntax

```
testtype(obj, Type::NonNegRat)
```

```
assume(x, Type::NonNegRat)
```

```
is(ex, Type::NonNegRat)
```

Description

`Type::NonNegRat` represents nonnegative rational numbers. `Type::NonNegRat` is a property, too, which can be used in an `assume` call.

The call `testtype(obj, Type::NonNegRat)` checks, whether `obj` is a nonnegative rational number and returns `TRUE`, if it holds, otherwise `FALSE`.

`testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT` and `DOM_RAT` and checks, if `bool(obj >= 0)` holds.

The call `assume(x, Type::NonNegRat)` marks the identifier `x` as a nonnegative rational number.

The call `is(ex, Type::NonNegRat)` derives, whether the expression `ex` is a nonnegative rational number (or this property can be derived).

This type represents a property that can be used in `assume` and `is`.

Examples

Example 1

The following numbers are of type `Type::NonNegRat`:

```
testtype(2, Type::NonNegRat),
testtype(3/4, Type::NonNegRat),
testtype(55/111, Type::NonNegRat),
testtype(0, Type::NonNegRat),
testtype(111/111111, Type::NonNegRat)
```

```
TRUE, TRUE, TRUE, TRUE, TRUE
```

Example 2

Assume an identifier is nonnegative rational:

```
assume(x, Type::NonNegRat):
is(x, Type::NonNegRat)
```

```
TRUE
```

Also nonnegative rational numbers are rational:

```
assume(x, Type::NonNegRat):
is(x, Type::Rational)
```

```
TRUE
```

However, rational numbers can be nonnegative rational or not:

```
assume(x, Type::Rational):
is(x, Type::NonNegRat)
```

```
UNKNOWN
```

```
delete x:
```

Parameters

`obj`

Any MuPAD object

x

An identifier or a mathematical expression containing identifiers

ex

An arithmetical expression

Return Values

See `testtype`, `assume` and `is`

See Also

MuPAD Functions

`assume` | `is` | `testtype` | `Type::Property` | `Type::Rational`

Type::NonNegative

Type and property representing nonnegative numbers

Syntax

```
testtype(obj, Type::NonNegative)
```

```
assume(x, Type::NonNegative)
```

```
is(ex, Type::NonNegative)
```

Description

`Type::NonNegative` represents nonnegative numbers. `Type::NonNegative` is a property, too, which can be used in an `assume` call.

The call `testtype(obj, Type::NonNegative)` checks, whether `obj` is a nonnegative real number and returns `TRUE`, if it holds, otherwise `FALSE`.

`testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT`, `DOM_RAT` and `DOM_FLOAT` and checks, if `bool(obj >= 0)` holds. This does not include arithmetical expressions such as `exp(1)`, which are not identified as of type `Type::NonNegative`.

The call `assume(x, Type::NonNegative)` marks the identifier `x` as a nonnegative real number.

The call `is(ex, Type::NonNegative)` derives, whether the expression `ex` is a nonnegative real number (or this property can be derived).

This type represents a property that can be used in `assume` and `is`.

Instead of `Type::NonNegative` the assumption can also be `assume(x >= 0)`.

Examples

Example 1

The following numbers are of type `Type::NonNegative`:

```
testtype(2, Type::NonNegative),
testtype(3/4, Type::NonNegative),
testtype(0.123, Type::NonNegative),
testtype(0, Type::NonNegative),
testtype(1.02, Type::NonNegative)
```

```
TRUE, TRUE, TRUE, TRUE, TRUE
```

The following expressions are exact representations of nonnegative numbers, but syntactically they are not of `Type::NonNegative`:

```
testtype(exp(1), Type::NonNegative),
testtype(PI^2 + 5, Type::NonNegative),
testtype(sin(2), Type::NonNegative)
```

```
FALSE, FALSE, FALSE
```

The function `is`, however, can find these expressions to be nonnegative:

```
is(exp(1), Type::NonNegative),
is(PI^2 + 5, Type::NonNegative),
is(sin(2), Type::NonNegative)
```

```
TRUE, TRUE, TRUE
```

Example 2

Assume an identifier is nonnegative:

```
assume(x, Type::NonNegative):
is(x, Type::NonNegative)
```

```
TRUE
```

This is equal to:

```
assume(x >= 0):
is(x >= 0)
```

```
    TRUE
```

Also nonnegative numbers are real:

```
assume(x, Type::NonNegative):
is(x, Type::Real)
```

```
    TRUE
```

But real numbers can be nonnegative or not:

```
assume(x, Type::Real):
is(x, Type::NonNegative)
```

```
    UNKNOWN
```

```
delete x:
```

Parameters

`obj`

Any MuPAD object

`x`

An identifier or a mathematical expression containing identifiers

`ex`

An arithmetical expression

Return Values

See `testtype`, `assume` and `is`

See Also

MuPAD Functions

`assume` | `is` | `testtype` | `Type::Property` | `Type::Real`

Type::NonZero

Type and property representing “unequal to zero”

Syntax

```
testtype(obj, Type::NonZero)
```

```
assume(x, Type::NonZero)
```

```
is(ex, Type::NonZero)
```

Description

`Type::NonZero` is a type of objects unequal to zero. `Type::NonZero` is a property, too, which can be used in an `assume` call.

The call `testtype(obj, Type::NonZero)` checks, whether `obj` is *not* zero and returns `TRUE`, if it holds, otherwise `FALSE`.

`testtype` only performs a syntactical test and uses the function `iszero` to determine, whether the object is not zero. This implies that identifiers without a value, for example, are considered as being different from zero, see “Example 1” on page 33-68.

The call `assume(x, Type::NonZero)` marks the identifier `x` as a complex number unequal to zero.

The call `is(ex, Type::NonZero)` derives, whether the expression `ex` is a complex number unequal to zero (or this property can be derived).

This type represents a property that can be used in `assume` and `is`.

The call `assume(x <> 0)` has the same meaning as `assume(x, Type::NonZero)`.

Examples

Example 1

Usage of `Type::NonZero` whith `testtype`:

```
testtype(1.0, Type::NonZero)
```

```
    TRUE
```

Since `iszero(x)` returns `FALSE`, the following call returns `TRUE`:

```
testtype(x, Type::NonZero)
```

```
    TRUE
```

Example 2

Usage of `Type::NonZero` whith `assume` and `is`:

```
is(x, Type::NonZero)
```

```
    UNKNOWN
```

Assumption: `x` is `Type::NonZero`:

```
assume(x, Type::NonZero):
is(x, Type::NonZero)
```

```
    TRUE
```

The same again:

```
assume(x <> 0):
is(x <> 0)
```

```
    TRUE
```

The difference between `testtype` and `is`:

```
delete x:
is(x, Type::NonZero), testtype(x, Type::NonZero)
```

```
    UNKNOWN, TRUE
```

x could be zero:

```
assume(x >= 0):
is(x, Type::NonZero), testtype(x, Type::NonZero)
```

```
UNKNOWN, TRUE
```

```
delete x:
```

Parameters

obj

Any MuPAD object

x

An identifier or a mathematical expression containing identifiers

ex

An arithmetical expression

Return Values

See `testtype`, `assume` and `is`

See Also

MuPAD Functions

`assume` | `is` | `testtype` | `Type::Zero`

Type::Numeric

Type representing numerical objects

Syntax

```
testtype(obj, Type::Numeric)
```

Description

With `Type::Numeric`, numeric objects (numbers) can be identified.

The call `testtype(obj, Type::Numeric)` checks, whether `obj` is a number and returns `TRUE`, if it holds, otherwise `FALSE`.

A number has the domain type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT` or `DOM_COMPLEX`.

This type does not represent a property.

Examples

Example 1

The following objects are numbers.

```
testtype(2, Type::Numeric),
testtype(3/4, Type::Numeric),
testtype(0.123, Type::Numeric),
testtype(1 + I/3, Type::Numeric),
testtype(1.0 + 2.0*I, Type::Numeric)
```

```
TRUE, TRUE, TRUE, TRUE, TRUE
```

The following objects are not numerical objects.

```
testtype(ln(2), Type::Numeric),
```

```
testtype(sin(3/4), Type::Numeric),
testtype(x + I/3, Type::Numeric)
```

```
FALSE, FALSE, FALSE
```

Parameters

obj

Any MuPAD object

Return Values

See `testtype`

See Also

MuPAD Functions

`assume` | `is` | `testtype` | `Type::Complex`

Type::Odd

Type and property representing odd integers

Syntax

```
testtype(obj, Type::Odd)
```

```
assume(x, Type::Odd)
```

```
is(ex, Type::Odd)
```

Description

`Type::Odd` represents odd integers. `Type::Odd` is a property, too, which can be used in an `assume` call.

The call `testtype(obj, Type::Odd)` checks, whether `obj` is an odd number and returns `TRUE`, if it holds, otherwise `FALSE`.

`testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT` and checks, if `bool(domtype((x-1)/2) = DOM_INT)` holds.

The call `assume(x, Type::Odd)` marks the identifier `x` as an odd number.

The call `is(ex, Type::Odd)` derives, whether the expression `ex` is an odd number (or this property can be derived).

This type represents a property that can be used in `assume` and `is`.

Examples

Example 1

The following numbers are of type `Type::Odd`:

```
testtype(1, Type::Odd),
```

```
testtype(-3, Type::Odd),
testtype(7, Type::Odd),
testtype(-11113, Type::Odd),
testtype(4185296581467695597, Type::Odd)
```

```
TRUE, TRUE, TRUE, TRUE, TRUE
```

Example 2

Assume an identifier is odd:

```
assume(x, Type::Odd):
is(x, Type::Odd)
```

```
TRUE
```

All odd numbers are integer:

```
assume(x, Type::Odd):
is(x, Type::Integer)
```

```
TRUE
```

However, integers can be odd or not:

```
assume(x, Type::Integer):
is(x, Type::Odd)
```

```
UNKNOWN
```

However, even numbers are not odd:

```
assume(x, Type::Odd):
is(2*x, Type::Odd)
```

```
FALSE
```

```
assume(n, Type::Even):
is(x*n, Type::Odd)
```

```
    FALSE
```

```
is(x*n + 1, Type::Odd)
```

```
    TRUE
```

```
delete x, n:
```

Parameters

`obj`

Any MuPAD object

`x`

An identifier or a mathematical expression containing identifiers

`ex`

An arithmetical expression

Return Values

See `testtype`, `assume` and `is`

See Also

MuPAD Functions

`assume` | `is` | `testtype` | `Type::Even` | `Type::Property`

Type::PolyExpr

Type representing polynomial expressions

Syntax

```
testtype(obj, Type::PolyExpr(unknowns, <coeff_type>))
```

Description

With `Type::PolyExpr`, polynomial expressions can be identified.

The call `testtype(obj, Type::PolyExpr(unknowns))` checks, whether `obj` is a polynomial expression in the indeterminates `unknowns` and, if so, returns `TRUE`, otherwise `FALSE`.

A polynomial expression in `indet` is an expression, where `indet` occurs only as operand of `_plus` or `_mult` expressions and in the base of `_power` with a positive integer exponent.

A polynomial expression is a representation of a polynomial, but it has the MuPAD type `DOM_EXPR` and is not produced by the function `poly`.

`indets` must be an identifier or a list of identifiers.

The optional argument `coeff_type` determines the type of the coefficients. If it is not given, `Type::AnyType` will be used.

This type does not represent a property.

Examples

Example 1

Is the object a polynomial expression with variable `x`?

```
X := -x^2 - x + 3:
testtype(X, Type::PolyExpr(x))
```

```
    TRUE
```

But `X` is not a MuPAD polynomial in `x`:

```
testtype(X, Type::PolyOf(x))
```

```
    FALSE
```

Is the object a polynomial expression with variables `x` and `y` and with integer coefficients?

```
X := -x^2 - x + 3:
testtype(X, Type::PolyExpr([x, y], Type::Integer))
```

```
    TRUE
```

The next example too?

```
X := -x^2 - y^2 + 3*x + 3*y - 1:
testtype(X, Type::PolyExpr([x, y], Type::Integer))
```

```
    TRUE
```

```
delete X:
```

Parameters

`obj`

Any MuPAD object

`unknowns`

An indeterminate or a list of indeterminates

coeff_type

The type of the coefficients; a type can be an object of the library Type or one of the possible return values of `domtype` and `type`

Return Values

See `testtype`

See Also

MuPAD Functions

`indets` | `poly` | `testtype` | `Type::PolyOf`

Type::PolyOf

Type representing polynomials

Syntax

```
testtype(obj, Type::PolyOf(coeff_type, <num_ind>))
```

Description

With `Type::PolyOf`, polynomials can be identified.

The call `testtype(obj, Type::PolyOf(coeff_type))` checks, whether `obj` is a polynomial with coefficients of type `coeff_type` and, if so, returns `TRUE`, otherwise `FALSE`.

Note: Only polynomials of type `DOM_POLY` can be identified with `Type::PolyOf`, see `Type::PolyExpr` for polynomial expressions.

`coeff_type` determines the type of the coefficients.

The optional argument `num_ind` determines the number of indeterminates. If this argument is not given, the polynomial may have any number of indeterminates.

This type does not represent a property.

Examples

Example 1

Is the object a polynomial with integer coefficients?

```
P := poly(-x^2 - x + 3):
testtype(P, Type::PolyOf(Type::Integer))
```

```
TRUE
```

Is the object a polynomial with integer coefficients and two indets?

```
P := poly(-x^2 - x + 3, [x, y]):
testtype(P, Type::PolyOf(Type::Integer, 2))
```

```
TRUE
```

```
delete P:
```

Parameters

`obj`

Any MuPAD object

`coeff_type`

The type of the coefficientes; a type can be an object of the library `Type` or one of the possible return values of `domtype` and `type`

`num_ind`

The number of indeterminates

Return Values

See `testtype`

See Also

MuPAD Functions

`indets` | `poly` | `testtype`

Type::PosInt

Type and property representing positive integers

Syntax

```
testtype(obj, Type::PosInt)
```

```
assume(x, Type::PosInt)
```

```
is(ex, Type::PosInt)
```

Description

`Type::PosInt` represents positive integers. `Type::PosInt` is a property, too, which can be used in an `assume` call.

The call `testtype(obj, Type::PosInt)` checks, whether `obj` is a positive integer number and returns `TRUE`, if it holds, otherwise `FALSE`.

`testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT` and checks, if `bool(obj > 0)` holds.

The call `assume(x, Type::PosInt)` marks the identifier `x` as a positive integer number.

The call `is(ex, Type::PosInt)` derives, whether the expression `ex` is a positive integer number (or this property can be derived).

This type represents a property that can be used in `assume` and `is`.

Examples

Example 1

The following numbers are of type `Type::PosInt`:

```
testtype(2, Type::PosInt),
testtype(3, Type::PosInt),
testtype(55, Type::PosInt),
testtype(1, Type::PosInt),
testtype(111, Type::PosInt)
```

```
TRUE, TRUE, TRUE, TRUE, TRUE
```

Example 2

Assume an identifier is positive integer:

```
assume(x, Type::PosInt):
is(x, Type::PosInt)
```

```
TRUE
```

Also positive integers are integers:

```
assume(x, Type::PosInt):
is(x, Type::Integer)
```

```
TRUE
```

However, integers can be positive or not:

```
assume(x, Type::Integer):
is(x, Type::PosInt)
```

```
UNKNOWN
```

```
delete x:
```

Parameters

`obj`

Any MuPAD object

x

An identifier or a mathematical expression containing identifiers

ex

An arithmetical expression

Return Values

See `testtype`, `assume` and `is`

See Also

MuPAD Functions
`assume` | `is` | `testtype` | `Type::Property`

Type::PosRat

Type and property representing positive rational numbers

Syntax

```
testtype(obj, Type::PosRat)
```

```
assume(x, Type::PosRat)
```

```
is(ex, Type::PosRat)
```

Description

`Type::PosRat` represents positive rational numbers. `Type::PosRat` is a property, too, which can be used in an `assume` call.

The call `testtype(obj, Type::PosRat)` checks, whether `obj` is a positive rational number and returns `TRUE`, if it holds, otherwise `FALSE`.

`testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT` and `DOM_RAT` and checks, if `bool(obj > 0)` holds.

The call `assume(x, Type::PosRat)` marks the identifier `x` as a positive rational number.

The call `is(ex, Type::PosRat)` derives, whether the expression `ex` is a positive rational number (or this property can be derived).

This type represents a property that can be used in `assume` and `is`.

Examples

Example 1

The following numbers are of type `Type::PosRat`:

```
testtype(2, Type::PosRat),
testtype(3/4, Type::PosRat),
testtype(55/111, Type::PosRat),
testtype(1, Type::PosRat),
testtype(111/111111, Type::PosRat)
```

```
TRUE, TRUE, TRUE, TRUE, TRUE
```

Example 2

Assume an identifier is positive rational:

```
assume(x, Type::PosRat):
is(x, Type::PosRat)
```

```
TRUE
```

Also positive rational numbers are rational:

```
assume(x, Type::PosRat):
is(x, Type::Rational)
```

```
TRUE
```

However, rational numbers can be positive rational or not:

```
assume(x, Type::Rational):
is(x, Type::PosRat)
```

```
UNKNOWN
```

```
delete x:
```

Parameters

`obj`

Any MuPAD object

x

An identifier or a mathematical expression containing identifiers

ex

An arithmetical expression

Return Values

See `testtype`, `assume` and `is`

See Also

MuPAD Functions
`assume` | `is` | `testtype` | `Type::Property`

Type::Positive

Type and property representing positive numbers

Syntax

```
testtype(obj, Type::Positive)
```

```
assume(x, Type::Positive)
```

```
is(ex, Type::Positive)
```

Description

`Type::Positive` represents positive numbers. `Type::Positive` is a property, too, which can be used in an `assume` call.

The call `testtype(obj, Type::Positive)` checks, whether `obj` is a positive real number and returns `TRUE`, if it holds, otherwise `FALSE`.

`testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT`, `DOM_RAT` and `DOM_FLOAT` and checks, if `bool(obj > 0)` holds. This does not include arithmetical expressions such as `exp(1)`, which are not identified as of type `Type::Positive`.

The call `assume(x, Type::Positive)` marks the identifier `x` as a positive real number.

The call `is(ex, Type::Positive)` derives, whether the expression `ex` is a positive real number (or this property can be derived).

This type represents a property that can be used in `assume` and `is`.

Instead of `Type::Positive` the assumption can also be `assume(x > 0)`.

Examples

Example 1

The following numbers are of type `Type::Positive`:

```
testtype(2, Type::Positive),
testtype(3/4, Type::Positive),
testtype(0.123, Type::Positive),
testtype(1, Type::Positive),
testtype(1.02, Type::Positive)
```

```
TRUE, TRUE, TRUE, TRUE, TRUE
```

The following expressions are exact representations of positive numbers, but syntactically they are not of `Type::Positive`:

```
testtype(exp(1), Type::Positive),
testtype(PI^2 + 5, Type::Positive),
testtype(sin(2), Type::Positive)
```

```
FALSE, FALSE, FALSE
```

Ths function `is`, however, realizes that they are, indeed, positive:

```
is(exp(1), Type::Positive),
is(PI^2 + 5, Type::Positive),
is(sin(2), Type::Positive)
```

```
TRUE, TRUE, TRUE
```

Example 2

Assume an identifier is positive:

```
assume(x, Type::Positive):
is(x, Type::Positive)
```

```
TRUE
```

This is equivalent to:

```
assume(x > 0):
is(x > 0)
```

```
    TRUE
```

Also positive numbers are real:

```
assume(x, Type::Positive):
is(x, Type::Real)
```

```
    TRUE
```

But real numbers can be positive or not:

```
assume(x, Type::Real):
is(x, Type::Positive)
```

```
    UNKNOWN
```

```
delete x:
```

Parameters

obj

Any MuPAD object

x

An identifier or a mathematical expression containing identifiers

ex

An arithmetical expression

Return Values

See `testtype`, `assume` and `is`

See Also

MuPAD Functions

`assume` | `is` | `testtype` | `Type::Property`

Type::Predicate

Type for testing object satisfying a given predicate

Syntax

```
testtype(obj, Type::Predicate(<pname>, predicate, <p1, p2, …>))
```

Description

`Type::Predicate(predicate)` represents the MuPAD objects which satisfy the predicate `predicate`.

The call `testtype(obj, pname, Type::Predicate(< pname >, predicate , < p1 , p2 >))` test whether `obj` satisfies `predicate`; that is it returns `predicate(obj, p1, p2, ...)`.

`Type::Predicate(predicate)`, `Type::Predicate(name, predicate)`, `Type::Predicate ( predicate( p1 , p2 , …))`, and `Type::Predicate(name, predicate p1 , p2 , , …)` are respectively pretty printed as `Type::Predicate(predicate)`, `Type::name`, `Type::Predicate(p, p1, p2, ...)`, and `Type::name(p1, p2, ...)`.

Examples

Example 1

We define a type which contains any MuPAD object:

```
t := Type::Predicate(x -> TRUE):
testtype(1, t), testtype(2, t), testtype(x, t)
```

```
TRUE, TRUE, TRUE
```

We define a type which contains all the MuPAD object which are solution of `(x-1)*(x+1)=0`:

```
t := Type::Predicate(x -> bool((x - 1)*(x + 1) = 0)):
testtype(1, t), testtype(2, t), testtype(x, t)
```

```
TRUE, FALSE, FALSE
```

We define a type for partitions, that is, decreasing lists of integers:

```
part :=
    Type::Predicate(l -> _lazy_and(testtype(l, Type::ListOf(Type::Integer)),
                                   bool(revert(sort(l)) = l))):
testtype(a, part), testtype([3, 6, 1], part), testtype([3, 2, 2], part)
```

```
FALSE, FALSE, TRUE
```

Using the naming facility is recommended to improve the readability of error messages:

```
part :=
    Type::Predicate("Partition",
                    l -> _lazy_and(testtype(l, Type::ListOf(Type::Integer)),
                                   bool(revert(sort(l)) = l))):
f := proc(p: part) begin end:
f(3);
```

```
Error: The object '3' is incorrect. The type of argument number 1 must be 'Type::Partit
  Evaluating: f
```

Parameters

pname

A string which will be used for pretty printing the type

predicate

A function of one argument which can return TRUE, FALSE or FAIL

obj, p1, p2, …

Any MuPAD objects

Return Values

See `testtype`

See Also

MuPAD Functions
`testtype`

Type::Prime

Type representing prime numbers

Syntax

```
testtype(obj, Type::Prime)
```

Description

`Type::Prime` represents prime numbers.

The call `testtype(obj, Type::Prime)` returns `TRUE` if `obj` is a prime number, and `FALSE` otherwise.

`testtype` only performs a syntactical test whether `obj` is an integer and `isprime(obj)` holds.

Examples

Example 1

The following numbers are of type `Type::Prime`:

```
testtype(2, Type::Prime),
testtype(3, Type::Prime),
testtype(7, Type::Prime),
testtype(11113, Type::Prime),
testtype(4185296581467695597, Type::Prime)
```

```
TRUE, TRUE, TRUE, TRUE, TRUE
```

Parameters

`obj`

Any MuPAD object

Return Values

See `testtype`

See Also

MuPAD Functions
`isprime` | `testtype`

Type::Product

Type representing sequences

Syntax

```
testtype(obj, Type::Product(typedef, …))
```

Description

`Type::Product` is the type of sequences of objects of different types.

The call `testtype(obj, Type::Product( typedef , …))` checks, whether `obj` is a sequence of objects, which have the types given by `typedef` and returns `TRUE`, if it holds, otherwise `FALSE`.

`obj` must have the same number of arguments as the sequence `typedef`. The elements of `obj` are checked one after another: the first element of `obj` is checked against the type given by the first element of `typedef` and so on. All elements and types must match.

typedef, … must be a nonempty sequence of types. A type can be an object of the library Type or one of the possible return values of `domtype` and `type`.

This type does not represent a property.

Examples

Example 1

The argument is a sequence of a positive integer followed by an identifier:

```
testtype((5, x), Type::Product(Type::PosInt, Type::Unknown))
```

```
    TRUE
```

Is the argument is a sequence of five positive integers?

```
testtype((5, 3, 5, -1, 0), Type::Product(Type::PosInt $ 5))
```

```
    FALSE
```

Parameters

`obj`

Any MuPAD object

`typedef`

A sequence of types; a type can be an object of the library Type or one of the possible return values of `domtype` and `type`

Return Values

See `testtype`

See Also

MuPAD Functions

`testtype` | `Type::ListProduct`

Type::Property

Type representing any property

Syntax

```
testtype(obj, Type::Property)
```

Description

With `Type::Property`, properties can be identified.

The call `testtype(obj, Type::Property)` checks, whether the MuPAD object `obj` is a property and returns `TRUE`, if it holds, otherwise `FALSE`.

Some elements of the library `Type` serve two functions. One is to perform syntactical tests to identify the type of an object (with `testtype`), the other is to occur as a property within `assume` and `is`.

Note: `Type::Property` itself is not a property.

To determine whether an element of `Type` is a property, `Type::Property` can be used with `testtype`.

This type does not represent a property.

Examples

Example 1

Is `Type::PosInt` a property?

```
testtype(Type::PosInt, Type::Property)
```

```
TRUE
```

Also an interval created with `Type::Interval` is a property:

```
testtype(Type::Interval(0, 1), Type::Property)
```

```
TRUE
```

Is `Type::Constant` a property?

```
testtype(Type::Constant, Type::Property)
```

```
FALSE
```

`Type::Constant` is not a property and cannot be used as argument of `assume`:

```
assume(x, Type::Constant)
```

```
Error: The second argument must be a property. [assume]
```

The next example shows the usage of `testtype` to select properties among operands of `Type`:

```
T := Type::Numeric, Type::PosInt, Type::Unknown, Type::Zero:
select(T, testtype, Type::Property)
```

```
Type::PosInt, Type::Zero
```

```
delete x, T:
```

Parameters

`obj`

Any MuPAD object

Return Values

See `testtype`

See Also

MuPAD Functions
`is` | `testtype`

Type::RatExpr

Type representing rational expressions

Syntax

```
testtype(obj, Type::RatExpr(indet, <coeff_type>))
```

Description

With `Type::RatExpr`, rational expressions can be identified.

The call `testtype(obj, Type::RatExpr(indet))` checks, whether `obj` is a rational expression in the indeterminante `indet`, i.e., the quotient of two polynomial expressions in `indet`. If it is, the result is `TRUE`, otherwise `FALSE`.

A rational expression in `indet` is a expression, and `indet` occurs only as operand of `_plus` or `_mult` expressions and in `_power` with an integer exponent.

`indet` must be an identifier, and `coeff_type` a type for the coefficients of the rational expression.

This type does not represent a property.

Examples

Example 1

A polynomial expression in `x` is also a rational expression in `x`:

```
testtype(-x^2 - x + 3, Type::RatExpr(x))
```

```
    TRUE
```

`testtype` is used to select all rational operands in `x` with positive integer coefficients:

```
EX := sin(x) + x^2 - 3*x + 2 + 3/x:
select(EX, testtype, Type::RatExpr(x, Type::PosInt))
```

$$\frac{3}{x} + x^2 + 2$$

```
delete EX:
```

Parameters

`obj`

Any MuPAD object

`indet`

An indeterminante

`coeff_type`

A type for the coefficientes; a type can be an object of the library Type or one of the possible return values of `domtype` and `type`

Return Values

See `testtype`

See Also

MuPAD Functions
`indets` | `testtype`

Type::Rational

Type and property representing rational numbers

Syntax

```
testtype(obj, Type::Rational)
```

```
assume(x, Type::Rational)
```

```
is(ex, Type::Rational)
```

Description

`Type::Rational` represents rational numbers. `Type::Rational` is a property, too, which can be used in an `assume` call.

The call `testtype(obj, Type::Rational)` checks, whether `obj` is a rational number and returns `TRUE`, if it holds, otherwise `FALSE`.

`testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT` and `DOM_RAT`.

The call `assume(x, Type::Rational)` marks the identifier `x` as a rational number.

The call `is(ex, Type::Rational)` derives, whether the expression `ex` is a rational number (or this property can be derived).

This type represents a property that can be used in `assume` and `is`.

Examples

Example 1

The following numbers are of type `Type::Rational`:

```
testtype(2, Type::Rational),
```

```
testtype(3/4, Type::Rational),
testtype(-1/2, Type::Rational),
testtype(-1, Type::Rational),
testtype(1024/11111, Type::Rational)
```

```
TRUE, TRUE, TRUE, TRUE, TRUE
```

Example 2

Integers are rational:

```
assume(x, Type::Integer):
is(x, Type::Rational)
```

```
TRUE
```

However, rational numbers can be integer or not:

```
assume(x, Type::Rational):
is(x, Type::Integer)
```

```
UNKNOWN
```

```
delete x:
```

Parameters

`obj`

Any MuPAD object

`x`

An identifier or a mathematical expression containing identifiers

`ex`

An arithmetical expression

Return Values

See `testtype`, `assume` and `is`

See Also

MuPAD Functions
`assume` | `is` | `testtype` | `Type::Property`

Type::Real

Type and property representing real numbers

Syntax

```
testtype(obj, Type::Real)
```

```
assume(x, Type::Real)
```

```
is(ex, Type::Real)
```

Description

`Type::Real` represents real numbers. `Type::Real` is a property, too, which can be used in an `assume` call.

The call `testtype(obj, Type::Real)` checks, whether `obj` is a real number and, if it is, returns `TRUE`, otherwise `FALSE`.

`testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT`, `DOM_RAT` and `DOM_FLOAT`. This does not include arithmetical expressions such as `exp(1)`, which are not identified as of type `Type::Real`.

The call `assume(x, Type::Real)` marks the identifier `x` as a real number.

The call `is(ex, Type::Real)` derives, whether the expression `ex` is a real number (or this property can be derived).

This type represents a property that can be used in `assume` and `is`.

Examples

Example 1

The following numbers are of type `Type::Real`:

```
testtype(2, Type::Real),
testtype(3/4, Type::Real),
testtype(0.123, Type::Real),
testtype(-1, Type::Real),
testtype(-1.02, Type::Real)
```

```
TRUE, TRUE, TRUE, TRUE, TRUE
```

The following expressions are exact representations of real numbers, but syntactically they are not of `Type::Real`:

```
testtype(exp(1), Type::Real),
testtype(PI^2 + 5, Type::Real),
testtype(sin(2), Type::Real)
```

```
FALSE, FALSE, FALSE
```

The function `is` performs a semantical, mathematically more useful check:

```
is(exp(1), Type::Real),
is(PI^2 + 5, Type::Real),
is(sin(2), Type::Real)
```

```
TRUE, TRUE, TRUE
```

Example 2

Integers are real numbers:

```
assume(x, Type::Integer):
is(x, Type::Real)
```

```
TRUE
```

But real numbers can be integer or not:

```
assume(x, Type::Real):
is(x, Type::Integer)
```

```
UNKNOWN
```

The sine of a real number is a real number in the interval [- 1, 1]:

```
getprop(sin(x))
```

$$[-1, 1]$$

```
delete x:
```

Parameters

obj

Any MuPAD object

x

An identifier or a mathematical expression containing identifiers

ex

An arithmetical expression

Return Values

See `testtype`, `assume` and `is`

See Also

MuPAD Functions

`assume` | `is` | `testtype` | `Type::Property`

Type::Relation

Type representing relations

Syntax

```
testtype(obj, Type::Relation)
```

Description

With `Type::Relation`, relational expression can be identified.

The call `testtype(obj, Type::Relation)` checks, whether `obj` is a relational expression and returns `TRUE`, if it is, otherwise `FALSE`.

A relation in MuPAD is an expression of the type `" _equal "`, `" _unequal "`, `" _less "` and `" _leequal "`.

Note: Expressions with the operations >= and > will be interpreted as expressions with <= and < by exchanging the operands (see “Example 2” on page 33-110).

This type does not represent a property.

Examples

Example 1

`x > 3` is a relation, while `TRUE` is not:

```
testtype(x > 3, Type::Relation),
testtype(TRUE, Type::Relation)
```

```
TRUE, FALSE
```

Example 2

MuPAD always interprets expressions with the operations >= and > as expressions with <= and < with the operands exchanged:

```
x > 3;
prog::exprtree(x > 3):
```

$$3 < x$$

```
_less
|
+-- 3
|
`-- x
```

The operator is *not* >, but <, and the operands have been swapped:

```
op(x > 3, 0..2)
```

$$\_less,\ 3,\ x$$

Parameters

obj

Any MuPAD object

Return Values

See `testtype`

See Also

MuPAD Functions
`testtype`

Type::Residue

Property representing a residue class

Syntax

```
assume(x, (rem, class, <sub_set>))

is(ex, (rem, class, <sub_set>))

testtype(ex, (rem, class, <sub_set>))
```

Description

`Type::Residue(rem, class)` represents the integers *n* for which *n* - *rem* is divisible by *class*.

The call `assume(x, Type::Residue(rem, class))` marks the identifier `x` as an integer divisible by `class` with remainder `rem`.

The call `is(ex, Type::Residue(rem, class))` derives, whether the expression `ex` is an integer divisible by `class` with remainder `rem` (or this property can be derived).

This type represents a property that can be used in `assume` and `is`.

`Type::Even` and `Type::Odd` are objects created by `Type::Residue`.

The call `testtype(obj, Type::Residue(rem, class))` checks, whether `obj` is an integer and is divisible by `class` with remainder `rem`. If the optional argument `sub_set` is given, `testtype` checks additionally `testtype(obj, sub_set)`.

Examples

Example 1

`Type::Residue` can be used in `testtype`:

```
testtype(6, Type::Residue(2, 4)),
testtype(13, Type::Residue(1, 20))
```

TRUE, FALSE

Example 2

x is assumed to be divisible by 3 with remainder 1:

```
assume(x, Type::Residue(1, 3))
```

Which properties has x + 2 got?

```
getprop(x + 2)
```

$\{3\,k \mid k \in \mathbb{Z}\}$

x is an integer, but it may be odd or not:

```
is(x, Type::Integer), is(x, Type::Odd)
```

TRUE, UNKNOWN

This example restricts possible values of x to odd integers:

```
assume(x, Type::Residue(1, 4));
is(x, Type::Odd),
is((-1)^x < 0)
```

TRUE, TRUE

Parameters

x

An identifier or a mathematical expression containing identifiers

`rem`

Remainder as integer number between `0` and `class - 1`; an integer larger than `class - 1` will be divided by `class` and `rem` gets the remainder of this division

`class`

The divider as positive integer

`sub_set`

A subset of the integers (e.g., `Type::PosInt`); otherwise `Type::Integer` is used

`ex`

An arithmetical expression

`obj`

Any MuPAD object

Return Values

See `assume`, `is` and `testtype`

See Also

MuPAD Functions

`assume` | `is` | `testtype` | `Type::Even` | `Type::Integer` | `Type::Odd`

Type::SequenceOf

Type representing sequences

Syntax

```
testtype(obj, (obj_type, <min_nr, <max_nr>>))
```

Description

With `Type::SequenceOf`, sequences with specified objects can be identified.

The call `testtype(obj, Type::SequenceOf(obj_type))` checks, whether `obj` is a sequence with elements of the given type `obj_type`. In that case, it `TRUE`, otherwise `FALSE`.

A sequence has the domain type `DOM_EXPR` and the type `"_exprseq"`.

The two optional parameters `min_nr` and `max_nr` determine the minimum and maximum number of arguments of the analysed sequence, respectively. If the numbers are not be given, the number of elements of the sequence will not be checked. If only the minimum is given, the sequence must have at least `min_nr` elements for the test to succeed.

This type does not represent a property.

Examples

Example 1

Is the given sequence a sequence of identifiers?

```
testtype((a, b, c, d, e, f), Type::SequenceOf(DOM_IDENT))
```

```
    TRUE
```

Is the given sequence a sequence of at least five real numbers?

```
testtype((0, 0.5, 1, 1.5, 2, 2.5, 3), Type::SequenceOf(Type::Real, 5))
```

```
    TRUE
```

Parameters

`obj`

Any MuPAD object

`obj_type`

The type of the objects; a type can be an object of the library Type or one of the possible return values of `domtype` and `type`

`min_nr`

The minimal number of objects as nonnegative integer

`max_nr`

The maximal number of objects as nonnegative integer

Return Values

See `testtype`

See Also

MuPAD Functions

`_exprseq` | `testtype` | `Type::ListOf`

Type::Series

Type representing truncated Puiseux, Laurent, and Taylor series expansions

Syntax

```
testtype(obj, (Puiseux | Laurent | Taylor))
testtype(obj, (Puiseux | Laurent | Taylor, x))
testtype(obj, (Puiseux | Laurent | Taylor, x = x0))
testtype(obj, (Puiseux | Laurent | Taylor, x, Undirected | Real | Right | Left))
testtype(obj, (Puiseux | Laurent | Taylor, x = x0, Undirected | Real | Right | Left))
```

Description

`Type::Series(Puiseux)`, `Type::Series(Laurent)`, and `Type::Series(Taylor)` represent truncated Puiseux series, Laurent series, and Taylor series, respectively.

The call `testtype(obj, Type::Series(T))` checks, whether `obj` is a truncated series expansion of domain type `Series::Puiseux` and of mathematical type `T`.

The call `testtype(obj, Type::Series(T, x = x0))` checks in addition, whether the series variable is `x` and the expansion point is `x0`. If `x0` is omitted, `x0 = 0` is assumed.

The call `testtype(obj, Type::Series(T, x = x0, dir))` checks in addition, whether the direction of expansion is compatible with the specified direction `dir`. See the help pages of `series` and `Series::Puiseux` for more details about the direction.

See “Example 1” on page 33-117 and “Example 3” on page 33-119.

If `obj` is of domain type `Series::Puiseux`, but not a Puiseux expansion in the mathematical sense, then `testtype(obj, Type::Series(T))` returns `FAIL`. This is the case if the coefficients of `obj` depend on the series variable, or if the type flag of `obj` is 1. See “Example 2” on page 33-118, and the help page of `Series::Puiseux` for more details.

A Laurent series is a Puiseux series with integral exponents. If the expansion point is finite, then a Taylor series is a Puiseux series with nonnegative integral exponents. If the expansion point is `complexInfinity`, then a Taylor series is a Puiseux series with nonpositive integral exponents. See “Example 1” on page 33-117 and “Example 4” on page 33-120.

For the expansion points `infinity` and `-infinity`, the directions `Left` and `Right`, respectively, are implicitly assumed.

Specifying `x0 = infinity` is equivalent to `x0 = complexInfinity` and `dir = Left`, and similarly `x0 = -infinity` is equivalent to `x0 = complexInfinity` and `dir = Right`.

See “Example 4” on page 33-120.

This type does not represent a property: it cannot be used in `assume` to mark an identifier as a truncated series expansion.

Examples

Example 1

The following call returns a Puiseux series:

```
s := series(sin(sqrt(x)), x);
domtype(s);
```

$$\sqrt{x} - \frac{x^{3/2}}{6} + \frac{x^{5/2}}{120} + O\left(x^{7/2}\right)$$

Series::Puiseux

However, `s` is not a Laurent series:

```
testtype(s, Type::Series(Puiseux)),
testtype(s, Type::Series(Laurent)),
testtype(s, Type::Series(Taylor))
```

TRUE, FALSE, FALSE

A Laurent series that is not a Taylor series:

```
s := series(1/sin(x), x);
domtype(s);
```

$$\frac{1}{x} + \frac{x}{6} + \frac{7\,x^3}{360} + O(x^5)$$

Series::Puiseux

```
testtype(s, Type::Series(Puiseux)),
testtype(s, Type::Series(Laurent)),
testtype(s, Type::Series(Taylor))
```

TRUE, TRUE, FALSE

The inverse of `s` is a Taylor series:

```
1/s;
testtype(1/s, Type::Series(Puiseux)),
testtype(1/s, Type::Series(Laurent)),
testtype(1/s, Type::Series(Taylor))
```

$$x - \frac{x^3}{6} + \frac{x^5}{120} + O(x^7)$$

TRUE, TRUE, TRUE

Example 2

`Type::Series` represents only objects of domain type `Series::Puiseux`:

```
s := 1 + x + 2*x^2 + O(x^3);
domtype(s), testtype(s, Type::Series(Puiseux));
```

$$x + 2\,x^2 + 1 + O(x^3)$$

DOM_EXPR, FALSE

```
s := series(exp(x + 1/x), x = infinity, 3);
domtype(s), testtype(s, Type::Series(Puiseux));
```

$$e^x + \frac{e^x}{x} + \frac{e^x}{2\,x^2} + O\left(\frac{e^x}{x^3}\right)$$

Series::gseries, FALSE

For objects of domain type `Series::Puiseux`, whose coefficients contain the series variable or whose type flag is 1, the result is `FALSE` as well:

```
s := series(psi(x), x = infinity);
domtype(s), coeff(s, 0), testtype(s, Type::Series(Puiseux));
```

$$\ln(x) - \frac{1}{2\,x} - \frac{1}{12\,x^2} + \frac{1}{120\,x^4} + O\left(\frac{1}{x^6}\right)$$

Series::Puiseux, $\ln(x)$, FALSE

```
s := series(sin(sqrt(-x)), x);
domtype(s), testtype(s, Type::Series(Puiseux));
```

$$\sqrt{-x} - \frac{(-x)^{3/2}}{6} + \frac{(-x)^{5/2}}{120} + O\left(x^{7/2}\right)$$

Series::Puiseux, FALSE

Example 3

By specifying further arguments, you can check for the series variable, the expansion point, and the direction of expansion as well:

```
s := series(sin(sqrt(-x)), x, Left);
testtype(s, Type::Series(Puiseux, y)),
testtype(s, Type::Series(Puiseux, x)),
testtype(s, Type::Series(Puiseux, x = 0)),
testtype(s, Type::Series(Puiseux, x = 2));
```

$$-\sqrt{x}\,\mathrm{i}-\frac{x^{3/2}\,\mathrm{i}}{6}-\frac{x^{5/2}\,\mathrm{i}}{120}+O\left(x^{7/2}\right)$$

FALSE, TRUE, TRUE, FALSE

```
Series::Puiseux::direction(s),
testtype(s, Type::Series(Puiseux, x, Undirected)),
testtype(s, Type::Series(Puiseux, x, Real)),
testtype(s, Type::Series(Puiseux, x, Right)),
testtype(s, Type::Series(Puiseux, x, Left));
```

Left, FALSE, FALSE, FALSE, TRUE

```
s := series(x^5/(x - 2), x = 2, 3);
testtype(s, Type::Series(Laurent, x)),
testtype(s, Type::Series(Laurent, x = 2)),
testtype(s, Type::Series(Laurent, x = 3));
```

$$\frac{32}{x-2}+80+80\,(x-2)+O((x-2)^2)$$

FALSE, TRUE, FALSE

If you specify a direction, `testtype` checks whether it is compatible with the direction of the series:

```
Series::Puiseux::direction(s),
testtype(s, Type::Series(Puiseux, x = 2, Undirected)),
testtype(s, Type::Series(Puiseux, x = 2, Real)),
testtype(s, Type::Series(Puiseux, x = 2, Right)),
testtype(s, Type::Series(Puiseux, x = 2, Left));
```

Undirected, TRUE, TRUE, TRUE, TRUE

Example 4

The following example is a Laurent series around `infinity`, but not a Taylor series:

```
s := series(z*exp(1/z), z = infinity);
testtype(s, Type::Series(Puiseux)),
testtype(s, Type::Series(Laurent)),
testtype(s, Type::Series(Taylor))
```

$$z + 1 + \frac{1}{2\,z} + \frac{1}{6\,z^2} + \frac{1}{24\,z^3} + \frac{1}{120\,z^4} + O\left(\frac{1}{z^5}\right)$$

TRUE, TRUE, FALSE

The expansion point is `infinity`, or equivalently, `complexInfinity` from the left:

```
Series::Puiseux::point(s), Series::Puiseux::direction(s);
testtype(s, Type::Series(Laurent, z)),
testtype(s, Type::Series(Laurent, z = 0)),
testtype(s, Type::Series(Laurent, z = infinity)),
testtype(s, Type::Series(Laurent, z = -infinity)),
testtype(s, Type::Series(Laurent, z = complexInfinity));
```

complexInfinity, Left

FALSE, FALSE, TRUE, FALSE, TRUE

```
testtype(s, Type::Series(Laurent, z = complexInfinity, Undirected)),
testtype(s, Type::Series(Laurent, z = complexInfinity, Real)),
testtype(s, Type::Series(Laurent, z = complexInfinity, Right)),
testtype(s, Type::Series(Laurent, z = complexInfinity, Left));
```

FALSE, FALSE, FALSE, TRUE

Mathematically, the expression is even an undirected expansion around `complexInfinity`:

```
s := series(z*exp(1/z), z = complexInfinity);
Series::Puiseux::point(s), Series::Puiseux::direction(s);
```

$$z + 1 + \frac{1}{2\,z} + \frac{1}{6\,z^2} + \frac{1}{24\,z^3} + \frac{1}{120\,z^4} + O\left(\frac{1}{z^5}\right)$$

```
      complexInfinity, Undirected
```

```
testtype(s, Type::Series(Laurent, z)),
testtype(s, Type::Series(Laurent, z = infinity)),
testtype(s, Type::Series(Laurent, z = -infinity)),
testtype(s, Type::Series(Laurent, z = complexInfinity));
```

```
      FALSE, TRUE, TRUE, TRUE
```

```
testtype(s, Type::Series(Laurent, z = complexInfinity, Undirected)),
testtype(s, Type::Series(Laurent, z = complexInfinity, Real)),
testtype(s, Type::Series(Laurent, z = complexInfinity, Right)),
testtype(s, Type::Series(Laurent, z = complexInfinity, Left));
```

```
      TRUE, TRUE, TRUE, TRUE
```

Parameters

`obj`

Any MuPAD object

`x`

The series variable: an identifier

`x0`

The expansion point: an arithmetical expression

Options

`Laurent`, **`Puiseux`**, **`Taylor`**

The type of series

`Left`, **`Real`**, **`Right`**, **`Undirected`**

The direction of the expansion

Return Values

See `testtype`

See Also

MuPAD Functions
`series` | `Series::Puiseux` | `testtype` | `Type::PolyExpr` | `Type::PolyOf`

Type::SetOf

Type representing sets

Syntax

```
testtype(obj, (obj_type, <min_nr, <max_nr>>))
```

Description

`Type::SetOf(obj_type)` describes sets of elements of type `obj_type`.

The call `testtype(obj, Type::SetOf(obj_type))` checks, whether `obj` is a set with elements of the given type `obj_type`. If it is, the function returns `TRUE`, otherwise `FALSE`.

A set has the domain type `DOM_SET`.

The two optional parameters `min_nr` and `max_nr` determine the minimum and maximum number of elements in the analysed set. If the numbers are not be given, the number of elements in the set will not be checked. If only the minimum is given, the set must contain at least `min_nr` elements for the test to succeed.

This type does not represent a property.

Examples

Example 1

Is the given set a set of identifiers?

```
testtype({a, b, c, d, e, f}, Type::SetOf(DOM_IDENT))
```

```
    TRUE
```

Is the given set a set of at least five real numbers?

```
testtype({0, 0.5, 1, 1.5, 2, 2.5, 3}, Type::SetOf(Type::Real, 5))
```

TRUE

Example 2

`testtype` is used to select sets with exactly two idetifiers:

```
S := {{a}, {a, b}, {d, 1}, {0, d}, {e}, {d, e}}:
select(S, testtype, Type::SetOf(DOM_IDENT, 2, 2))
```

$\{\{d, e\}, \{a, b\}\}$

Parameters

`obj`

Any MuPAD object

`obj_type`

The type of the objects; a type can be an object of the library `Type` or one of the possible return values of `domtype` and `type`

`min_nr`

The minimal number of objects as nonnegative integer

`max_nr`

The maximal number of objects as nonnegative integer

Return Values

See `testtype`

See Also

MuPAD Domains
DOM_SET

MuPAD Functions
testtype | Type::ListOf | Type::Union

Type::Set

Type representing set-theoretic expressions

Syntax

```
testtype(obj, Type::Set)
```

Description

`Type::Set` comprises all expressions in which the operators are set-theoretic operations

A set-theoretic expression is defined to be any of the following: a set constant, an identifier, an unevaluated call to a set-valued function, or the composition of set-theoretic expressions by the operator `union`, `intersect`, or `minus`.

The following objects are set constants: finite sets of type `DOM_SET`, intervals, the `universe`, and every object that belongs to a domain of category `Cat::Set`.

The following functions are set-valued: `solve`, `discont`, `RootOf`, and `solvelib::Union`.

The union, intersection, or difference of objects is not a set-theoretic expression unless each of the objects is. See “Example 2” on page 33-128.

Examples

Example 1

Sets are set-theoretic expressions.

```
testtype({3}, Type::Set)
```

```
    TRUE
```

Example 2

Unions, intersections, and differences are set-theoretic expressions if and only if all operands are.

```
testtype(a union {4}, Type::Set)
```

TRUE

```
testtype(a+1 union {4}, Type::Set)
```

FALSE

Example 3

If the call to a set-valued function as `solve` returns unevaluated, then the result is a set-theoretic expression.

```
solve(x^2 = sin(x + 1), x)
```

$$\text{solve}(x^2 - \sin(x+1) = 0, x)$$

```
testtype(%, Type::Set)
```

TRUE

Parameters

`obj`

Any MuPAD object

Return Values

See `testtype`

See Also

MuPAD Functions
`testtype`

Type::Singleton

Type representing exactly one object

Syntax

```
testtype(obj, (t_obj))
```

Description

`testtype(x, Type::Singleton(y))` is equivalent to `bool(x = y)`.

The call `testtype(obj, Type::Singleton(t_obj))` is equivalent to `bool(x = y)`, but the latter is faster.

`Type::Singleton` can be used to create combined types, especially in conjunction with `Type::Union`, `Type::Equation` and other types expecting type information for subexpressions (see “Example 2” on page 33-131).

This type does not represent a property.

Examples

Example 1

Check, if `x` is really `x`:

```
testtype(x, Type::Singleton(x))
```

```
    TRUE
```

But the next call does the same:

```
bool(x = x)
```

```
    TRUE
```

Example 2

`Type::Singleton` exists to create special testing expressions:

```
T := Type::Union(Type::Singleton(hold(All)), Type::Constant):
```

With the type `T` the option `All` and any constant can be identified with one call of testtype:

```
testtype(4, T), testtype(hold(All), T), testtype(x, T)
```

```
TRUE, TRUE, FALSE
```

But (e.g., in procedures) the following example works faster:

```
test := X -> testtype(X, Type::Constant) or bool(X = hold(All)):
test(4), test(hold(All)), test(x)
```

```
TRUE, TRUE, FALSE
```

One way to test a list of options for syntactical correctness is the following:

```
T := Type::Union(
       // Name = "..."
       Type::Equation(Type::Singleton(hold(Name)), DOM_STRING),
       // Mode = n, n in {1, 2, 3}
       Type::Equation(Type::Singleton(hold(Mode)),
                      Type::Interval([1,3], Type::Integer)),
       // Quiet
       Type::Singleton(hold(Quiet))
     ):

testtype((Name = "abcde", Quiet), Type::SequenceOf(T))
```

```
TRUE
```

We only allow the values 1, 2, and 3 for `Mode`, however:

```
testtype((Quiet, Mode = 0), Type::SequenceOf(T))
```

```
FALSE
```

Obviously, it would be a good idea to tell the user which options we could not grok:

```
error("Unknown option(s): ".expr2text(
        select((Quiet, Mode = 0),
               not testtype, Type::SequenceOf(T))))
```

```
Error: Unknown option(s): Mode = 0
```

```
delete T, test:
```

Parameters

obj

Any MuPAD object

t_obj

Any object to identify

Return Values

See `testtype`

See Also

MuPAD Functions

`_equal` | `bool` | `testtype` | `Type::Union`

Type::TableOfEntry

Type representing tables with specified entries

Syntax

```
testtype(obj, (obj_type))
```

Description

`Type::TableOfEntry(obj_type)` describes tables with *entries* of type `obj_type`.

The call `testtype(obj, Type::TableOfEntry(obj_type))` checks, whether `obj` is a table and all entries of this table are of the type `obj_type`. If both conditions are met, the call returns `TRUE`, otherwise `FALSE`.

The entries of a table are the right hand sides of the operands of a table.

This type does not represent a property.

Examples

Example 1

The following table uses identifiers as keys and integers as entries:

```
T := table(a = 1, b = 2, c = 3, d = 4):
testtype(T, Type::TableOfEntry(DOM_INT))
```

```
    TRUE
```

`Type::TableOfEntry` only checks the type of the entries, not the keys:

```
T := table(a = 1, b = 2, c = 3, d = 4):
testtype(T, Type::TableOfEntry(DOM_IDENT))
```

```
    FALSE

delete T:
```

Parameters

obj

Any MuPAD object

obj_type

The type of the entries; can be an object of the library Type or one of the possible return values of `domtype` and `type`

Return Values

See `testtype`

See Also

MuPAD Functions

`table` | `testtype` | `Type::TableOfIndex`

Type::TableOfIndex

Type representing tables with specified indices

Syntax

```
testtype(obj, (obj_type))
```

Description

`Type::TableOfIndex(obj_type)` represents tables with *indices* (keys) of type `obj_type`.

The call `testtype(obj, Type::TableOfIndex(obj_type))` checks, whether `obj` is a table and all indices (keys) are of the type `obj_type`. If both conditions are met, the call returns `TRUE`, otherwise `FALSE`.

The indices of a table are the left hand sides of the operands of a table.

This type does not represent a property.

Examples

Example 1

The following table uses identifiers as keys and integers as values:

```
T := table(a = 1, b = 2, c = 3, d = 4):
testtype(T, Type::TableOfIndex(DOM_IDENT))
```

```
    TRUE
```

`Type::TableOfIndex` only checks the types of the keys of the table, so the following call returns `FALSE`:

```
T := table(a = 1, b = 2, c = 3, d = 4):
```

```
testtype(T, Type::TableOfIndex(DOM_INT))

    FALSE

delete T:
```

Parameters

obj

Any MuPAD object

obj_type

The type of the indices; can be an object of the library Type or one of the possible return values of `domtype` and `type`

Return Values

See `testtype`

See Also

MuPAD Functions

`table` | `testtype` | `Type::TableOfEntry`

Type::TableOf

Type representing tables

Syntax

```
testtype(obj, (<indices_type, <entries_type>>))
```

Description

`Type::TableOf` represents tables; the types of the *indices* and of the *entries* can be specified.

The call `testtype(obj, Type::TableOf(indices_type, entries_type))` checks, whether `obj` is a table with indices of type `indices_type` and entries of type `entries_type`.

The *indices* (resp. the *entries*) of a table are the left (resp. right) hand sides of the operands of a table.

`indices_type` and `entries_type` default to `Type::AnyType`

This type does not represent a property.

Examples

Example 1

We test if the following objects are tables:

```
testtype(x, Type::TableOf());
testtype(table(), Type::TableOf())
```

```
    FALSE

    TRUE
```

We test if the following objects are tables with integer indexes:

```
testtype(table(a = 1), Type::TableOf(Type::Integer));
testtype(table(1 = 2), Type::TableOf(Type::Integer))
```

```
    FALSE
```

```
    TRUE
```

We test if the following objects are tables with integer entries:

```
testtype(table(a = a), Type::TableOf(Type::AnyType, Type::Integer));
testtype(table(a = 2), Type::TableOf(Type::AnyType, Type::Integer))
```

```
    FALSE
```

```
    TRUE
```

We test if the following objects are tables with integer indexes and entries:

```
testtype(table(a = a), Type::TableOf(Type::Integer, Type::Integer));
testtype(table(1 = 2), Type::TableOf(Type::Integer, Type::Integer))
```

```
    FALSE
```

```
    TRUE
```

Example 2

Test if the following table uses identifiers as indexes:

```
T := table(a = 1, b = 2, c = 3, d = 4):
testtype(T, Type::TableOf(DOM_IDENT))
```

```
    TRUE
```

Test if the following table uses integers as indexes:

```
T := table(a = 1, b = 2, c = 3, d = 4):
testtype(T, Type::TableOf(DOM_INT))
```

```
    FALSE
```

```
delete T:
```

Example 3

The following table uses identifiers as keys and integers as entries:

```
T := table(a = 1, b = 2, c = 3, d = 4):
testtype(T, Type::TableOf(Type::AnyType, DOM_INT))
```

```
    TRUE
```

`Type::TableOf` only checks the type of the entries, not the keys:

```
T := table(a = 1, b = 2, c = 3, d = 4):
testtype(T, Type::TableOf(Type::AnyType, DOM_IDENT))
```

```
    FALSE
```

```
delete T:
```

Parameters

`obj`

Any MuPAD object

`indices_type`

The type of the indices. It can be an object of the library Type or one of the possible return values of `domtype` and `type`

`entries_type`

The type of the entries.

Return Values

See `testtype`

See Also

MuPAD Functions
`table` | `testtype` | `Type::TableOfEntry` | `Type::TableOfIndex`

Type::Union

Type representing several types as one type object

Syntax

```
testtype(obj, (obj_types, …))
```

Description

`Type::Union ( type1 , type2 , …)` represents all objects having at least one of the types type1, type2, …

The call `testtype(obj, Type::Union( obj_types , …))` checks, whether `obj` has the type of at least one of the given types obj_types, …. If such a type is found, the call returns `TRUE`, otherwise `FALSE`.

The call `testtype(Type::Union( obj , obj_types , …))` is thus equivalent to the call `_lazy_or(map(obj_types, x -> testtype(obj, x)))`, testing `obj` against all types in turn until one is found which matches.

obj_types, … must be a (nonempty) sequence of types (see `testtype`).

This type does not represent a property.

Examples

Example 1

Check, whether the given object is a positive or negative integer:

```
testtype(2, Type::Union(Type::PosInt, Type::NegInt))
```

```
    TRUE
```

`x` however, is neither a positive nor a negative number:

```
testtype(x, Type::Union(Type::Positive, Type::Negative))
```

$$\text{FALSE}$$

Example 2

`testtype` is used to select positive and negative integers:

```
SET:= {-2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2}:
select(SET, testtype, Type::Union(Type::PosInt, Type::NegInt))
```

$$\{-2, -1, 1, 2\}$$

```
delete SET:
```

Parameters

`obj`

Any MuPAD object

`obj_types`

A sequence of types; a type can be an object of the library Type or one of the possible return values of `domtype` and `type`

Return Values

See `testtype`

See Also

MuPAD Functions
`testtype`

Type::Unknown

Type representing variables

Syntax

```
testtype(obj, Type::Unknown)
```

Description

`Type::Unknown` represents identifiers and indexed identifiers.

The call `testtype(obj, Type::Unknown)` checks, whether `obj` is an identifier or an indexed identifier with an integer index. If it is, the call returns `TRUE`, otherwise `FALSE`.

An identifier has the domain type `DOM_IDENT`. An indexed identifier is an expression with type `_index` and two operands, the first of which is an identifier and the second one is an integer. A local variable is not of type `Type::Unknown`.

This type does not represent a property.

Examples

Example 1

`Type::Unknown` accepts identifiers:

```
testtype(x, Type::Unknown)
```

```
    TRUE
```

`x[0]` is an indexed identifier accepted by `Type::Unknown`:

```
testtype(x[0], Type::Unknown)
```

```
    TRUE
```

The index must be an integer:

```
testtype(x[-1], Type::Unknown),
testtype(x[1.0], Type::Unknown)
```

```
TRUE, FALSE
```

Parameters

`obj`

Any MuPAD object

Return Values

See `testtype`

See Also

MuPAD Functions

`indets` | `testtype`

Type::Zero

Type and property representing zero

Syntax

```
testtype(obj, Type::Zero)
```

```
assume(x, Type::Zero)
```

```
is(ex, Type::Zero)
```

Description

`testtype(obj, Type::Zero )` is equivalent to `iszero(obj)`. `Type::Zero` is a property, too, which can be used in an `assume` call.

The call `testtype(obj, Type::Zero)` is equivalent to `iszero(obj)`, which performs a syntactical test if `obj` is zero. If it is, the call returns `TRUE`, otherwise, `FALSE` is returned.

The call `assume(x, Type::Zero)` marks the identifier `x` as zero.

The call `is(ex, Type::Zero)` derives, whether the expression `ex` is zero (or this property can be derived).

This type represents a property that can be used in `assume` and `is`.

The call `assume(x = 0)` has the same meaning as `assume(x, Type::Zero)`.

Examples

Example 1

`testtype` determines the syntactical equality to zero:

```
testtype(0.0, Type::Zero)
```

```
TRUE
```

```
testtype(x, Type::Zero)
```

```
FALSE
```

Example 2

`Type::Zero` can be used within `assume` and `is`:

```
is(x, Type::Zero)
```

```
UNKNOWN
```

Assumption that `x` is zero:

```
assume(x, Type::Zero):
is(x^2, Type::Zero)
```

```
TRUE
```

The next example shows the difference between `testtype` and `is`:

```
is(x, Type::Zero), testtype(x, Type::Zero)
```

```
TRUE, FALSE
```

Now the property of `x` is removed:

```
delete x:
is(x, Type::Zero), testtype(x, Type::Zero)
```

```
UNKNOWN, FALSE
```

A positive number cannot be zero:

```
assume(x > 0):
is(x, Type::Zero), testtype(x, Type::Zero)
```

```
FALSE, FALSE
```

But in the next example `x` could be zero:

```
assume(x >= 0):
is(x, Type::Zero), testtype(x, Type::Zero)
```

```
UNKNOWN, FALSE
```

```
delete x:
```

Parameters

`obj`

Any MuPAD object

`x`

An identifier or a mathematical expression containing identifiers

`ex`

An arithmetical expression

Return Values

See `testtype`, `assume` and `is`

See Also

MuPAD Functions

`assume` | `is` | `testtype` | `Type::NonZero`